Future Service Oriented Networks
www.fusion-project.eu

Deliverable D3.1

Initial Specification of Algorithms and Protocols for Service-Oriented Network Management

Public report, Version 1.0, 23 January 2014

Authors

UCL	David Griffin, Raul Landa, Miguel Rio
ALUB	Frederik Vandeputte, Luc Vermoesen
TPSA	Dariusz Bursztynowski
SPINOR	Michael Franke, Folker Schamel
IMINDS	Pieter Simoens

Reviewers Dariusz Bursztynowski, David Griffin, Folker Schamel, Frederik Vandeputte

Abstract This deliverable describes the initial functional and non-functional requirements, interfaces and protocols as identified by the FUSION consortium for realizing distributed service orchestration and management across a heterogeneous set of execution zones. We discuss how composite services can be described and managed in the FUSION networking architecture, and present an initial design for the FUSION orchestration domain as well as the execution domain. We also provide a set of initial algorithms and models for efficient service provisioning and scaling.

Keywords FUSION, distributed service management, orchestration, execution, requirements, design, interfaces, algorithms

© Copyright 2014 FUSION Consortium
University College London, UK (UCL)
Alcatel-Lucent Bell NV, Belgium (ALUB)
Telekomunikacja Polska S.A., Poland (TPSA)
Spinor GmbH, Germany (SPINOR)
iMinds vzw, Belgium (IMINDS)

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

Revision history

Date	Editor	Status	Version	Changes
16/09/2013	Frederik Vandeputte	Initial Version	0.1	Initial ToC
08/11/2013	Frederik Vandeputte	Initial Version	0.3	Revised ToC with partial input
02/12/2013	Frederik Vandeputte	Draft	0.4	First stable draft
17/12/2013	Frederik Vandeputte	Stable Draft	0.5	Ready for review
23/12/2013	Frederik Vandeputte	Stable Version	0.6	Review comments included
23/01/2014	Frederik Vandeputte	Final Version	1.0	Final version

Copyright © FUSION Consortium, January 2014

GLOSSARY OF ACRONYMS

Acronym	Definition
Amazon S3	Amazon Simple Storage Service
AMQP	Advanced Message Queuing Protocol
API	Application Program Interface
AWS	Amazon Web Services
BG	BackGround
BLOB	Binary Large OBject
BPEL	Business Process Execution Language
BPMN	Business Process Modelling Notation
BSS	Business Support System
CAMP	OASIS Cloud Application Management for Platforms
CCM	Continuity Check Message
CDF	Cumulative Distribution Function
CDN	Content Delivery Network
CEP	Complex Event Processor
CLI	Command Line Interface
CPU	Central Processing Unit
CS	Composite Service
CSAR	Cloud Service ARchive
DC	Data Centre / Distributed Computing
DHT	Dynamic Hash Table
DM	Deployment Manager
DMA	Direct Memory Access
DNS	Data Naming Service
DRM	Digital Rights Management
DSL	Data Specification Language
EC2	Amazon Elastic Compute Cloud
ED	Execution Domain
ELB	Elastic Load Balancing
Enum	Enumeration
EP	Execution Point
EPG	Electronic Program Guide
ESB	Elastic Block Store
ETSI	European Telecommunications Standards Institute
EZ	Execution Zone
FPGA	Field Programmable Gate Array

Copyright © FUSION Consortium, January 2014

FPS	Frames Per Second
FUSION	Future Service Oriented Networks
GPU	Graphics Processing Unit
HTTP	Hypertext transfer Protocol
HTTPS	HTTP Secure
HW	HardWare
I/O	Input/Output
IaaS	Infrastructure as a Service
ICN	Information Communication Network
INET	Internet
iOS	A mobile operating system developed and distributed by Apple Inc
IP	Internet Protocol
IRMOS	Interactive Realtime Multimedia Applications on Service Oriented Infrastructures
ISONI	Intelligent Service Oriented Network Infrastructure
ISP	Internet Service Provider
ITU	International Telecommunication Union
IVC	Inter-VM Communication
Ivshmem	Inter-VM shared memory
IXB	ISONI eXchange Box
JSDL	Job Submission Description Language
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LXC	LinuX Containers
MAPE	Monitor-Analyze-Plan-Execute functional composition of an autonomic manager
Mgmt	Management
MMO(RG)	Massively multi-player online (role game)
MoSCoW	Must, Should, Could, Wont
MPEG-TS	Moving Picture Experts Group Transport Stream
MPI	Message Passing Interface
NaaS	Network as a Service
NAT	Network Address Translation
NFV	Network Function Virtualization
NGSON	Next Generation Service Overlay Network
OCCI	Open Cloud Computing Interface
OFED	OpenFabrics Enterprise Distribution
OpenVZ	Container-based virtualization for Linux
OS	Operating System

Copyright © FUSION Consortium, January 2014

OSGi	Open Services Gateway initiative
OSS	Operation Support System
PaaS	Platform as a Service
PM	Path Manager
POSIX	**P**ortable **O**perating **S**ystem **I**nterface
QEMU	Quick EMUlator
QoE	Quality of Experience
QoS	Quality of Service
R	FUSION Service Router
RDMA	Remote Direct Memory Access
REST	Representational state transfer
RM	Resource Manager
RoCE	RDMA over Converged Ethernet
RTP	Real-time Transport Protocol
RTSP	Real-time Streaming Protocol
SC	Service Component
SD	Standard Definition
SDK	Software Development Kit
SLA	Service Level Agreement
SR	Service Router
SSL	Secure Socket Layer
STP	Spanning Tree Protocol
STX	Amazon Streaming Protocol
TCP	Transport Control Protocol
TCP/IP	Transport Control Protocol / Internet Protocol
TOSCA	Topology and Orchestration Specification for Cloud Applications
UDP	User Datagram Protocol
URI	Universal Resource Identifier
URL	Universal Resource Locator
VFM	Virtual Network Function Manager
VLAN	virtual local area network
VM	Virtual Machine
VMM	Virtual Machine Monitor
VMPI	Valve Message Passing Interface
VMware	VMware, Inc. is an American software company
VNFM	Virtual Network Function Manager
VOD	Video On Demand
VOIP	Voice Over IP

Copyright © FUSION Consortium, January 2014

VSN	Virtual Service Network
Xen	A type-1 hypervisor
Z	FUSION Execution Zone

Copyright © FUSION Consortium, January 2014

EXECUTIVE SUMMARY

This document is a public deliverable of the “Future Service-Oriented Networks” (FUSION) FP7 project. This deliverable focuses on the initial requirements, design and interface protocols for realizing distributed service orchestration, execution and lifecycle management of demanding interactive services across a distributed network of heterogeneous execution zones.

The scope of this deliverable is threefold. First, it captures the requirements for orchestrating and managing demanding interactive services and execution resources in a distributed manner within and across one or more networking domains. Second, it describes the initial high-level designs and interfaces of the FUSION orchestration, execution and service layers. Third, it provides initial algorithms for dealing with service provisioning as well as initial models for dealing with service scaling by analyzing service usage patterns.

In FUSION, we want to be able to automatically deploy and manage demanding interactive services across a network of heterogeneous execution environments. FUSION services can be either pre-deployed or deployed on demand in optimized locations throughout a network domain for handling both short-lived request/response as well as long-lived interactive demanding applications. To enable this, we envision four key layers, namely an orchestration layer for managing services across execution environments across a networking domain, an execution layer for handling service state management and resource management, a service layer for describing and managing FUSION services, and a networking layer for efficiently routing service requests to the best service instances. The first three layers are described in this deliverable, the networking layer is described in Deliverable D4.1.

This deliverable first describes the initial requirements and constraints related to distributed service management, placement and orchestration in a network domain. It discusses the impact of service composition, parameterization and interactivity constraints onto key orchestration, lifecycle management as well as resource management functions, including deployment, placement, monitoring, heterogeneity, etc. We also discuss the key requirements regarding deploying FUSION orchestration and management on top of existing data centre management infrastructures.

In the second part of the deliverable, each of the key layers (i.e., service layer, orchestration layer and execution layer) are then discussed in detail in separate sections. We start by describing various types of service composition, and their impact on FUSION. We also describe what aspects of the services need to be described and provided to FUSION when registering a new service to enable automated and optimized deployment and management of FUSION services within the FUSION architecture. Next, we introduce the concept of service session slots as a possible solution for dealing with many of the issues regarding scalability and manageability of service instances within the service routing plane. Finally, we provide an initial set of interfaces for managing FUSION services.

Next we discuss the overall architecture and high-level design of the domain orchestration layer, identify and discuss the key functions, and describe the concept of evaluator services for flexible decentralized service placement across multiple execution zones. We also provide an initial set of interfaces for managing service orchestration and deployment within a FUSION domain.

Finally, we discuss the service execution, state as well as resource management, which is the responsibility of a zone manager in a FUSION execution zone. We describe an initial high-level design of a FUSION execution zone and a zone manager and propose an overlay method using a data centre abstraction layer to enable easy deployments of an execution zone on top of various types of data centre management infrastructures. We also focus on service deployment scenarios and the FUSION monitoring infrastructure. We finish this part by describing the initial set of interfaces for managing a FUSION execution zone.

Copyright © FUSION Consortium, January 2014

In the last part of the deliverable, we provide initial models for describing use patterns, which can be used for automatically scaling service instances up or down based on predicted usage patterns. We also provide a methodology and algorithm for service provisioning of atomic services in a decentralized execution environment. We propose an auction-based resource allocation model for this purpose.

In the second year of the project, we will focus on expanding each of the layers, studying the interdependencies and interactions of each layer in more detail, and start refining and validating the high-level designs and interfaces of each layer. We will also study and evaluate the proposed placement, service selection and deployment methods and algorithms in more detail, using the demonstrator setup as a proof-of-concept evaluation platform.

Copyright © FUSION Consortium, January 2014

TABLE OF CONTENTS

GLOSSARY OF ACRONYMS 3

EXECUTIVE SUMMARY 7

TABLE OF CONTENTS 9

1. SCOPE OF THIS DELIVERABLE 13

2. REQUIREMENTS AND CONSTRAINTS 14

2.1 Decision levels 14
2.2 Distributed demanding interactive services 15
2.3 Service composition 16
2.4 Service parameterization 17
2.4.1 When can parameters or parameter values be provided 17
2.4.2 Visibility of the service parameters 18
2.4.3 Parameter classes 18
2.4.4 Impact for the FUSION architecture 19
2.4.5 Summary of requirements 20
2.5 Service registration 20
2.6 Service selection 21
2.7 Service provisioning and deployment 21
2.8 Service placement 23
2.9 Monitoring 27
2.10 Inter-service communication and late binding 28
2.11 Heterogeneous execution environments 30
2.12 Light-weight virtualization and deployment 31
2.13 Security and integrity 31
2.14 Evolvability and backwards compatibility 32

3. RELATED WORK ON DISTRIBUTED SERVICE MANAGEMENT 33

3.1 IRMOS 33
3.1.1 IRMOS architecture and orchestration 33
3.1.2 ISONI infrastructure 35
3.1.3 Virtual Service Network: service description 35
3.2 NGSON 36
3.3 NfV 37
3.4 Amazon AWS 39
3.4.1 Amazon EC2 39
3.4.2 Amazon CloudFormation 39
3.4.3 Amazon AppStream 40
3.5 OpenStack 40
3.6 Service description and orchestration 42
3.6.1 TOSCA 42
3.6.2 Cloudify 44
3.6.3 OpenStack Heat 45
3.7 Light-weight virtualization 46
3.7.1 LXC 47
3.7.2 OpenVZ 48
3.7.3 Docker 48
3.8 Late binding 48
3.8.1 Between user space libraries and system call layer 48
3.8.2 Below system calls layer & above transport layer 49
3.8.3 Below IP layer 49
3.9 Monitoring 50
3.9.1 Amazon CloudWatch 50

Copyright © FUSION Consortium, January 2014

3.9.2 Ceilometer ... 50

4. FUSION SERVICE SPECIFICATION ... 52

4.1 Service composition ... 52

4.1.1 Service graphs: describing composite services ... 52

4.1.1.1 Variant A: Instantiating service graphs ... 52

4.1.1.2 Variant B: Instantiate and integrate service graphs ... 53

4.1.1.3 Variant C: Stepwise instantiation and integration of service graphs ... 54

4.1.1.4 Variant D: Instantiation of individual services ... 55

4.1.1.5 Variant E: Future-looking instantiation of individual services ... 56

4.1.2 Granularity ... 58

4.1.3 Deployment strategies ... 58

4.1.4 Service binding & lifecycle management ... 59

4.1.5 Distribution ... 59

4.1.6 Addressing composite services ... 60

4.1.7 Level of interactivity ... 61

4.1.8 Constraints ... 61

4.2 Service description ... 62

4.2.1 General information ... 62

4.2.2 Service graph ... 62

4.2.3 Deployment and platform dependencies ... 62

4.2.4 Lifecycle management ... 63

4.2.5 Resource usage ... 63

4.2.6 Monitoring ... 63

4.2.7 Mapping ... 63

4.2.8 Policies ... 63

4.2.9 Security and privacy ... 64

4.2.10 Business aspects ... 64

4.3 Service sessions ... 65

4.3.1 Service Session Implementation ... 65

4.3.2 Benefits ... 66

4.3.2.1 Light-weight service request routing ... 66

4.3.2.2 Separation of resource allocation and service request routing ... 66

4.3.2.3 Hierarchical aggregation ... 66

4.3.2.4 Service-type neutral ... 67

4.3.2.5 Stability ... 67

4.3.2.6 Auto-scaling ... 67

4.3.2.7 Billing ... 67

4.3.2.8 Trust ... 67

4.3.3 Issues for further investigation ... 67

4.4 Inter-service communication and late binding ... 69

4.5 Service management interface ... 70

4.5.1 Lifecycle management interface ... 71

4.5.2 Inter-service interface ... 74

4.5.3 Resource management interface ... 75

4.5.4 Routing and networking interface ... 76

5. DISTRIBUTED SERVICE ORCHESTRATION ... 77

5.1 Functional design ... 77

5.1.1 Overall design decisions ... 77

5.1.2 High-level design ... 77

5.2 Key FUSION orchestration actors ... 79

5.3 Lifecycle management of the FUSION orchestration services ... 80

5.4 Key functions ... 80

5.4.1 Service registration ... 80

5.4.2 Service placement ... 81

5.4.2.1 Four-step service placement ... 82

5.4.2.2 Evaluator services ... 83

5.4.2.3 Execution zone resource allocation 88
5.4.3 Service scaling 88
5.4.4 Service deployment 90
5.4.5 Service and resource monitoring 90
5.4.6 Inter-domain orchestration 91
5.4.7 Service routing 93
5.5 Management interfaces 94
5.5.1 Orchestration management interface 95
5.5.1.1 Service registration interface 95
5.5.1.2 Service querying interface 97
5.5.1.3 Service deployment interface 98
5.5.1.4 Accounting/billing interface 99
5.5.1.5 Security interface 99
5.5.2 Execution zone management interface 99
5.5.2.1 Zone registration interface 99
5.5.2.2 Monitoring interface 101
5.5.3 Inter-orchestration management interface 102
5.5.3.1 Inter-domain routing management interface 102
5.5.4 Routing and networking management interface 104
5.5.4.1 Router configuration interface 104
5.5.4.2 Monitoring interface 106

6. DISTRIBUTED SERVICE EXECUTION MANAGEMENT 107

6.1 Functional design 107
6.1.1 Overall design decisions 107
6.1.2 High-level design 108
6.2 Lifecycle management of an execution zone 109
6.3 Key functions 110
6.3.1 Selecting a zone for service deployment 110
6.3.2 Service placement 111
6.3.3 Service deployment 112
6.3.3.1 Service deployment scenario 112
6.3.3.2 Service lifecycle management 114
6.3.3.3 Optimizing service placement and deployment 116
6.3.4 Monitoring 117
6.3.4.1 Probing 117
6.3.4.2 Aggregation 118
6.3.4.3 Reporting 118
6.3.4.4 Service and Resource Monitoring Architecture 118
6.3.5 Data centre abstraction layer 119
6.3.6 Service gateway 120
6.3.7 Light-weight virtualization and deployment 120
6.4 Management interfaces 122
6.4.1 Orchestration management interface 123
6.4.1.1 Zone selection 123
6.4.1.2 Service lifecycle management 124
6.4.1.3 Monitoring 125
6.4.2 Routing management interface 126
6.4.3 Service management interface 126
6.4.3.1 Service lifecycle management 126
6.4.3.2 Service monitoring 127
6.4.4 Data centre management interface 127
6.4.4.1 Service DC environment 127
6.4.4.2 Monitoring 129
6.4.4.3 Security interface 131

7. INITIAL SERVICE PROVISIONING ALGORITHMS 132

7.1 Usage patterns 132
7.1.1 Service popularity 132

7.1.2 Geographic locality 133
7.1.3 Variations over time 134
7.1.4 Session length (Service holding time) 135
7.1.5 User arrival rate 136
7.2 Inter-zone service placement 136
7.2.1 Introduction 136
7.2.2 Architecture 137
7.2.3 Problem definition 138
7.2.4 Evaluation 139
7.2.4.1 Simulator setup 139
7.2.5 Related work 140
8. CONCLUSIONS 142
9. REFERENCES 143

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. We start by describing the key requirements for distributed service management, followed by a high-level decomposition and design of each layer and an initial description of the key management interfaces and protocols.

In the first year of the project, we focused on the key requirements and constraints related to the automated deployment and management of demanding interactive services across a heterogeneous network of execution environments for deploying such services. We studied the impact of service composition and parameterization on service instantiation and service selection. Then we identified the key functions of each layer, followed by an initial high-level design of each layer. We also started working on a number of strategies and algorithms regarding efficient service placement, service scaling, and service deployment. We also identified already initial high-level interfaces at each layer for managing the services, the resources within a FUSION domain as well as within the FUSION execution zones.

Each of these requirements, interfaces, designs and algorithms will be evaluated and expanded in the second and third year of the project. Each of the functions will be described and modelled in more detail, and the key interfaces will be worked out in more detail and evaluated with real FUSION deployment scenarios. We will rely on the initial demonstrator design and dedicated test cases for feedback regarding the effectiveness of the design and interfaces, and use this as input for updating and tuning each of the key management and orchestration components.

Copyright © FUSION Consortium, January 2014

2. REQUIREMENTS AND CONSTRAINTS

In this section, we will describe the key functional and operational requirements and constraints that the FUSION architecture should support with respect to the services, the orchestration layer and the execution environments. We will focus on a number of key issues and start by highlighting the various options that we could consider, and discuss what direction we plan to take with FUSION for managing real-time demanding services distributed across a set of heterogeneous execution resources with varying network constraints. At the end of each topic, we explicitly list the key requirements and how FUSION should or must be able to handle these requirements and scenarios.

2.1 Decision levels

FUSION enables a very dynamic distributed deployment and management of services and routing of service requests to these services. Each of these aspects involve a number of key decisions that can be made by different components at different moments in the life cycle of the services. At one end of the spectrum, these decisions can be made statically upfront by the service provider. At the other end of the spectrum, these decisions could be made very dynamically, involving multiple components to make the final decision. Different service and content distribution systems handle these decisions at different levels. In this section, we will briefly give an overview of the various decision levels, and compare the FUSION architecture with a number of existing service and content distribution systems.

- Level 0: none, external decision

  The decision is made up front, outside the infrastructure in which the services are running. For example, a classic webhosting company managing their own services decides how many instances to deploy to handle the incoming requests. The decision to spawn extra instances to handle the increased load is done externally by the webhosting company.

- Level 1: static, preconfigured or random decision

  In this case, the decision is preconfigured and done in a rather static manner. For example, the webhosting company registers its host address(es) to a DNS service. These registered IP address (or addresses) are then used when resolving a domain name.

- Level 2: making the decision involves the network level

  The decision to for example create a new service instance or to select an existing instance is done on-the-spot at the network level or service routing level when a service request comes in or when the network decides to scale some services based on predicted demands.

- Level 3: making the decision involves the orchestration level

  The decision to for example create a new service instance or to select an existing instance is done on-the-spot and involves (also) a domain-level orchestrator that manages a distributed set of resources and services. In this case, the network layer involves an orchestration layer to help making the decision.

- Level 4: making the decision involves the application level

  The decision to for example create a new service instance or to select an existing instance is done on-the-spot and involves application software. This involves ultimate flexibility, as this means application-specific services are involved in making the final decision of where to deploy a new instance or which service instance to route a particular request to. An example of the former is the concept of evaluator service that we introduce in FUSION, which helps the orchestrator to decide where and how to deploy new instances in a particular execution zone.

Copyright © FUSION Consortium, January 2014

As an example, we illustrate for a number of platforms and architectures at what level of flexibility the decision for selecting an instance to route a request to and for instantiating new services is performed. This is shown in Figure 1 for two key functions, namely service instantiation and service selection. In FUSION, we envision that the decision where and when to deploy new instances could be triggered at the network level, the orchestration level and may also involve the application level (for example, by means of evaluator services). Although obviously FUSION can just as well handle the more static instantiation scenarios, we mainly envision the more dynamic and flexible instantiation scenarios for dynamically deploying new instances in an optimal location. Regarding service selection, we envision FUSION to mainly operate at the networking and orchestration layer for automatically selecting an optimal instance, based on for example geographical, network or other requirements.

Comparing that to a classic cloud infrastructure, it is obvious that FUSION is targeting much more dynamic service selection and instantiation scenarios. In classic cloud deployments, services are typically deployed explicitly in a rather static manner, and rely on the elastic cloud scaling functionality for automatically scaling up and down new instances inside the cloud environment. Service selection is typically handled by an elastic load balancer acting as a front-end towards one or more instances deployed across the cloud infrastructure. The service requests are typically balanced either randomly or based on the load in each instance, rather than based on for example geographical or network related metrics. Similar comparisons can also be made for other service deployment and selection architectures, as also discussed on more detail in the related work sections.

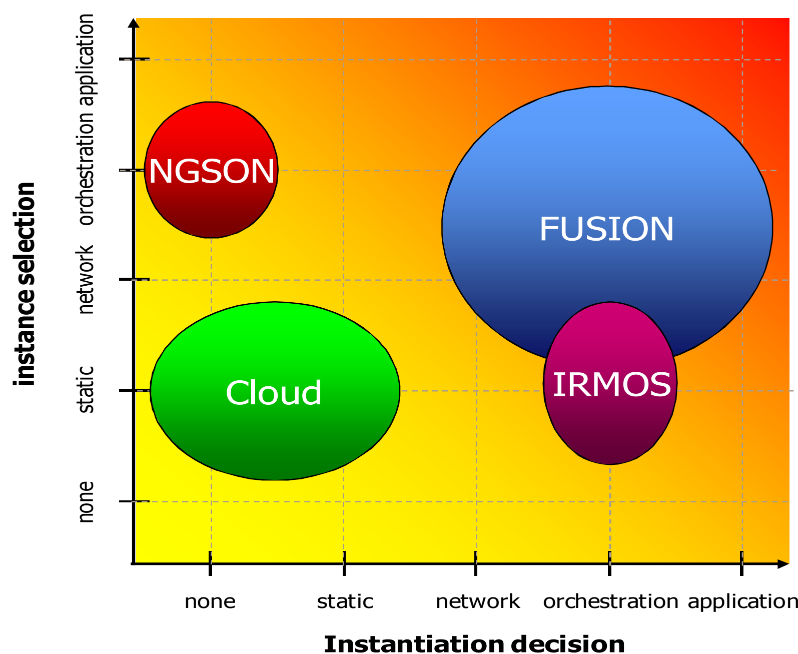

Figure 1: Dynamicity of service selection and instantiation for various frameworks

2.2 Distributed demanding interactive services

Within the FUSION project, we are targeting applications that go far beyond the static content distribution or classic three-tier web applications. We are targeting interactive demanding real-time applications that often have very specific requirements with respect to their execution environment, geographical deployment, etc. We refer to Deliverable D5.1 for a more detailed description of these requirements.

Copyright © FUSION Consortium, January 2014

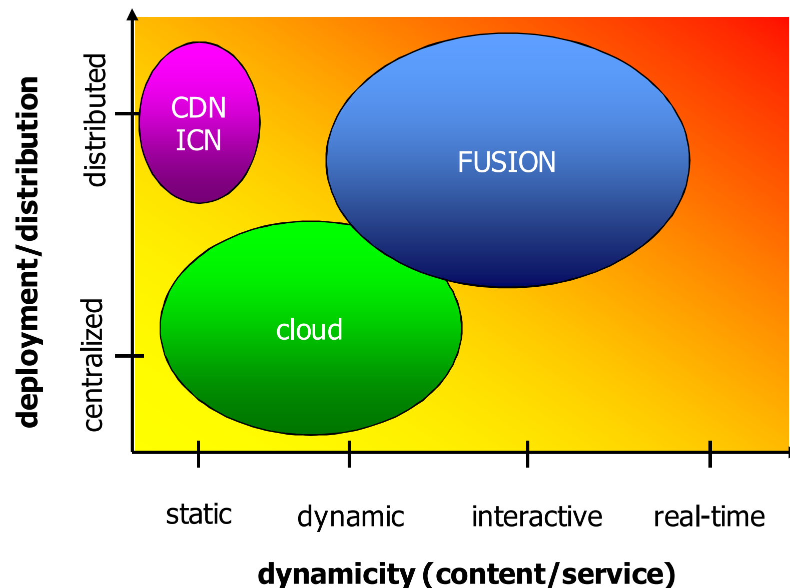

Figure 2: Comparison of FUSION related to other frameworks regarding service dynamicity and service or content distribution

Req no.	Description	Level[1]
DS-1	Support for automatically placing, deploying and scaling distributed composite services	M
DS-2	Support for demanding interactive services that are automatically deployed and managed across distributed execution zones	M

2.3 Service composition

In FUSION, we want to be able to support composite services that are deployed in a distributed manner across multiple heterogeneous execution zones that are spread across the network. This allows service providers to optimally take advantage of the network and compute resources offered by a FUSION orchestrator (e.g., an operator), enabling higher QoS guarantees towards these services. For example:

- More latency-bound sensitive services that require close interaction with the end users and/or are very personalized for a particular user can be deployed in a very distributed manner on smaller execution zones with limited resources near the access network of the end user.
- Common service components, data bases, etc., which are less sensitive or which require more centralized computing capabilities and/or which are shared amongst many end users can or should be deployed more centralized in the network, for example in large data centres with virtually infinite compute and internal networking resources.

Secondly, this also allows multiple services to work together to implement a desired high-level functionality. For example:

[1] The abbreviations indicate the level of importance and follow the MoSCoW method. http://en.wikipedia.org/wiki/MoSCoW_Method

Copyright © FUSION Consortium, January 2014

- A game server service has to communicate with multiple cloud game clients to implement a multi-player game scenario.
- An EPG service streaming personalized content to a thin client, but itself receiving video streams from various other sources.
- A dashboard presenting different media like photos, private videos, VOD mash-ups, and video chats needs a variety of service components connected together.

Req no.	Description	Level
SC-1	Optimized and automated deployment and management of different types of static and dynamic service graphs that are distributed across heterogeneous execution zones	M
SC-2	Support for personalized and multi-user collaborating related services	M

2.4 Service parameterization

When deploying or interacting with a particular service, it may be necessary or desired to be able to customize the runtime behaviour of a service instance to the requirements and needs of a particular customer or end user, without having to manage each particular flavour as a completely different service type. In this section, we will discuss some of the possibilities and implications of allowing different levels of parameterization at different stages in the lifecycle of a service instance and for selecting a particular service type.

2.4.1 When can parameters or parameter values be provided

Service parameter values can be provided at several stages:

- At design time

  A service developer can specify a number of service-level parameters and default values that will be applied later when deploying or accessing an instance of that service.

- At deployment time

  In FUSION, a deployment of a new service instance may be issued by different entities. For example, a FUSION domain orchestrator may decide to automatically deploy new instances based on predicted service load or active monitoring information. For this, it may use specific parameter values, provided by the service provider that registered the service into a FUSION domain. New services instances may also be deployed explicitly or implicitly by other service instances, potentially with very specific deployment parameters (e.g., resolution, input sources, etc.). Lastly, it may also be explicitly triggered by a service provider that instructs FUSION to deploy specific instances, possibly at specific locations and moments in time, with specific deployment parameters.

- At request time

  When a client makes a FUSION service request to initiate a new session, it may provide a number of parameter values to customize the session (e.g., encoder type, resolution, frame rate, etc.). This may or may not impact the selection of a particular service instance to host that specific session (e.g., some instances may only support SD quality or H.264 encoding).

- During an active service session

  As we are targeting interactive services, the client will typically impact the behaviour of the service sessions through one or more feedback channels. This could involve changes in the

Copyright © FUSION Consortium, January 2014

dynamic service instance graph, for example when the client selects a different video channel or starts a new game session from a dashboard.

A key question is to what extent FUSION needs to be aware of some or all of these parameters, and to what extent it may impact some of the instantiation, placement or even service selection protocols. We will discuss this topic in the next sections.

2.4.2 Visibility of the service parameters

A key question is how FUSION should treat these parameters. Three options are possible:

- All service parameters are invisible to FUSION

  In this case, the service parameters are merely seen as a blob of data from the perspective of FUSION. FUSION does not care about these parameters nor does it need to understand the underlying format and protocol used by the service type and the invoker (which can be the service provider, another service, the client, etc.). On the downside, it also means FUSION cannot directly leverage some of these parameter values as input in its decision process (e.g., during placement, deployment, service routing, etc.).

- All service parameters are visible to FUSION

  In this case, all service parameters are visible to FUSION, and FUSION can use some of these parameters to make a better decision where to deploy a new service instance or to what instance to route a service request. However, for FUSION to understand these parameters, they must all be represented by a standardized scheme, possibly making service parameterization much more complex or constrained in terms of parameter types, formats, etc.

- Hybrid parameterization scheme: some parameters are visible, whereas others are not

  In this model, some of the parameters are visible and understandable by FUSION and others are not. This thus provides a combination of both service-specific and more FUSION-specific service parameters; in this case the responsibility for the selection of service type and/or service instance that meets the requested parameter values is left up to the application layer.. The former can be very application-specific and FUSION does not care about these parameters. The latter consist of parameters that may be relevant for FUSION in making decisions.

In the initial architecture and design of FUSION, we assume that all service parameters are invisible to FUSION components. This immediately also implies that none of these parameters will be able to directly impact some of the key functions of FUSION, like service placement, service deployment and service routing. Indirectly, these service parameters may still influence for example service placement, by means of the evaluator services, that are capable of interpreting these service-specific parameters and that may impact in what execution zone a new service would be deployed.

2.4.3 Parameter classes

Assuming some of these parameters are visible to FUSION, we envision a number of key parameter classes that may provide valuable information towards the service instance as well as FUSION.

- Functional parameters

  This is a set of mostly very service-specific parameters that may directly impact the overall operation of that instance while handling one or more service sessions.

- QoS parameters

  Potential QoS parameters may involve bandwidth and latency requirements, jitter, traffic profiles, resolution, frame rate, visual quality, etc.

- Business parameters

Copyright © FUSION Consortium, January 2014

This includes parameters like cost, licensing aspects, etc. For example, some clients may only have access to low-cost instances that offer lower visual qualities or lower overall QoE, whereas other clients may pay for accessing premium instances that offer the best QoE, but at a higher cost.

- Geographical parameters

  As we are targeting low-latency demanding interactive service, the location of particular service instances with respect to the location of other communicating services (cfr., the service graph) or the application client may be of key importance and will likely impact service placement and service selection decisions.

- User and session parameters

  This set consists of parameters that involve user identification and authorization, user preferences and its environment (e.g., co-players, family members also involved in the same application, a multi-person video conference, etc.)

- Security related parameters

  These parameters may involve encryption, DRM, and other more generic security related parameters.

- Other non-functional types of parameters

  There may be other types of parameters that are also relevant for particular service types or service deployments or service sessions.

2.4.4 Impact for the FUSION architecture

As already mentioned above, each of these parameter values may impact some of the core FUSION functions. For example, it may impact where and how an instance is deployed, what instance to choose from (e.g., based on distance, cost, etc.). The core FUSION functions alone may not always be able to interpret what impact it may have at the orchestration layer or the service routing layer, and therefore could rely on evaluator services in assessing the situation and assisting FUSION in making an optimal decision. Obviously, relying on external evaluator services may have an impact latency, so a key aspect will be to decide to what extent and where we use these evaluator services for making such assessments. This aspect is discussed in more detail in other sections that involve the evaluator services.

For the initial design of FUSION, we take rather conservative approach, where we only want parameters to have impact on FUSION when strictly required. We obviously want to take into account geographical information during deployment and service request routing, as well as take into account the capabilities of an execution environment when deploying a particular service using evaluator services.

For other parameters, we will assume for now that they will not impact FUSION with respect to service routing, service placement or service placement. With this strategy, a service provider then needs to register services with specific requirements as separate service types, in which case the parameterization will be done implicitly by selecting the appropriate service type. For example, low cost instances and premium instances of the same service in this scenario can be modelled as separate service types. The disadvantage of this strategy is that it restricts the overall flexibility of the solution, or may result in an explosion of similar service types that all have to be registered separately. Note that the latter issue could be partially alleviated by providing an automatic intelligent multi-service registrar service, where a service provider only has to deploy one service type, which is then automatically translated into a series of service types.

Copyright © FUSION Consortium, January 2014

2.4.5 Summary of requirements

Below a summary of the initial set of requirements related to service parameterization.

Req no.	Description	Level
SP-1	Allow for service parameterization at design time	M
SP-2	Allow for service parameterization at deployment time	M
SP-3	Allow for service parameterization at request time	M
SP-4	Allow for service parameterization during a service session	M
SP-5	Require all service parameters to be visible to FUSION	W
SP-6	Allow FUSION-visible service parameters at all levels	C

2.5 Service registration

In FUSION, we want multiple service providers to be able to register and deploy services into a FUSION orchestration and routing domain, enabling internal and external service developers and providers to leverage the available infrastructure and QoS capabilities that an orchestration domain offers. This enables on the one hand ISPs to easily and quickly deploy new service types onto their own infrastructure, but also allows external entities to deploy new service types without having to invest in the corresponding compute and networking infrastructure. Consequently, FUSION requires a service registrar where services can be registered or unregistered. To enable automated (pre)deployment and management of these services, these services should be fully contained and self-describing so that FUSION knows its requirements, policies as well as deployment information. As part of the service registration procedure, this may automatically trigger FUSION to start deploying instances of the new service type and start configuring the FUSION routing plane so that clients can immediately start accessing the new (or updated) service type.

Req no.	Description	Level
SR-1	Allow internal service providers to (un)register new services at any moment in time	M
SR-2	Allow external service providers to (un)register new services at any moment in time (depends on business model)	S
SR-3	The service description contains all requirements, dependencies and policies to be able to automatically deploy and manage the service	M
SR-4	After service registration, FUSION needs to automatically make the new service accessible within a FUSION domain. This may include automatically deploying instances of the new service, and configuring the service routing plane.	M

Copyright © FUSION Consortium, January 2014

2.6 Service selection

One of the key functions of FUSION is automatically selecting the best instance for each individual service request. Depending on the service type, service selection may be done at different levels. For some service types, service selection can be done by the FUSION routing plane, which then may directly forward the request to the selected instance or merely return its corresponding locator.

For more complex services like a multi-player game, this is not always feasible, as multiple users may want to join the same game session, which means they have to be able to specifically join the same game server that is hosting the game session. In this case, the optimal selection of a common game server could be done via the FUSION orchestration plane, which needs to take into account also the location of the game client services of the involved users.

For long-lived sessions with very specific requirements or parameters, evaluator services could be used for helping a FUSION orchestration domain in finding a proper service instance.

Req no.	Description	Level
SS-1	Network-driven service selection	M
SS-2	Orchestration-driven service selection	M
SS-3	Application-driven service selection	S

2.7 Service provisioning and deployment

Cloud is primarily driven by economics and hence a service provider aims to accommodate as many requests as possible with the objective to maximize profit. Pre-provisioning, which is the act of reserving the necessary resources in advance without already necessary using them, allows for faster re-action times for handling future service requests. Consequently, the cost for pre-provisioning is amortized by the benefits for respecting the service SLAs regarding the response time for handling the service requests.

A challenge of cloud computing is the management of optimal allocation strategies for computational resources for various types of service requests. It is essential that the trends of different request streams are identified to organize pre-allocation strategies in a predictive way. This calls for designs of intelligent modes of interaction between the client request and cloud computing resource manager. See also [GOPA11].

In the case of on-demand deployment, the resource utilisation is increased since resources are only deployed when needed. However, on-demand deployment induces additional stress on several levels:

- Stress in timeliness
    - On the level of mapping algorithms whereby actual state of the ecosystem is taken and the on-demand request is mapped onto the available resources. This mapping problem is an NP complete problem for which a timely response is expected to have reasonable request-response times. Moreover, due to the parameterisation in FUSION on several levels (manifests, topology, heterogeneous HW etc), the complexity for the mapping increases.
    - On the level of deployment itself, notably concerning the speed with which one can deploy new instances of a service. For example, the start-up time of container-based virtualization

Copyright © FUSION Consortium, January 2014

versus virtual machine start-up, the overhead for transporting and installing all software packages or VM images, starting the application services, installing monitoring probes, etc.

- On the interactivity of FUSION control signalling (soft real-time interactive versus batched).
- On the propagation of current state metrics of the executing layer with propagation delay.
- Quality of service deployment, any impact on actual deploying of resources (e.g. lack of resources due to transient state differences) and need to re-deploy, possibly re-orchestrate, impacting time (current is batch processing).
- On the provisioning of necessary networking resources.

- Stress in optimized resource usage
  - For example in case a service request is received, a new instance may have to be deployed or an existing instance or deployment may be reused. In other cases, services could be co-located, depending on their processing and resource characteristics.

In case of pre-deployment, resources are reserved and used in advance, mainly to reduce service request-response delays. Different stages of pre-deployment are possible, including:

- Physical server start-up but no deployment of any specific service.
- Pre-deployment of the service software onto the execution environments that will host the service, but without already starting the service.
- Pre-deployment of the runtime components of a service towards different servers that will host the service.

Pre-provisioning and pre-deployment require an assessment of near-future service utilisation and corresponding execution layer resource requirements whereby resource consumption and cost are highly dependent on an accurate prediction of service and corresponding resource consumption over time. This requires service utilisation monitoring in order to learn service request patterns.

Note that one should make a distinction between pre-fetching, caching and pre-provisioning. In case of pre-fetching, this indicates fetching service software prior to a request. This induces extra network and storage cost in case of inefficient pre-fetching. Caching on the other hand is a post-deployment optimization where portions of a service deployment are maintained for a period of time although the service is currently no longer used. This serves as a service pre-allocation for future service requests. Caching may induce a cost related to compute, network and storage resources in case of inefficient caching strategies or predictions. Pre-provisioning as defined here is the deployment of a service up to a running state. In case of pre-provisioning, the following parameters could be taken into account:

- Time to get up and running
- Used computing resources and its induced cost
- Due to the reservation of compute resources, other services cannot always use these resources, so there may be an important impact on revenue and cost models.

From the above, pre-deployment has a positive impact on:

- Faster service deployment
- Predictability and stability of the entire ecosystem in case or qualitative prediction
- Load balancing benefits within an orchestration domain or execution zone
- Monitoring for service routers

Copyright © FUSION Consortium, January 2014

There are a number of mixed forms possible for on-demand and pre-provisioning, based on a number of criteria:

- Service types: some service are pre-deployed, others deployed on demand. Different criteria can be used to make segregation (e.g. service request-response time, service deployment timing, etc.)
- Mixed forms of virtual pre-provisioning whereby one makes an estimation of the amount of services that can be supported by an execution zone without actually pre-deploying the resources. This technique allows to plan in advance resource allocation but induces compartmentalisation of physical resources and ultimately sub-optimal cost and revenue management.
- Extending the previous option to allow the correction of virtual allocation of services and resources on an on-demand basis is likely to reduce compartmentalization as well as improving resource utilisation.

Req no.	Description	Level
SPD-1	Fully automated service provisioning, deployment and instantiation	M
SPD-2	Pre-deployment of services based on predicted service demands	M
SPD-3	Automatic scaling of FUSION services across a domain	M
SPD-4	On-demand deployment of FUSION services based on service requests	M

2.8 Service placement

This section discusses some of the complexities, requirements and constraints regarding service placement and selection of related services.

Related services are services that communicate with each other. A common aspect of related services is that performance aspects (e.g. network connection QoS related indicators) depend not only on the location and indicators of individual service instances (e.g. the location of service instances, for example of a game server), but also on their relative location (for example whether a game server and game clients are close to each other regarding network QoS metrics). In other words, in many of these cases the best location of each service instance depends on the location of related service instances.

Often these dependencies are circular, meaning that the best location of one service instance depends on the location of another service instance, but the location of that other service instance depends on the location of the first service instance.

For example, suppose the following situation, visualized in Figure 3: two users are connected to the internet via the same ISP with a very low-latency last mile. The ISP is connected to three data centres A, B and C. Data centres B and C are very well connected, for example because they are merely different groups of servers belonging to the same physical data centre. The ISP and data centre C has Linux based servers without GPUs, while data centres A and B have Windows based servers with GPUs, A having better GPUs than B.

Copyright © FUSION Consortium, January 2014

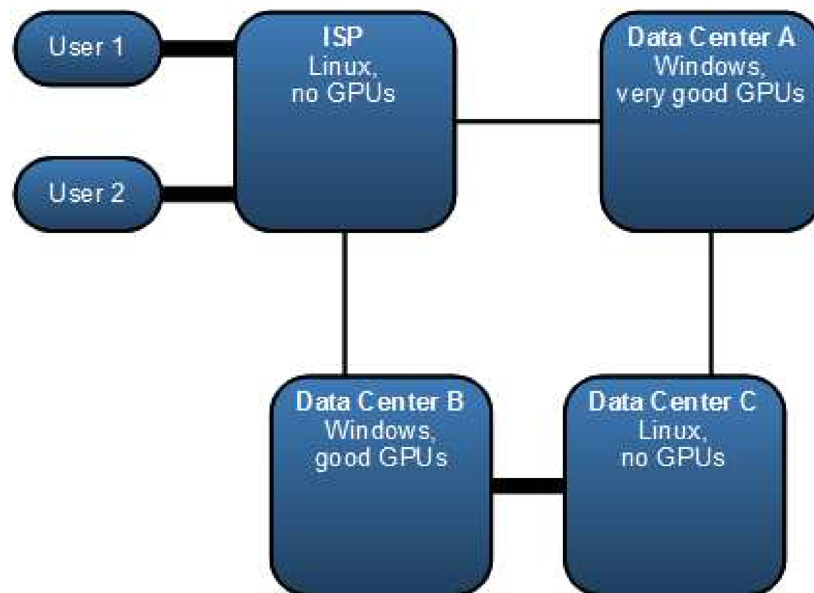

Figure 3: Example of a trade-off between capabilities and requirements

Now suppose we want to deploy a multi-player game consisting of a Linux-based game server connected to two Windows-based game clients, each of which are connected to one of the users running a thin client:

FUSION Service	Constraint: Required OS	Parameter to optimize
Game Server (S)	Linux	Close to the clients and close to the users
Game Client 1 and 2 (C)	Windows	Proximity to the user is important. Better GPU is nice, but in this scenario we assume that being close to the server is more important than a better GPU.

Now let us consider different options for how FUSION orchestration could operate.

1. Suppose the orchestration first chooses the best server location alone without taking possible locations of the client into account. The best server location obviously is the ISP, because it is closest to both users. Once the server location is chosen, then the clients must be placed in A or B, and the better choice is obviously A because of the better GPUs. As a result, the connection between the servers and the clients is not optimal, see Figure 4.

Copyright © FUSION Consortium, January 2014

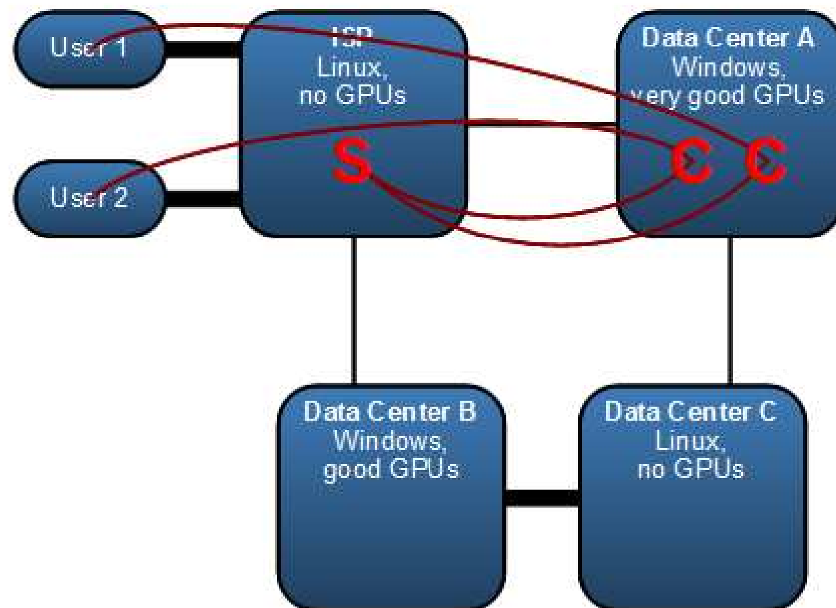

Figure 4: Independently deploying the game service before the game clients

2. If the orchestration first chooses the best locations for the clients, but without taking possible locations of the server into account, then the best choice for the clients would be A, because A is as far away from the users as B, but A has better GPUs. Once the locations of the clients are chosen, then the server must be placed at the ISP or at C. In the first case we get the same situation as before, and in the second case we get an equally bad situation, where the connection between the servers and the clients is bad, as is depicted in Figure 5.

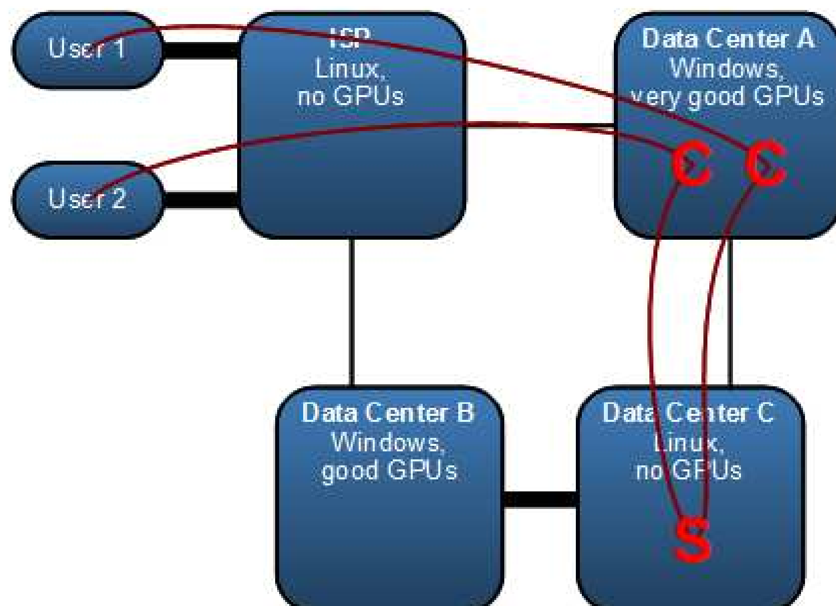

Figure 5: Independently deploying the game clients before the game server

Copyright © FUSION Consortium, January 2014

3. Now let us assume the orchestration can choose the best locations for the server and the clients together by considering all possible combinations of locations. Then the best choice may be the following, because our assumption was that the better connection between server and client may be much more important than slightly better GPUs, see Figure 6.

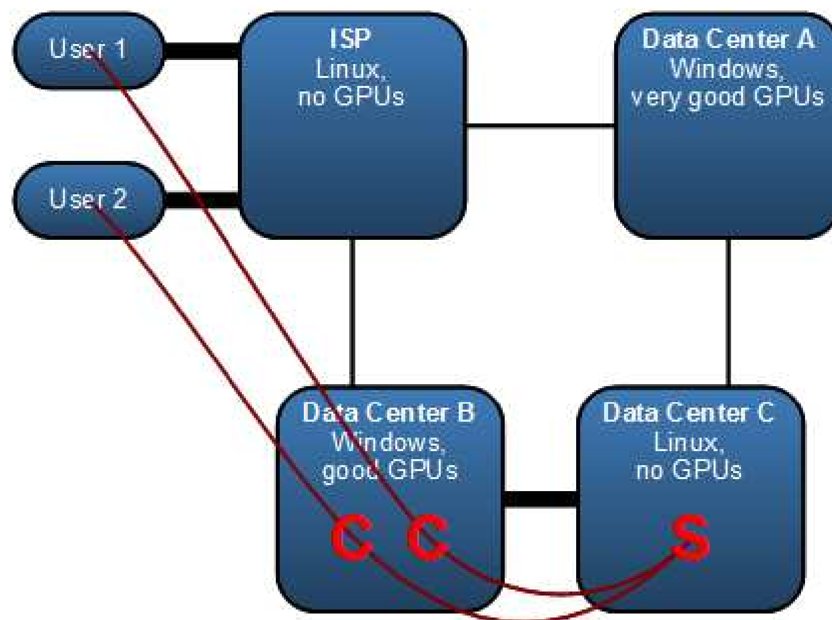

Figure 6: Deploying the game service and game clients together

Of course, there may also be situations where GPU performance is more important than the connection between the server and clients. The purpose of this particular example is to demonstrate fundamental limitations of a simple orchestration approach of making the decision of the placement of service instances in a service graph one after the other. The conclusion from this exercise is that choosing the best locations for service instances only by considering the best location of each individual service one after the other can lead to worse results in realistic situations, leading to worse performance.

The above situation is one example for this. Other motivations for having a placement algorithm that considers more than only individual service instances, may include:

- It may be beneficial to run both the game server and the game clients on the same execution point (EP) to take advantage of a fast shared-memory communication, even when there are more optimal EPs for either client or server, but none of them can run both client and server in the most optimal way.
- An augmented reality service needs a service providing location-based information and must connect to a particular database. Then the best location of the augmented reality service does not only depend on the distance to the user and to the database, but also on the possible locations of the service providing the location based information.

The challenge is that the FUSION orchestration placement function, which wants to determine the best locations for such a set of related services, may need to rely on service-specific evaluation parameters, which is a non-trivial problem.

Copyright © FUSION Consortium, January 2014

Req no.	Description	Level
SP-1	Independent placement of service atoms across execution zones	M
SP-2	Combined placement of all components of a service graph across execution zones	S
SP-3	Placement taking into account resource type availability	M
SP-4	Placement taking into account application-specific scores	M
SP-5	Placement taking into account current and forecasted demand	M

2.9 Monitoring

Currently there are already different distributed monitoring systems available (e.g. ganglia). For the FUSION project, different roles, functionalities and decisions can be distinguished:

- Actual service monitoring
- FUSION network routing monitoring
- Service usage monitoring
- Resource utilisation monitoring
- Aggregation of monitoring information
- Reporting of monitoring information
- Propagation of monitoring information (scalability and security)

In the FUSION project, the focus will be on the execution node platform monitoring, network monitoring and service monitoring. Concerning data collection and from an execution layer point of view, monitoring can apply to either the application services, the guest OS, hypervisor, host OS as well as all hardware resources (CPU, memory, networking, accelerators, etc.)

It is on the latter topic that there is a significant degree of complexity due to the presence of heterogeneous HW in possibly different data centres. Furthermore, after having collected the monitoring data from multiple locations using various probes, another challenge is how this information can be combined into aggregated metrics that can be used by the zone manager, the domain orchestrator or networking plane. In the FUSION project, we will investigate how fine-grained metrics can be used to improve orchestration and placement.

It is the intent to evaluate an ecosystem whereby the different components that contribute to monitoring can be described in a configurable manner and allow to be combined via scriptable logic. The aggregated monitoring data will be used as input to the zone manager, FUSION domain orchestrator and routing domain, the placement algorithms, operational management and billing as well as for SLA monitoring towards the service and infrastructure providers (i.e., the execution zone and routing plane administrators).

Copyright © FUSION Consortium, January 2014

Req no.	Description	Level
M-1	Monitoring of available session slots	M
M-2	Monitoring and aggregation of general application-specific monitoring data	M
M-3	Monitoring of execution zone resources	S
M-4	Aggregation of service and resource monitoring information	M
M-5	Enable scriptable monitoring logic	S

2.10 Inter-service communication and late binding

Depending on the service placement, two communicating services may have been deployed on the same (physical or virtual) machine or another, either inside the same execution zone, across multiple execution zones or across execution domains (inter-domain late binding). Depending on what option was selected (for whatever reason), the optimal communication protocol and data transport mechanism may vary. For example, if two communicating services are collocated on the same machine, they may pass raw data directly via shared memory. If the same two services are placed on different machines, potentially on different execution zones or domains, the data could be transported over standard TCP/IP with varying underlying data transport protocols (e.g., standard Ethernet, RDMA, etc.). Additionally, it may be necessary to add one or more transformation functions in order to:

- Minimize network resource consumption (e.g., to save bandwidth using H.264 transcoding);
- To increase overall security (SSL encryption/decryption), both at the expense of increased compute resources and increased end-to-end latency.

As a result, FUSION should support a late binding mechanism for services, to enable using an optimal communication channel depending on the chosen service placement distribution and physical mapping. Next to this, FUSION should also allow services to implement proprietary communication channels, to be able to support custom and dedicated communication mechanisms. An example is the efficient sharing of GPU buffer pointers across multiple service instances running on the same machine to avoid expensive copying and wasting huge amounts of bandwidth. As this is a very dedicated protocol, it is too specific for FUSION to support this out-of-the-box. However, FUSION should make it easy for services to detect these scenarios and deploy their own communication mechanisms via custom libraries in a reliable and secure way. In case of inter-domain late binding, orchestration is mainly involved as a mediator and enabler.

Different communication technologies already exist today, which can be distinguished in a number of ways based on different criteria, for example, based on proximity:

- On board communication: shared memory communication, Unix domain sockets, DMA transfer, IP loopback interface, sockets, Unix pipes, ...
- Off board communication: RDMA, RoCE, OFED, IP related protocols such as sockets HTTP, TCP, UDP, RTP, etc.

Each of these technologies have their specific APIs, synchronization methods for sending data from one location to another.

Different components contribute to a communication between collaborating services such as specific HW adapters, host OS and installed protocol stacks, kernel modules, user space libraries enabling

Copyright © FUSION Consortium, January 2014

specific communication technologies, etc. Different options for late binding with respect to inter-service communication are available based on where the decision for late binding is implemented, as also described in [RLZ13]:

- Integrated into the application. This requires that the application needs to support the different technologies. In order to have optimal placement, orchestration needs to learn the possible supported interconnection technologies from the application so that it can combine this information with supported technologies on the hosting platform. A drawback is that the application needs to support several technologies even though some of these can be obsolete at placement or deployment time.
- An alternative approach, also in user space, is the use of modified user libraries and system call layer so that the application itself is unaware from actual underlying communication technology used. In this case the hosting platform needs to support the different technologies. This technique does not apply for proprietary OS or user space libraries. Selection of actual underlying technology can be determined on the platform itself or by a management layer instructed by placement logic. Zone orchestration placement needs to learn the connectivity requirements of an application along with the execution capabilities in order to provide optimal placement and connectivity.
- Below system calls layer and above transport layer. From orchestration point of view, this is similar to the above case. This option has the benefit that only the host OS needs to be modified.
- Below IP layer.

Implementation in different layers may bring different impacts on programming transparency, kernel and hypervisor transparency, seamless agility and performance overhead. Additional complexity is introduced in case of migration. Communication protocol state needs to be synchronised over different technologies. Regarding security, each communication technology differs in the security levels it offers. For example, shared memory communication requires that the collaborating services have a trust relationship. This topic impacts orchestration, placement and deployment.

Some common features that should be taken into account for FUSION when discussing inter VM communication optimisation, are the following:

- Speed: faster than TCP/IP on regular communication channel (Ethernet or loopback)
- Auto co-residency detection (e.g., in the guest or at the host level)
- VMM extension avoidance (general applicability)
- Kernel patching avoidance
- POSIX interface
- Migration: switching between shared memory and TCP/IP
- Security
- Reliability
- Stability
- Performance: no matter whether the network protocol is TCP or UDP, the size of messages is extremely small or large, the arriving frequency of messages is normal or badly high, the number of co-resident VMs is large scale or not, the performance is expected to be reasonably stable and the system is supposed to operate normally.

Metrics about different techniques are discussed in [ZLR13].

Copyright © FUSION Consortium, January 2014

Req no.	Description	Level
LB-1	Support intra-node (on-board inter-VM) late binding	M
LB-2	Support intra-zone inter-node (off-board) late binding	S
LB-3	Support inter-domain late binding	C
LB-4	Support heterogeneous communication protocols and hardware	S
LB-5	Support injection of transformation components depending on communication channel	M
LB-6	Support injection of security components depending on communication channel	M

2.11 Heterogeneous execution environments

Due to the distributed nature of the FUSION architecture, execution zones will typically be very heterogeneous in nature, meaning that the physical (and logical) infrastructure of each execution zone may be fundamentally different:

- They may significantly differ in size: execution zones closer to the users may be very distributed but relatively small and perhaps specialized in nature with limited capacity; whereas the more centralized execution zones will likely be much larger and more generic in nature, containing a virtually infinite amount of capacity. The FUSION orchestrator needs to take this into account when deploying FUSION services onto these sets of execution zones with varying deployment characteristics.
- Different execution zones may be deployed on different types of data centres, grids or cloud environments, each having their own management interfaces. Some may be fully virtualized and automated environments; others may be more dedicated physical environments. The zone manager needs to be able to deal with this heterogeneity in infrastructure management systems.
- Due to the demanding nature of the types of services we intend to deploy on FUSION, we envision also the adoption of physical infrastructures with specialized hardware accelerators like GPUs, FPGAs and others, which FUSION services should be able to leverage for more efficient processing. A number of key challenges will be on the one hand towards the services, which need to specify somehow what functional and hardware capabilities they expect from an execution zone, and on the other hand, how an execution zone could safely and reliably expose these accelerators to the appropriate services.

Req no.	Description	Level
HE-1	Support heterogeneous data centres	M
HE-2	Support hardware acceleration (GPU, etc.)	M
HE-3	Deployment and placement taking into account heterogeneity and accelerators	M

Copyright © FUSION Consortium, January 2014

2.12 Light-weight virtualization and deployment

In the FUSION architecture, new service instances may have to be deployed relatively fast on a new execution zone close to the end user. To minimize deployment time, light-weight deployment and virtualization mechanisms may be required. Secondly, the inherent real-time nature of typical FUSION services can demand fine-grained resource isolation mechanisms. For example, light-weight virtualization techniques allow deploying new containers within seconds, whereas for full-blown virtualization, this can often be one to two orders of magnitude longer.

Some additional requirements that may be relevant for making the decision to opt for a particular virtualization technique include the footprint, quota, scheduling and latency requirements, security constraints, billing purposes, scaling efficiency, migration support, package management and overall management tools, dependencies with respect to the host OS and libraries, etc. Other considerations include real-timeness, resource requirements and efficient communication mechanisms in between isolated services. Three example light-weight virtualization frameworks are OpenVZ [KKVZ06], LXC [LXC13], and Docker [DOCK13].

In this project, we will evaluate the feasibility and effectiveness of light-weight virtualization and deployment mechanisms for deploying and managing particular types of services. These requirements can be divided into a number of categories:

- Isolation requirements

  The virtualization software should be able to segregate the basic resources and the network stack of the systems running under its supervision. Optionally, this can be extended towards heterogeneous hardware resources as well.

- Efficiency requirements

  The tool should be able to cause the lowest possible overhead in order to keep the maximum of the host's resources available to the VMs.

- Scalability requirements

  The relationship between the consumption of system resources and the number of running VMs should be approximately linear, giving enough elasticity for the number of nodes that can be created in the virtual plane.

- Flexibility requirements

  The virtualization tool must support the attachment of multiple virtual network adapters for each VM.

- Monitoring requirements

  The tool should allow for monitoring of its resources and health in a configurable manner.

Req no.	Description	Level
LV-1	Support fast light-weight virtualization environment	S
LV-2	Support fast light-weight deployment environment	M

2.13 Security and integrity

The security aspects and requirements for FUSION have been discussed already in detail in Deliverable D2.1. Regarding orchestration and execution management, FUSION shares a large

Copyright © FUSION Consortium, January 2014

amount of common vulnerabilities and risks as classical cloud orchestration systems. As we will build FUSION on top of existing cloud management platforms, we can easily leverage the existing security capabilities provided by the underlying platform. Obviously, at the FUSION orchestration level, additional security functions need to be provided and implemented to guarantee security and integrity of:

- FUSION services
- FUSION service instances
- Service routing towards specific services (see also Deliverable D4.1)
- Overall resource management and isolation

In this project, we will not focus on these security topics in detail, but rather on the technical challenges related to the FUSION architecture. We will however validate whether a particular design and implementation is compliant with all relevant security requirements we described in Deliverable D2.1.

2.14 Evolvability and backwards compatibility

Evolvability means that the FUSION architecture can be gradually deployed over today's Internet, rather than building on a clean-slate design. To improve the likelihood of adoption, it should be possible to easily deploy legacy services onto the FUSION architecture on the one hand, and to be able to easily deploy the FUSION architecture on existing network and compute infrastructures, data centres, without forcing early adopters to have to radically change their entire infrastructure and allocate all available resources to FUSION.

Consequently, both for the domain orchestration and execution zone management, we opt for an overlay approach. The FUSION domain orchestration functions can be deployed anywhere, either by encapsulating them as classic data centre services, or by deploying them inside FUSION execution zones themselves. The execution zones and corresponding zone managers will be deployed on existing data centre infrastructures by adding a data centre abstraction layer, which is an agent that mediates between the FUSION zone manager and the existing data centre management middleware. FUSION services will be automatically wrapped into whatever enclosing environment the data centre management layer supports. Legacy services can be easily transformed into FUSION services by means of creating appropriate wrappers for handling state management and creating FUSION service manifests for handling configurability and automating deployment.

In a later stage, we can investigate how to extend the OpenStack APIs for supporting the key FUSION orchestration and management functions. This would enable any data centre management platform supporting the OpenStack APIs to also support the FUSION orchestration and management capabilities, avoiding the need for an extra layer of abstraction for such environments.

Req no.	Description	Level
EBC-1	Deploy execution zones on existing data centre infrastructures	M
EBC-2	Wrap and deploy legacy services as FUSION services	M
EBC-3	Integrate FUSION APIs into existing data centre management platforms	C

3. RELATED WORK ON DISTRIBUTED SERVICE MANAGEMENT

IN this section, we will give an overview of some of the related work regarding distributed service management. We will focus on orchestration aspects, execution aspects, service description/manifest aspects and service composition/distribution aspects.

3.1 IRMOS

3.1.1 IRMOS architecture and orchestration

The general goal of IRMOS (Interactive Realtime Multimedia Applications on Service Oriented Infrastructures, see [IRMO11a] and related reports of the IRMOS consortium) is to enhance SLAs in a grid/cloud computing platform with strict quality guarantees in the transport network. To this end, all computational, storage and network elements of IRMOS platform provide guarantees to individual activities while the physical resources are shared across multiple services. IRMOS provides means for automatic deployment of services on best fitting resources distributed in a network. Within the IRMOS platform, it is the ISONI (Intelligent Service Oriented Network Infrastructure) component that is responsible for implementing the overall architecture including resource control plane, path manager, and execution environment. A schematic representation of the IRMOS architecture with an emphasis on the orchestration/control plane is depicted in Figure 7. Major elements of the platform are explained in the remainder of this section.

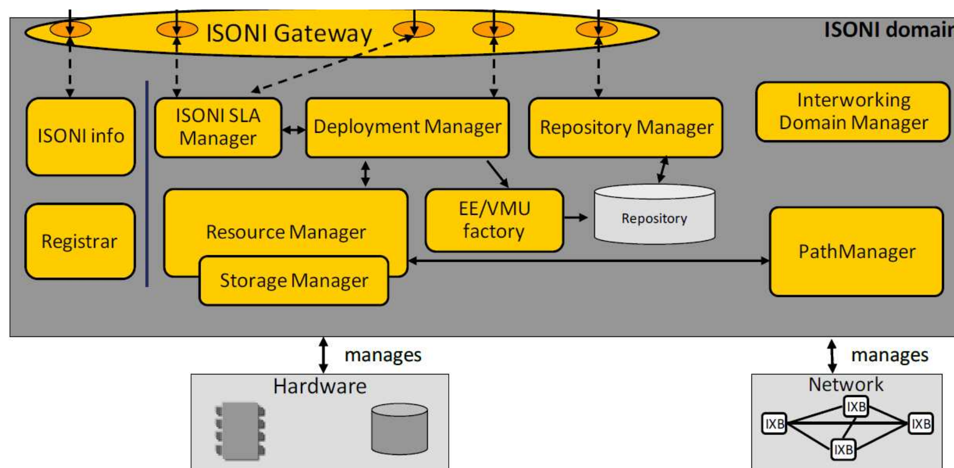

Figure 7: overall IRMOS architecture.

The overall coordination of service deployment process in ISONI is accomplished by the Deployment Manager which (in a functionally centralised manner) matches computing and network resources required by deployed services by negotiating needed resources with Resource Manager and Path Manager, respectively.

The deployment and instantiation of a developers' service is based on an abstract description of all the execution environment requirements of the service (in the form of Virtual Service Network, VSN), including the description of the connectivity interconnections between service components and their individual QoS demands. ISONI thus orchestrates service execution based on VSN specification while VSN description serves as a template used to instantiate "actual" VSN being an instance of the service. VSN, when instantiated, takes form of a virtual network with a private address namespace for its service components (SCs). Connections exist between service components (SC, implemented by corresponding virtual machines) of a given instance of VSN to exchange application data between

Copyright © FUSION Consortium, January 2014

them. The connectivity architecture of IRMOS/ISONI with virtual service abstractions is shown in Figure 8.

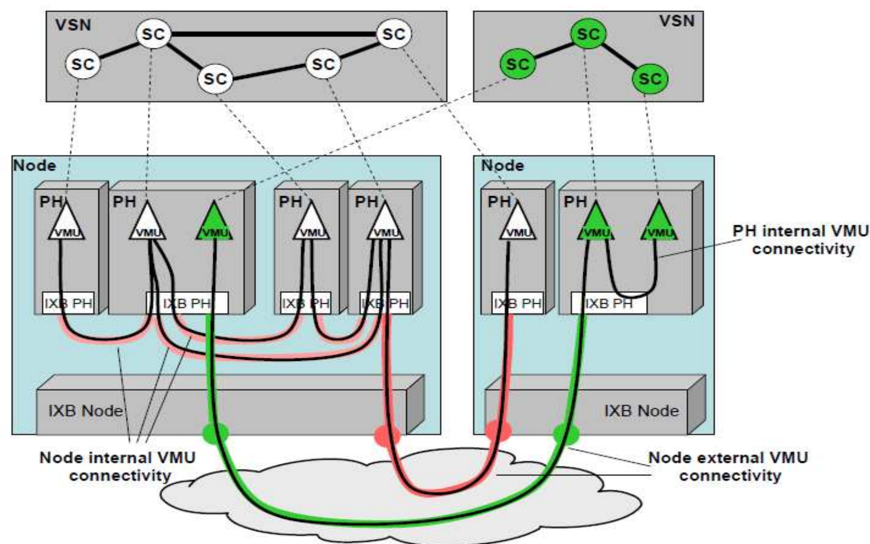

Figure 8: IRMOS/ISONI infrastructure and connectivity abstractions.

In the above setting, Resource Manager is responsible for the selection of computational and storage resources. In particular, the timing requirements of services are taken into account through advanced resource reservation mechanisms. Complementary to this, Path Manager is the key functional block that implements control functions related to providing connectivity QoS guarantees. In particular, Path Manager is responsible for resource discovery, selection and necessary configurations and supervision of all network allocations required during the VSN life-cycle. So, Path Manager can be thought of as a bandwidth broker whose operation could comply with the NaaS paradigm. It adopts the network resource model that is derived from the ITU-T G.805 series models. A 2-level hierarchical architecture of Path Manager in IRMOS strictly corresponds to the hierarchical organisation of IRMOS platform into IRMOS “nodes” that contain (gather sets of) IRMOS “physical hosts” and IRMOS “domains” that are composed of physically interconnected IRMOS nodes. This structure allows Path Manager to have full control over basic topological aspects of connectivity services.

Actually, IRMOS/ISONI assumes that Resource and Path management are split into domain and node level (node corresponding to a data centre, and domain to a set of nodes). The actual flow control of activities when instantiating a VNS takes the form of a two-step process as depicted in Figure 9 [IRMO11b]. In the first step, Deployment Manager checks for a set of valid nodes through consulting with domain Resource/Path managers, and then in the second step resource reservation takes place through node Resource/Path managers. After successful reservation, node Resource/Path managers autonomously deploy service components and establish connectivity using reserved resources.

Copyright © FUSION Consortium, January 2014

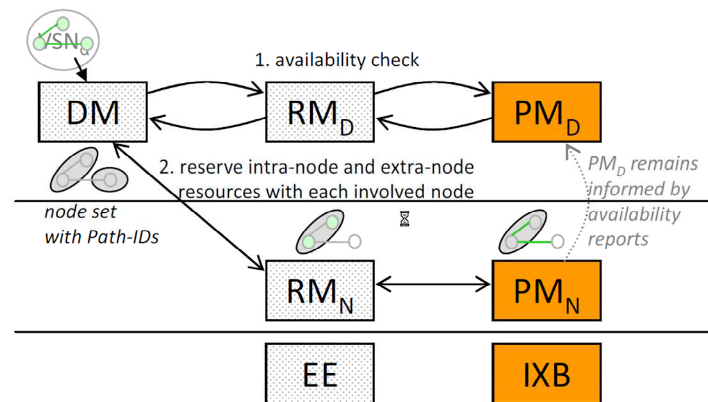

Figure 9: Architecture of resource and path management blocks.

The separation of traffic of different VSNs is achieved by tunnels created and managed by ISONI using the IXB (ISONI eXchange Box) entities (IXB Phys. Hosts and IXB Nodes). IXBs are responsible for routing, encapsulation, decapsulation and virtual-physical address mapping, and also implement packet handling mechanisms to enable QoS enforcement. Each VSN creates a unique namespace and IXBs always forward packets internally based on a combination of namespace and virtual address. It is assumed that once a VSN has been created within the platform, it will not be possible to change its structure at run-time. More specifically, a decision about where and when the application service components are to be executed is made at reservation time; after that no changes are made concerning the component during run time such as migrating to another machine in case of breaking the SLA or finding a better resource for execution.

3.1.2 ISONI infrastructure

The unique feature of ISONI consists in integrating into one platform the network resource management and allocation functions and cloud services under the overall orchestration provided by Deployment Manager. In this setting, Resource Manager is responsible for the selection of computational and storage resources, and service encapsulation is achieved through VM. This is the responsibility of the execution environment that provides global resource management, allocation policy, and run-time SLA control. The execution environment allows to meet the requirements negotiated in the form of SLA including such detailed specifications as VSN-level resource requirements of VMs and OS environment (CPU, memory, network interfaces, dedicated devices, etc), and service component-level requirements as to library dependencies and software modules needed. Actual implementation (execution) of computational requirements such as isolation, timing, persistence (save/restore resources) is realised through sophisticated mechanisms like VM CPU scheduling, reservation etc. in ISONI physical nodes.

The execution of connectivity QoS requirements takes place through sophisticated packet queuing and filtering disciplines applied to VM traffic in IRMOS IXB routers (see Figure 8).

Overall, the execution capabilities of IRMOS/ISONI component correspond to those in advanced virtualised infrastructures with enhancements to NaaS-like capabilities. From the FUSION perspective they can be positioned as proprietary data centre mechanisms accessed by a FUSION zone manager. It must be stressed however that according to the current decision of the FUSION consortium, a light weight approach to connectivity layer QoS is pursued in FUSION, and potential inclusion of NaaS capabilities can be considered in a later phase of the project.

3.1.3 Virtual Service Network: service description

A central role in resource management in IRMOS/ISONI is played by Virtual Service Network (VSN) description being a formal specification of the requested resources needed by the application (see

Figure 8). VSN is used to select and schedule resources at the ISONI domain and node level and to derive the admission policy for deployment of virtual machines and network links. This description has to be delivered by the service developer who does not need special knowledge about the network infrastructure.

It takes the form of a graph in which nodes correspond to service components (realised by VM), and links represent network connectivity between particular components. In principle, any parameters that are desired by the developer and are supported by the platform can be specified to describe both service components and connectivity links. For example, a link between components can specify whether uni- or bidirectional communication takes place, bandwidth in each direction, delay, jitter, etc. Similarly, service component description could comply with, but is not restricted to, specification in the form of JSDL (Job Submission Description Language); in fact, IRMOS seems to have not proposed any standard notation in this respect, although it introduced network resource ontology for the network resource model [IRMOS11b].

3.2 NGSON

Next Generation Service Oriented Network (NGSON) [NGSO11] identifies several individual architectural components and functionalities that are relevant for FUSION. As of today, only the functional architecture of NGSON has been standardized, but no interface specifications are available. The overall architecture of NGSON is depicted in Figure 10 [LEKA12]. In the figure, an exemplary flow for the signalling layer is presented (black dotted curve) in order to show how components of NGSON may cooperate to route client requests among multiple services under the orchestration of a Service Composition functional element. In particular, it is shown that Service Composition element controls “service routing” towards elementary services; the latter can be based on a per-service specification given in the form of a manifest (note that some additional information flows that can occur in such a case – e.g., between Service Composition and Service Discovery&Negotiation elements – are not shown for compactness reasons). In [LEKA12] BPEL (Business Process Execution Language) notation was adopted to specify such service manifests.

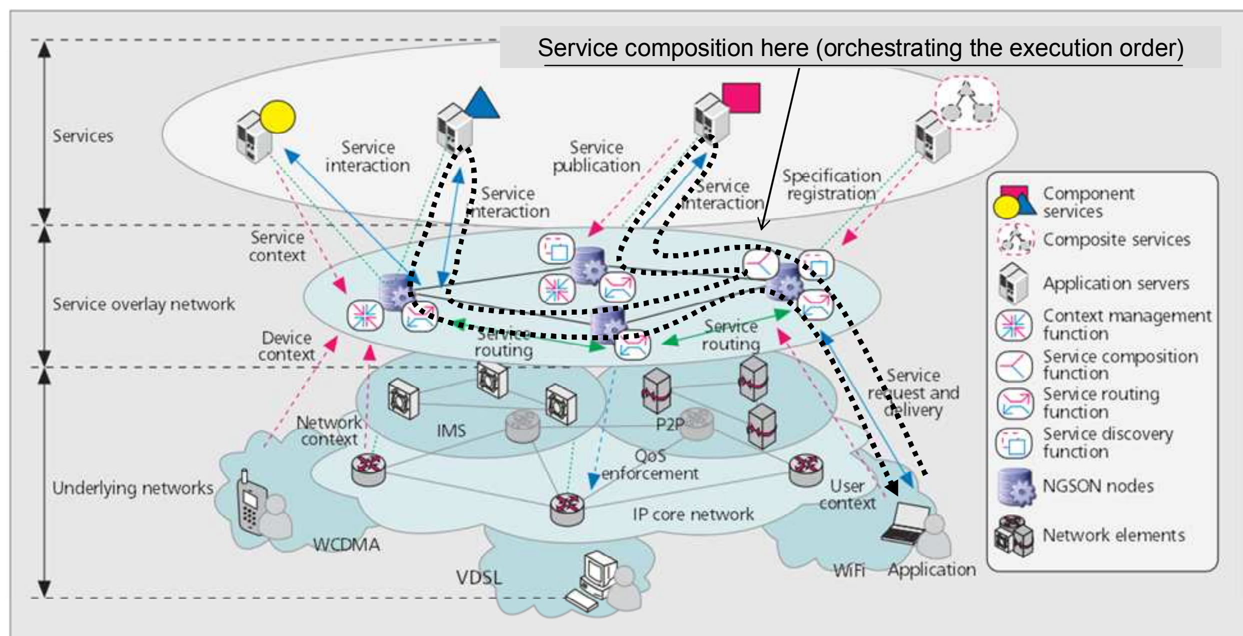

Figure 10: Overall architecture of NGSON [LEKA12].

NGSON does not cover resource management in data centres. In principle, NGSOM can only map service requests to the best running service instance (through Service Routing function that uses Service Discovery&Negotiation function and Service Policy Decision function). Accordingly, functions like service atom placement are out of scope of NGSON. Thus, NGSON best corresponds to the service routing plane of FUSION architecture where service instances can be run in distributed data centres.

Regarding network resources, NGSON adopts a classical bandwidth-broker approach to network resource and QoS control (through Service Policy Decision fed by the service routing – see QoS enforcement arrow from the Service Routing node towards the IP router). Obviously, this approach can in theory be adopted by FUSION, but its tight integration with the management of cloud-based resources (computation, storage) may rise strong scalability concerns.

NGSON provides certain capabilities for service composition/orchestration which take the form of coordinating the invocation of several basic services (service atoms) in response to a single signalling message received from the requester. In abstract terms, the latter resembles in essence the operation of Initial Filter Criteria known from IMS (however, NGSON does not impose any particular notation of service manifest for such purposes, and one example is BPEL which has been adopted in a demo implementation of NGSON as reported in [LEKA12]).

3.3 NfV

A reference architecture of Network Function Virtualization (NFV) has been discussed in ETSI and at the time of writing this report the architecture takes a form whose simplified version is depicted in Figure 11. For sake of comparison, the figure also proposes components that correspond to FUSION architecture and which are missing from original NFV architecture. The main NFV components and the main interfaces between them (solid lines for the latter) are represented together with some exemplary functions supported by these components; NFV elements of lower relevance to this report have been omitted. Dashed lines in the figure represent connectivity services between functions (services). FUSION components that are missing from the original NFV architecture are in particular service routing/forwarding functions and zone manager (blue blocks in color print) and related interfaces (dotted lines).

Copyright © FUSION Consortium, January 2014

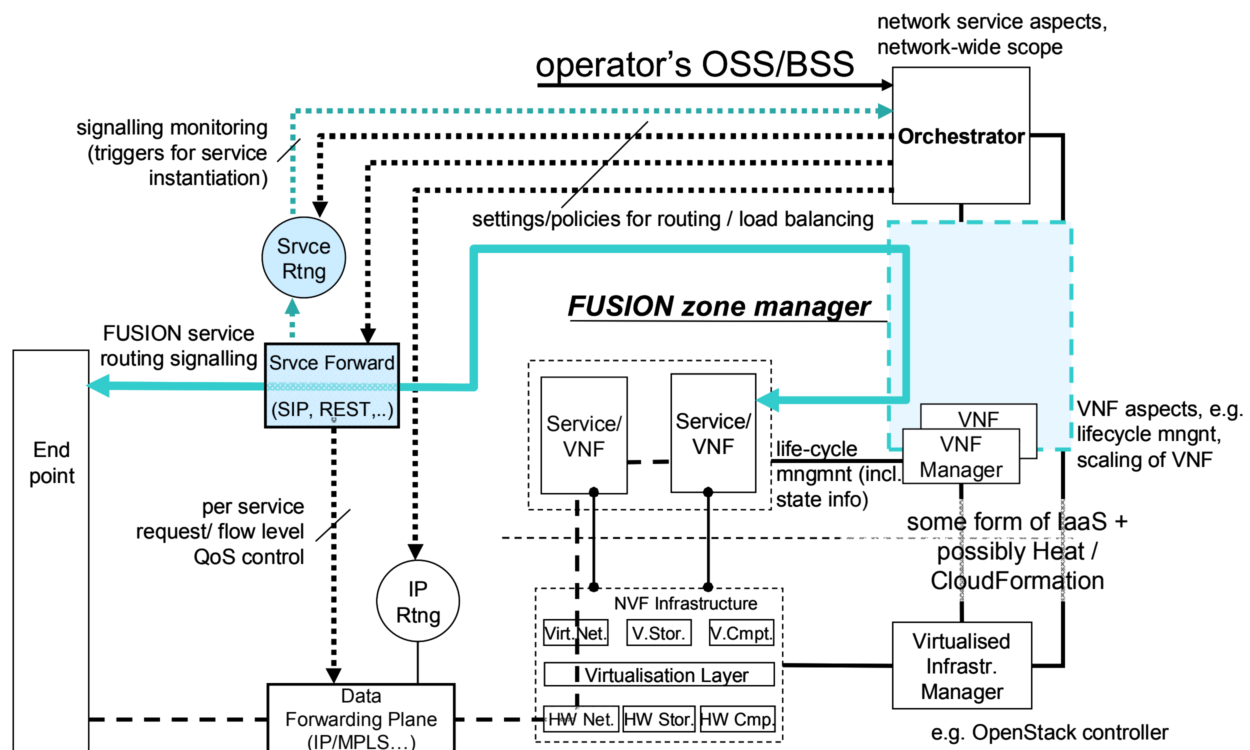

Figure 11: NVF architecture and mapping of FUSION components.

Roughly, NFV orchestration can be mapped onto a FUSION orchestrator provided that the former is enhanced with additional capabilities to be able to cooperate with the service routing plane and possibly with the data plane (e.g., IP) routing. The zone manager function defined in FUSION features several capabilities not present in original NFV; recall that in FUSION there is a need to route service requests towards service instances – a function that has not been considered in NFV. Thus, the way end users connect to services is probably one of the most important differences between both architectures. On the other hand, zone manager could potentially inherit at least some of the functions pertaining to VFM (Virtual Network Function Manager). On the other hand, both NFV and FUSION can be positioned similarly with respect to the infrastructure with possible use of virtualization and the access to virtual resources mediated through interfaces such as IaaS/OCCI or orchestrator APIs like those available from OpenStack and Amazon AWS.

From FUSION perspective it may be important that NFV framework allows that the services (Service / VNF – Virtual Network Function in the figure) are self-aware of their needs to scale up or down with changing load or lack of resources and they may trigger corresponding actions themselves. Such actions may require specific cooperation patterns between the service (VNF), VNFM and the orchestrator. In FUSION, this aspect of functional responsibilities between of components still needs more detailed considerations.

Last but not least, we note that with FUSION we are not (just) targeting network services (and corresponding VNFs) managed and deployed by the operator within its network, but we also envision external parties to deploy and manage QoS-sensitive services within the carrier-grade network, exploiting the distribution and QoS aspects that are already there. Hence the specific registration functions, etc. need to be exposed rather that remain hidden in operator's OSS/BSS. Similarly, external parties could provide their own execution zones as computation resources, but still rely on the operator's network. To this end, additional functions may be applicable in FUSION, for example the possibility to control data plane by service routing/forwarding elements (see respective interfaces to data plane elements in Figure 11).

Copyright © FUSION Consortium, January 2014

3.4 Amazon AWS

3.4.1 Amazon EC2

Amazon EC2 is an IaaS solution with virtual servers placed on Amazon's infrastructure in user-defined places according to user-specified requirements with respect to all typical parameters such as CPU/memory allocation, image (predefined or user-provided), storage access (local on the VM or virtual EBS), etc.

Apart from that, EC2 provides many specialised types of virtual server instances, for example high storage instances, high I/O instances, cluster instances (high compute and high networking), GPU instances, and many more.

Amazon EC2 features a long list of capabilities such as e.g. multiple locations, CloudWatch for instance monitoring services, Auto Scaling (enabled by CloudWatch), Elastic Load Balancing, Virtual Private Cloud, to mention a few. Their short description follows.

CloudWatch is a monitoring service to gather measurements for particular server instances. Actually, its functionality has been followed quite closely by OpenStack's Ceilometer so at a high level both products are equivalent. The measurements cover many important performance parameters related to running instances and thus can be used across OpenStack to serve multiple purposes that range from orchestration including auto scaling to billing.

Auto Scaling enables scaling the service by adding or removing server instances in predetermined increments based on specified conditions (e.g., CPU load). Included is the ability to scale the service smoothly, that is to add/remove server instances after a configurable cool-down period, which makes Auto Scaling wait for a predefined time after a scaling action before it evaluates scaling conditions again.

Elastic Load Balancing (ELB) allows to distribute application traffic among multiple server instances. The set of instances can be fixed or, in case of using Auto Scaling capability, it can be variable depending on predefined conditions. Load balancers can be used to face Internet or inside Virtual Private Clouds; they can load balance traffic within Amazon's Availability Zone or among multiple zones. Load balancing can be done on TCP, SSL, HTTP and HTTPS levels. Recently, the Proxy Protocol option (http://haproxy.1wt.eu/download/1.5/doc/proxy-protocol.txt) has been provided to enable load balancing of any INET protocol while preserving session between a client and the server (so called sticky sessions).

3.4.2 Amazon CloudFormation

Amazon CloudFormation is an orchestration tool that allows to deploy and modify associated collection of resources (called a stack). It can be done using AWS Management Console, CloudFormation command line tools or APIs.

CloudFormation works with so called templates, which are JSON documents that describe infrastructure requirements for a stack. A template contains a pattern that specifies how stack elements have to be orchestrated throughout a stack lifecycle. A template can contain several sections, some of which are listed below.

- Parameters serve to pass values to the template at run time and can be referenced from other sections.
- Mappings contain conditional values. For example, it is possible to specify a list of server image identifiers and assign particular image identifiers to regions where the elements of a stack can be instantiated based on those images.
- Conditions control the creation of resources or the assignment of values to resource properties during stack creation or update.

Copyright © FUSION Consortium, January 2014

- Resources have types and properties, and are elements of a stack that are to be managed by CloudFormation. Properties can take values in various forms, e.g., as literals, parameter references and even intrinsic functions to pass values that are not known until run time.
- Outputs define information that should be sent back to the user of the template. Literal values and intrinsic functions can be used for passing information.

It can be seen that templates provide a notation that allows to define quite complex service orchestration patterns. Noticeably, OpenStack's Heat is another example of a similar tool compatible with CloudFormation. Concluding, we suggest that one of the options for FUSION could be to use an interface to cloud infrastructure that supports CloudFormation orchestration capabilities. This could relieve FUSION orchestrator from lower-level orchestration tasks while still being able to apply its own orchestration policies in a distributed heterogeneous inter-cloud environment.

3.4.3 Amazon AppStream

Amazon AppStream [APP13a] [APP13b] provides a flexible and low-latency service that offers resource intensive streaming applications and games from the cloud thereby enabling use cases in these areas that would not be possible running natively on mass-market devices.

AppStream includes an SDK supporting streaming apps from Microsoft Windows Server 2008 R2 to devices running FireOS, Android, iOS, and Microsoft Windows. From the above, it is clear that AppStream is currently limited to:

- Specific platforms on server and client side;
- Although gaming is specified (where latency and jitter is of prime importance and these are closely related to the position of the server in the network), it is unclear what kind of gaming is intended;
- The streaming between client and server is via a proprietary STX protocol (which is an Amazon proprietary alternative to the open standard SPICE protocol as described by RedHat).

FUSION on the other hand targets heterogeneous cloud and support for different cloud orchestration layers with focus on orchestration and placement for distributed cloud computing whereby the execution zones are located deeper into the network in order account for latency/jitter in an optimal way. Additionally, FUSION intends to support composite services from an orchestration point of view. In this respect AppStream only relies on other AWS functions as AutoScaling and Elastic Load Balancer. The AppStream entitlement service offers control on the client to application connectivity grants (identity management). In FUSION, this is left to the application developer.

3.5 OpenStack

OpenStack is a cloud management system that defines a set of interfaces that provide a full range of infrastructure (IaaS) services. A simplified architectural view of OpenStack is provided in Figure 12. In the figure, the functional blocks are shown, omitting their detailed internal structure and only exemplary interfaces are provided for sake of clarity of the drawing. A full description of the architecture can be found e.g. in [Open13].

Copyright © FUSION Consortium, January 2014

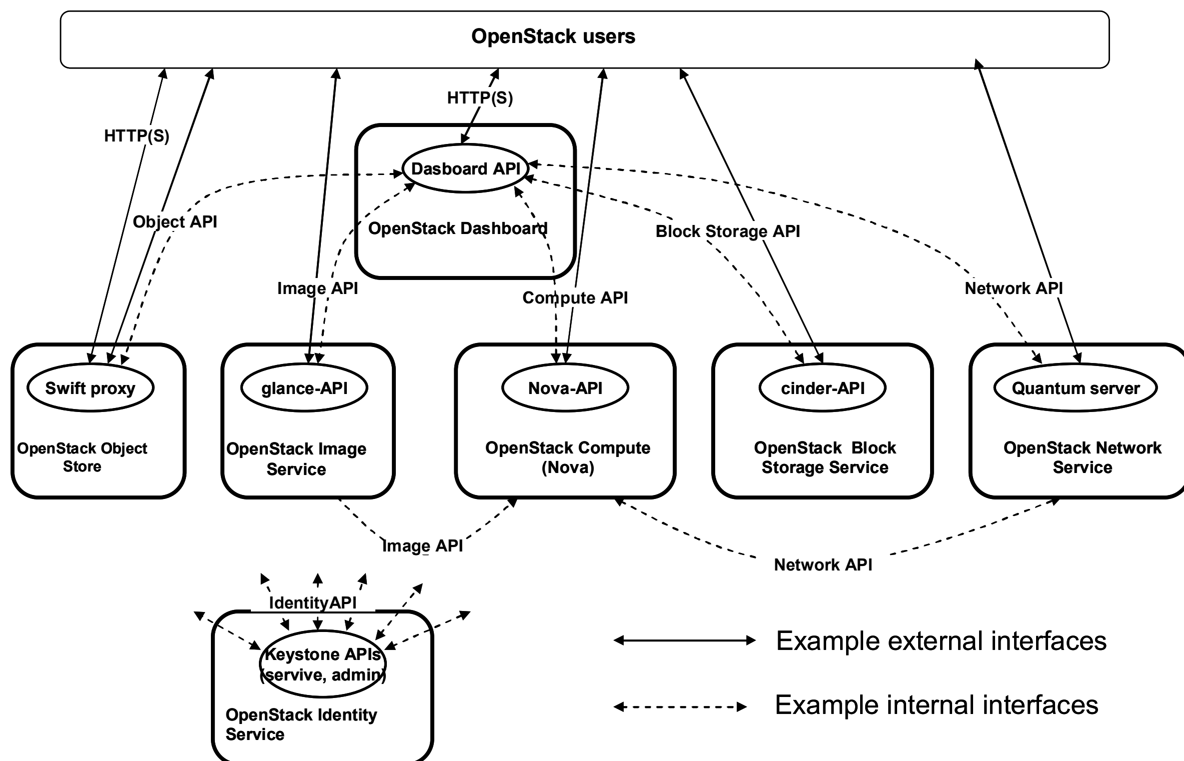

Figure 12 OpenStack architecture.

OpenStack compute a.k.a. Nova, is a Python-based software used to orchestrate cloud and manage virtual machines and networks. Nova allows us to create and manage virtual servers using machine images. To this end OpenStack supports many popular hypervisors, e.g. KVM, QEMU, Xen, VMware. For example, nova-scheduler service is responsible for determining how to dispatch compute and volume requests among physical resources. In doing that it uses a filter scheduler to filter and weight tasks to optimise decisions on where a new instance should be created. Filters are selected by the OpenStack admin based on a set of standard classes; custom algorithms can also be implemented by administrators. Host weights are dynamically calculated according to predefined rules based on both task parameters as well as detailed data about the host itself. The message for FUSION is that the functionality of Nova should be in principle hidden in the execution zone behind the zone manager.

OpenStack Object Storage a.k.a. Swift is roughly similar to Amazon S3. Swift allows to store objects in a massively scalable infrastructure with built-in redundancy and fail-over. It can be used to store static data (like images and videos), make back-ups, archive data, and so on. Swift will write copies of data to multiple redundant servers that are logically grouped into Zones. Zones are isolated from each other to safeguard from failures. One can configure Swift and decide the number of Zones and replicas there need to be in the system. Object is the basic storage entity in Swift. An object can be anything like a document, audio, or video data. A container, which is similar to buckets in S3, allows to organize objects by grouping them. Swift simply provides API endpoints to store and manipulate objects. One cannot use Swift as a file system and objects are not accessible via any file sharing protocols.

OpenStack Image Service a.k.a. Glance, is responsible for storage, discovery, and retrieval of virtual machine images. Glance can be configured to store VM images in Object Storage, Amazon S3, or a simple file-system. Glance-registry and Glance-api are the two important components of Image Service. Glance-registry stores and retrieves metadata about images. Nova interacts with Glance using Glance-api for querying and retrieving actual VM images.

Moreover, Open VSwitch defines the OpenStack Network API, which is intended to provide "network connectivity as a service" between devices managed by OpenStack compute service. The service is

Copyright © FUSION Consortium, January 2014

based on the notion of virtual networks that effectively are virtual L2 broadcast domains. On a high abstraction level, such virtual domains could correspond to ISONI VSNs, but as the focus of IRMOS is more on services, the detailed models of its VSNs differ significantly from those of OpenStack.

In OpenStack, a related network-level construct is Open VSwitch [http://openvswitch.org/] that is a NFV-like virtual switch that supports a multitude of useful features, e.g., Standard 802.1Q VLAN model with trunking, A subset of 802.1ag CCM link monitoring, STP (IEEE 802.1D-1998), fine-grained QoS control, OpenFlow protocol support, to mention a few.

The potential role of OpenStack in the context of FUSION seems to be similar to that of Amazon services. As has already been noticed, both platforms share a lot in common, especially when it comes to orchestration capabilities (AWS CloudFormation and OS Heat). The key fact is that from the FUSION viewpoint, OpenStack is responsible for tasks internal to FUSION execution zone so the most probable position of OpenStack (and also Amazon AWS) in FUSION architecture is the data centre hidden behind the zone manager function. In particular, FUSION could use a set of OpenStack APIs for orchestration thus being isolated from the internal details of the data centre (the latter being comparable to the FUSION execution zone).

3.6 Service description and orchestration

3.6.1 TOSCA

The OASIS consortium very recently released the a first version of a new standard on Topology and Orchestration Specification for Cloud Applications (TOSCA) [TOSCA13]. The main goal of this new standard is to be able to describe the topology of a cloud application along with their dependent environments, services and artefacts inside a single service template, allowing to deploy and manage these services across multiple cloud infrastructures. This enables better portability and management of cloud applications across heterogeneous cloud infrastructures, which is very relevant for FUSION due to its inherently heterogeneous and distributed nature of FUSION execution zones.

The TOSCA specification defines a metamodel for defining IT services. It formalizes many of the interactions between a service developer delivering a service and the entity that is operating and deploying the service. This metamodel describes both the structure of the service in a TOSCA topology template as well as how to manage that service throughput its lifecycle, something that is described in a TOSCA plan. The main components for describing a service in TOSCA are depicted in Figure 13.

Copyright © FUSION Consortium, January 2014

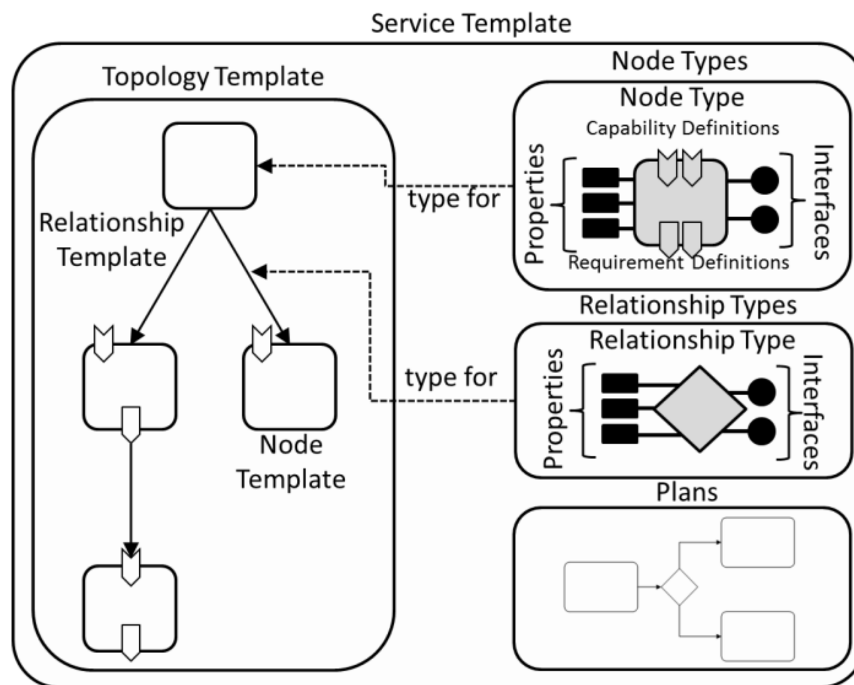

Figure 13: Structural elements of a TOSCA service template and their relations.

A topology template captures both the node templates as well as the relationship templates between the nodes. The nodes and relationships are instantiations of particular node and relationship types, each of which define the properties as well as the operations or interfaces that are available for manipulating the component. Node and relationship types can also define a set of requirements and capabilities regarding for example dependencies on other components or regarding expected capabilities of the hosting environment.

A service template may also contain a plan describing the management aspects of service instances, with specific focus on service creation and termination. It relies on existing languages like BPEL or BPMN for describing the workflow of the service, although any process model language could be used.

A deployed service is an instance of a service template. For this, a special build plan is typically used, which provides actual values for the various properties of the defined nodes and relationships. This may include concrete IP addresses for the actual service instance.

A TOSCA service template may contain references to two kinds of artefacts, namely implementation artefacts as well as deployment artefacts, which contains all necessary content (scripts, application binaries, images, etc.) for executing or deploying the service instance in a TOSCA-aware environment.

A TOSCA service template may also contain non-functional behavior or QoS via TOSCA policy templates and policy types, each of which may be associated with particular node templates. Example policy settings include monitoring behavior, payment conditions, scalability, etc.

All information, namely the service templates, types, plans as well as the artifacts, can be combined in an archive format called CSAR (Cloud Service ARchive), which is a self-describing archive containing all definitions and models for describing the service in TOSCA.

Copyright © FUSION Consortium, January 2014

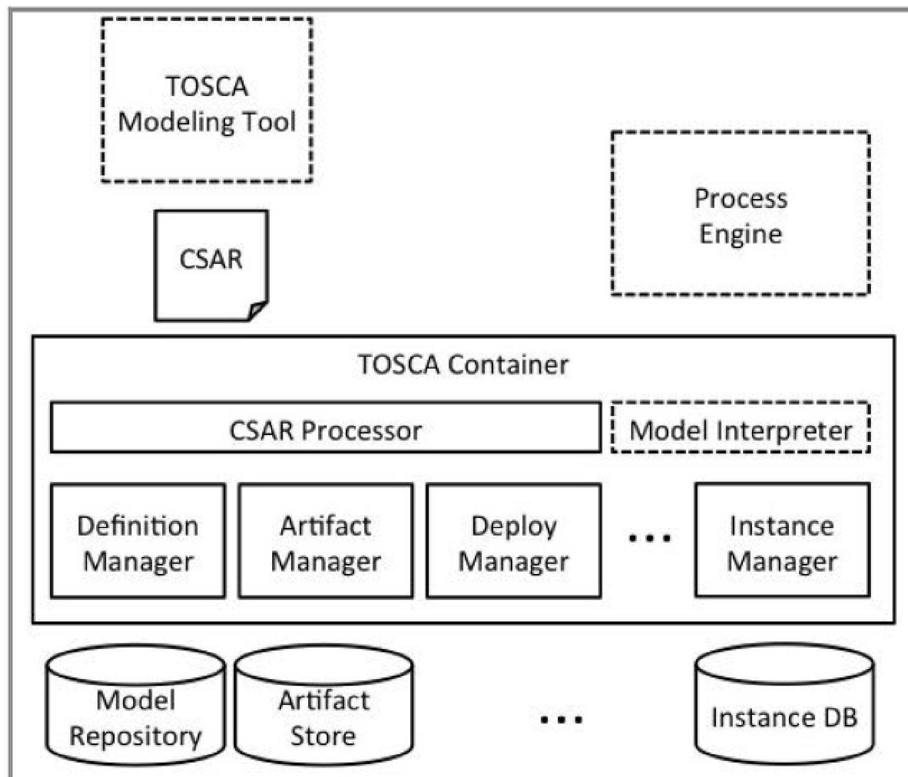

Figure 14: Sample architecture of a TOSCA environment.

Figure 14 depicts a sample architecture for a TOSCA-enabled environment. Obviously, there needs to be a CSAR processor that is capable of interpreting the CSAR archive and its containing content. A model interpreter is required in case particular process models are used for modelling service behaviour (e.g., via BPEL). Other key functions include an artefact manager and store for managing all artefacts, a deployment manager for creating a new instance based on a service template as well as an instance manager for service state management of all active TOSCA service instances.

In FUSION, we will evaluate TOSCA in more detail and analyze its applicability for describing FUSION services. One of the potential limitations we observe at this point is that TOSCA is mainly targeting static service graphs that fully describe a service graph in a rather preconfigured static way.

3.6.2 Cloudify

Cloudify is designed to bring any application to any cloud enabling enterprises, ISVs, and managed service providers alike to quickly benefit from the cloud automation and elasticity. It provides externally managed orchestration for an application's deployment and runtime. By treating infrastructure as code, it allows to describe deployment and post–deployment steps for any application through an external blueprint (also called a recipe).

The steps that can be distinguished in the Cloudify approach are:

1) Upload recipe

   Install an application with a single shell or REST command (no code changes)

2) Create VMs

   Provision compute resources needed on demand, on the different cloud technologies using a "Cloud Driver".

3) Install agent

   Auto-install its agents (Cloudify processing components) on each VM to process the recipe.

Copyright © FUSION Consortium, January 2014

4) Process recipe

The above installed agents process the recipe and install any application tiers, which is tightly integrated with OpsCode's Chef.

5) Install application services

Deploy the application services within the cloud infrastructure.

6) Monitoring

Run cloud monitoring capabilities so that application services can be monitored for availability and performance (custom metrics can be defined and integrated).

7) Autoscaling

To enable elastic cloud, scaling rules based on any custom metric can be defined and instruct the Cloudify management layer to scale out or scale in as needed.

3.6.3 OpenStack Heat

Heat is a service to orchestrate multiple composite cloud applications using templates, through both an OpenStack-native ReST API and a CloudFormation-compatible Query API.

Heat orchestration is currently based on Amazon web services templates, and is currently being redesigned so that other formats such as TOSCA can be mapped to it. This allows for modules that can be implemented to translate from other formats into the native DSL. This translator can be built as an add-on component outside core Heat initially. Later on, a pluggable translation layer could be built more closely into Heat and the TOSCA translation component could be refactored as a module into that pluggable layer.

Heat architecture is currently in a transitional state, which is depicted in the following architectural diagram.

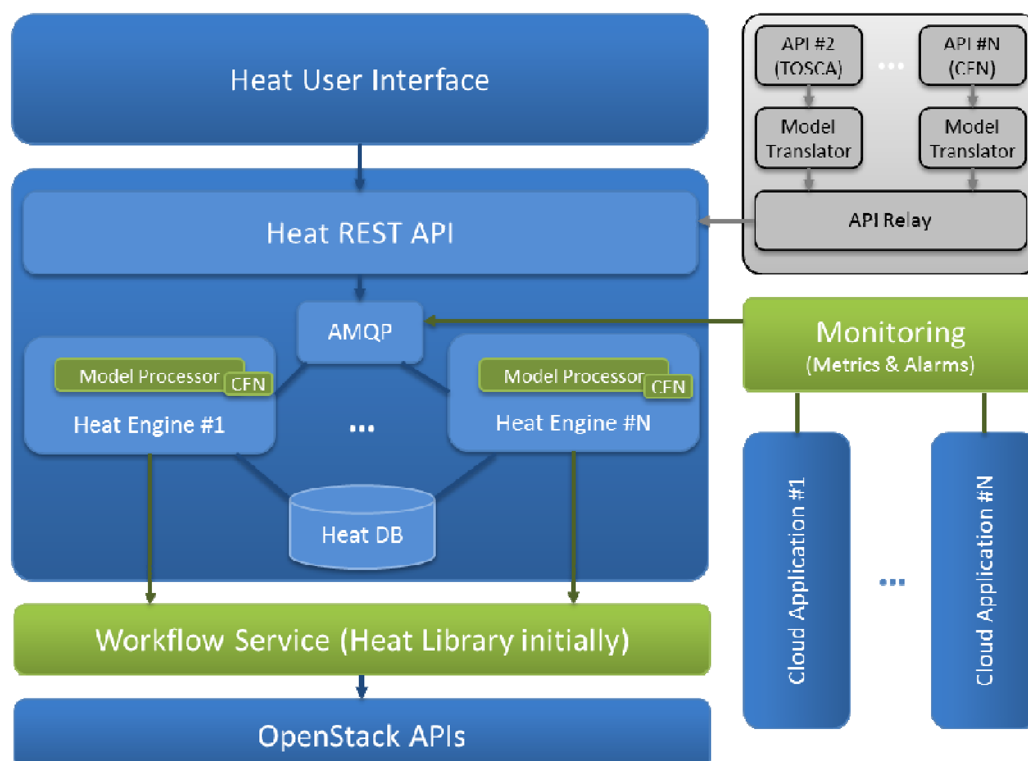

Figure 15: Architectural diagram of the HEAT-based orchestration engine

A Heat Template is passed via the REST API and goes to the queue (AMQP). One of the Heat Engines picks up the request and start processing. The Model Processor (corresponds to Parser in 1st Heat architecture diagram) reads the Heat Template's topology information, transforms this into the internal objects and derives a processing flow. Once translated into a Heat Template, the API relay will pass the Heat Template onto the native API. The entire component will be driving as an add-on component. The monitoring system posts updates to the queue (AMQP) of the Heat Engine. The update events are picked up by the right Heat Engine which then processes the update and acts accordingly.

Heat uses an Open API and a related DSL as the common expression of an orchestration. A new solution pattern will be used for compatibility with alternate template formats, allowing implementations of various emerging cloud standards. Each may implement a Model Interpreter, which will expose an appropriate service API, and decompose the given template into the common format. Once the resulting template has been generated in the open DSL format, the common API is triggered by the API Relay. This is where templates will be handled in the future, such as TOSCA, and alternate API's such as CAMP.

The Model Interpreter is responsible for parsing the DSL, and composing a deployment plan. It builds a graph of the deployment plan, and hands it off to the Heat Engine's Model Processor.

While Heat focuses on orchestration of resources, the Task System is responsible for:

- A sequence of tasks that have a start and an end.
- A persistent job/process (for example an Auto-Scale policy) that remains running until manually terminated.
- A job to run for a specified duration (such as running this automated stress test for 2 days, then exit).

Auto-scale policies may be implemented in Heat. Ceilometer provides metrics (events triggered upon evaluating sensor data) from running servers and alerts that are passed to one or more user-defined webhooks. The MAPE (monitor, analyze, plan and execution stages) will be implemented, and the "A" and "P" stages will be handled by the user-defined Complex Event Processor component.

A user-defined Complex Event Processor (CEP) can apply logic to determine what actions to take under various conditions, including triggering the Workflow Service, such as "add a node to this cluster", or orchestrations like "deploy a new cluster" or combinations of each, such as "destroy a failed cluster (workflow), and start up a new one (orchestration)". It may also send webhooks back into Ceilometer, which will be relayed to Heat, which will listen for specific scaling events in order to trigger ScaleUp and ScaleDown actions. The CEP is not hosted by Heat. It is provided by the user as an HTTP service that can accept a webhook call. *FUSION should leverage ceilometer together with a FUSION CEP to support heterogeneous HW and distributed data centres.*

Concerning OpenStack for federated datacentres, several blueprints (architecture proposals) are being discussed and proposed at the moment of writing of this document. The main issues being tackled are in the area of keystone (identity service) in order to enable single sign-on over federated datacentres. No specific architecture information about collaborating OpenStack data centres has been found until now.

3.7 Light-weight virtualization

From a system architecture point of view, lightweight virtualization, also known as OS-level virtualization, uses OS kernel features and exposes these via a user-level API. For the different OSes, different names are in use:

Copyright © FUSION Consortium, January 2014

OS	Virtualization
Linux	Cgroups, LXC, OpenVZ
FreeBSD	Jails
Solaris	Zones

In the further description, we will use container as the generic term. Containers partition the resources managed by a single operating system into isolated groups, whereas full system virtualization completely abstracts physical environments, each running their own operating system.

Containers have the advantage over VMs in that they are very lightweight and easier to manage, benefit from faster boot times making them more agile, and as terminology indicates, have an order of magnitude less overhead in size compared to VMs (VMs bring their own OS and virtualized memory space) thus significantly impacting scaling possibilities. In a real-time oriented environment where applications may have to be created on demand as envisioned by FUSION, these are key benefits.

Containers have their own isolated process space through the use of name spaces. Inside a container, application code, its libraries, its data and any package manager necessary are completely isolated from one another thereby allowing different versions of applications, libraries and packages at any given time.

Being co-located, deployed containers share underlying OS resources (e.g., page cache resources etc.) which, compared to VMs, allows for better re-use of OS resources.

Security may still be an issue with some of the current implementations of light-weight containers. Similarly to several hypervisors, some implementations of light-weight virtualization are known to have possible security issues that may allow users to take control over the host OS.

Being light-weight, the overhead of these containers are negligible, as they basically share almost all of the functionality and structures of the same host OS. The key downside of light-weight containers however is that all containers share the same OS and kernel version. In other words, it is not possible for different containers running on the same system to require different kernel versions. There is currently also no support for Windows-based containers.

Apart from the light-weight isolation and deployment characteristics, these containers also allow for fine-grained resource control of compute, memory and I/O resources.

3.7.1 LXC

The term LXC stands for **L**inu**X** **C**ontainer Tools and their main objective is to enable light-weight containers on any Linux kernel version and host distribution (but container and host architecture must match). The overall aspects are described above. Isolation is provided via Linux namespaces, that enable a customizable set of user, file system and network isolation. Resource control is provided via the Linux cgroups functionality on the level of memory, CPU, BLKIO and devices.

The corresponding LXC tools consist mainly of:

- liblxc tool to manage container itself (life-cycle management and state monitoring);
- Iproute2 package to manage the network interfaces in containers.

LXC however does not provide any orchestration, deployment, placement functionality and relies on other software to provide this functionality.

Copyright © FUSION Consortium, January 2014

3.7.2 OpenVZ

OpenVZ, previously called VServer, is the predecessor of LXC, introducing light-weight containers into a customized Linux kernel. For an introduction on the technology, see [KKVZ06]. An OpenVZ container can have root access, users, IP addresses, memory, processes, files, applications, system libraries and configuration files. Control is offered using the shell command vzctl to manage OpenVZ containers and therefore is an alternative to LXC tools. Compared to LXC, OpenVZ software consists of an optional custom Linux kernel that improves on security, stability and other features compared to the mainstream Linux kernel at the expense of reducing general applicability.

3.7.3 Docker

The company dotCloud uses containers for software delivery, and open-sourced Docker [DOCK13], the next generation of the containers engine to power a PaaS. The main intent is to commoditize management and deployment of LXC containers. It uses a Dockers-file (metadata description) to implement these goals. In Dockers, a container is a tar ball with some metadata that, through the use of “Copy on Write”, allowing for quick provisioning.

Dockers allows to create and share images. Images can be authored and can be run in any multiplicity and any server. Images are maintained in a registry where they can be committed, retrieved and searched. Registries can be public or private.

The overall concept is as follows:

1) Select an image from registry;
2) Build and run its associated Docker file;
3) Production server runs the compiled Docker file and fetches container from registry;
4) In case of network related services, load balancers are updated.

Shipper [SHIP13] provides scriptable deployment for Docker containers. Docker itself does not provide orchestration functionality. The following projects are designed to support applications in production:

- Flynn (https://flynn.io/)
- Deis (http://deis.io/)
- Coreos (http://coreos.com/)
- Maestro(https://github.com/toscanini/maestro)

From an execution layer point of view, Docker is part of OpenStack as a nova driver using a glance translator (implementing step 2 above). The Docker daemon manages the images, builds and containers and offers an API that is HTTP CLI based. At the moment of writing of this document, Dockers 0.7 has been released. Dockers 1.0 intends to handle topics such as integration with libvirt, qemu, KVM, OpenVZ.

3.8 Late binding

In this section, we provide a brief overview of existing mechanisms concerning shared memory based communication between collaborating VMs, whereby the structure as described in [RLZ13] is maintained, but focussing on the relevant sections for FUSION.

3.8.1 Between user space libraries and system call layer

- **IVC** is a user space VM aware communication library that supports shared memory based communication with a socket style interface that is derived from MVAPICH2 (MPI library over Infiniband), and a kernel driver that is called by the user space libraries to grant the receiver VM

the right to access the sharing buffer allocated by the IVC library and gets the reference handles from Xen hypervisor.

- **VMPI** for KVM only supports two types of local channels, namely one for fast MPI data transfers between co-resident VMs based on shared buffers accessible directly from guest OSes' user spaces, and one to enable direct data copies through the hypervisor. VMPI provides a virtual device that supports these two types of local channel changes in kernel and VMM.
- **Nahanni** (ivshmem): the host creates POSIX shared memory and qemu has to boot with ivshmem device information. An ivshmem_server (i.e., a shared memory server) is a user process implementing inter-VM notification. Currently, Nahanni only has local scope and not remote, and does not support migration. The shared memory on the guest is exposed in user space as a device, therefore a rewrite of the application is necessary to support this interface.

3.8.2 Below system calls layer & above transport layer

- **XWAY** has three parts, namely a switch, a protocol and a device driver. It is implemented with kernel patching and a kernel module. Conceptually, a switch layer determines co-residency and selects the lower layer communication & transport mechanism. A protocol layer does the actual data transmission using the device driver. The device driver does the effective reading/writing into shared memory.
- **XenVMC**: the guest OS hosts a non intrusive kernel module that uses a thin layer in layer 2. The solution consists of 6 parts that take care of about same functions as above (different functional split in components from architecture point of view).
- **socket-outsourcing** is realized with a socket layer guest module, a VMM extension and a user level host module. The guest achieves near native OS throughput. This option has no live migration support.

3.8.3 Below IP layer

- **XenSocket** is a one way co-resident channel based on shared memory between two VMs that offers bypass of TCP/IP in case of co-residency. Two types of memory pages are used: a descriptor page for control information storage and buffer pages for actual data transmission. This technique does not support automatic co-residence detection or switching from local to remote.
- **XenLoop** provides fast inter-VM shared memory channels for co-resident VMs based on Xen memory sharing facilities, resides at the layer of netfilter. It is offered as a set of 2 modules, namely a XenLoop guest module, implemented on top of netfilter, and a domain discovery module. This technique lacks remote support.
- **MMnet** works with fido network. In, Fido, three base facilities are offered:

  a) a shared memory mapping mechanism,

  b) an inter-VM synchronization signalling mechanism , and

  c) a connection handling mechanism.

  Fido maps the entire kernel space of the sender VM to that of the receiver VM in a read-only manner to avoid unnecessary data copies and to ensure security. It is designed for communicating between VMs that are trustable to each other, where the mapping of guest OSes' memory is acceptable.

In the scope of FUSION, orchestration and placement, the concept of co-location and the possibility of improving inter-service communication depending on platform and host/guest-OS capabilities as to improve on resource load reduction or improved service scaling is an important benefit. At the

Copyright © FUSION Consortium, January 2014

moment, capability information of services, guest-OS, host-OS, platform and topology interconnect is not being presented towards orchestration. Enabling the mediation of these capabilities towards orchestration and placement algorithms via FUSION, can leverage key benefits.

3.9 Monitoring

3.9.1 Amazon CloudWatch

Amazon CloudWatch is a part of the Amazon AWS suite and provides a framework for monitoring cloud resources and customer applications running in the cloud. It provides log data and alarms that can be used internally by other AWS and retrieved in the form of graphs or programmatically by external applications.

CloudWatch supports all AWS services, i.e., EC2, AutoScaling, ElasticCache, ELB, BlockStore, etc. For each of them, a broad range of metrics is monitored. Metrics such as CPU utilization, latency, and request counts are provided automatically for servers running in the AWS cloud. CloudWatch is also able to gather custom metrics from user applications thus extending the scope of monitored parameters and thus allowing for a customized management of cloud resources.

CloudWatch functionality is accessible via an API, command-line tools, the AWS SDK, and the AWS Management Console. Data access is available at 5 minutes and 1 minute time intervals.

Amazon CloudWatch can be used for a number of purposes like billing, accounting, capacity planning, operational management, etc. From FUSION perspective, it offers useful capabilities to FUSION orchestrator and service routing, namely a rich set of metrics for services and cloud resources, elastic system for profiling and customizing logs and alarms, and easy access to all data, while still hiding the details of the execution zone from FUSION components.

3.9.2 Ceilometer

Ceilometer's architecture is extensively described in [CEIL13]. A graphical representation of Ceilometer is depicted in Figure 16.

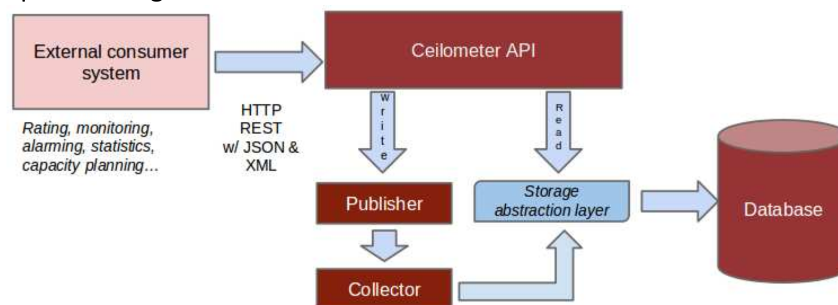

Figure 16: Ceilometer architecture

The Ceilometer component is part of OpenStack and provides a framework to collect metrics and events/alarms, store the information and offers an interface towards other projects.

It provides:

- an infrastructure to collect any information in a standardized way and irrespective of the end-goal of the metrics, and presents this information for any other OpenStack components (e.g. Heat). This topic is referred to as "metering" in the OpenStack community.
- also functionality that checks for key value variations in order to trigger various events and related reactions. This topic is referred to as "alarming" in the community.

Ceilometer has three different techniques to collect data:

- listening: listening to events being generated in OpenStack and creating a Ceilometer sample out of it
- push: have a component to push specific data of specific projects.
- poll: actually polling APIs to obtain necessary data.

Collected data is stored in a database generally. Multiple types of databases are supported through the use of different plug-ins. Through a REST API, this information is accessible to any external system consumers.

The ceilometer framework provides for a multi-publisher method whereby a technical meter potentially reports to different consumers at different frequencies and using different transports. Ceilometer possibly provides input for billing, accounting, capacity planning, operational management, etc.

The alarming component of Ceilometer provides, in a configurable manner, threshold evaluation of sampled data, which is for example used to provide auto-scaling in HEAT (for details, see [LV11]).

Copyright © FUSION Consortium, January 2014

4. FUSION SERVICE SPECIFICATION

This section discusses the key aspects regarding FUSION application services. This includes service composition, service description, efficient communication protocols and the initial FUSION application service interfaces.

4.1 Service composition

In FUSION, we envision supporting different types of composite services, consisting of multiple service atoms that are connected to each other in a service graph, and which can be deployed in a distributed manner across multiple execution points and execution zones. This section discusses the overall scope and flexibility of service composition within FUSION and how it may impact the design or implementation of several key FUSION functions. We start by discussing possible types of service composition and their impact.

4.1.1 Service graphs: describing composite services

A composite service in FUSION could have both a static as well as a dynamic service graph. In static service graphs, the service atoms and their connections are known beforehand (e.g., via the service manifest) and exist for the entire lifecycle of that composite service. In dynamic service graphs, the service atoms and their connections may vary during the lifecycle of the (dynamically changing) composite service. In between, we also envision service graph templates, where the service atoms and their potential connections are known upfront, but which may change dynamically during the lifecycle of the composite service. In the following sections, we will discuss the various identified options in more detail. In Section 4.2, we describe how such services can be described in service manifests.

4.1.1.1 Variant A: Instantiating service graphs

A pure implementation of the static deployment without any dynamic aspects may work in the following way. The orchestrator gets a service graph description, for example described in a manifest, and then calculates the optimal instantiation of this service graph as a service instance graph. Depending on the request it may create for example one or a given number of service graph instances. The main challenge of the orchestrator is to calculate the optimal mapping of the service instances to execution points based on given requirements, e.g., QoS requirements for given user locations in the network.

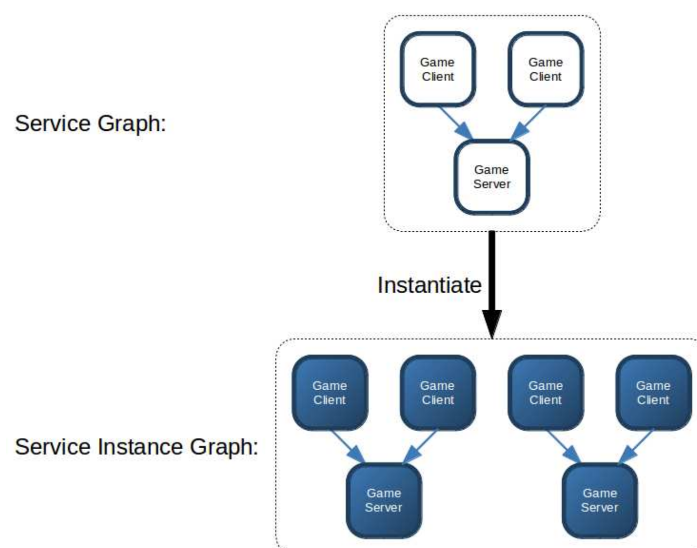

Figure 17: Variant A, instantiating static service graphs

Copyright © FUSION Consortium, January 2014

The advantages of this approach are:

- The orchestrator has a complete picture for calculating the best mapping of the service instances to the execution points.

Disadvantages are:

- This approach in its pure form does not allow to connect new service instances to existing services. For example, this approach does not support connecting a newly deployed MMO client service (connecting to a thin user client) to an existing MMO server service instance.
- This approach does not support that services connect to other services dynamically later. For example, an EPG dashboard user may choose a TV channel or a VOD movie to connect to.

4.1.1.2 Variant B: Instantiate and integrate service graphs

To overcome the limitations of the previous pure approach the service graph description may support connections to already existing services. In contrast to the purely static approach this means that the orchestrator needs access to the currently running service instances.

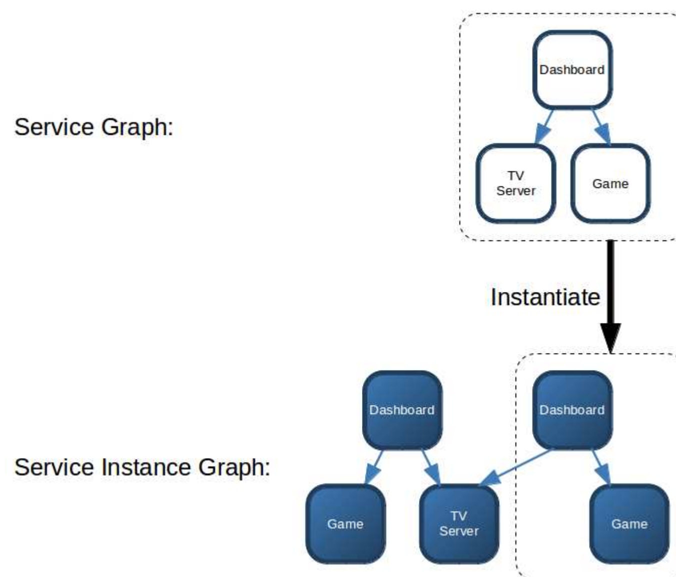

Figure 18: Variant B, instantiating and integrating static service graphs

The advantages of this modified approach are:

- The orchestrator has a full picture for calculating the best mapping of the service instances to execution point, knowing both the service graph of the desired new service instances plus the currently running services to connect to.
- New service instances can connect to existing service instances. For example a new game client service instance can connect to an existing game server service instance, or a EPG dashboard service can connect to running VOD services.

Disadvantages are:

- Orchestrator does not know about future instantiation. For example when instantiating a game server instance the orchestrator does not know about upcoming requests of game client service instances, or when instantiating an EPG dashboard service it may not take into account to which VOD services the EPG dashboard will likely connect in the future.

4.1.1.3 Variant C: Stepwise instantiation and integration of service graphs

The limitations of the previous approach can be overcome by providing the orchestrator also information about future instantiations. For example, the orchestrator gets a service graph containing both a game server service and for example four game client services, but then the orchestrator only instantiates the game server service alone, but taking possible future game client service locations into account. Then the game clients are only instantiated when needed. In other words, by providing the orchestrator with a service graph of the full future planned or possible situation beforehand, the orchestrator can optimize the mapping of a selected subset of service instances to execution points by taking the future instantiations into account.

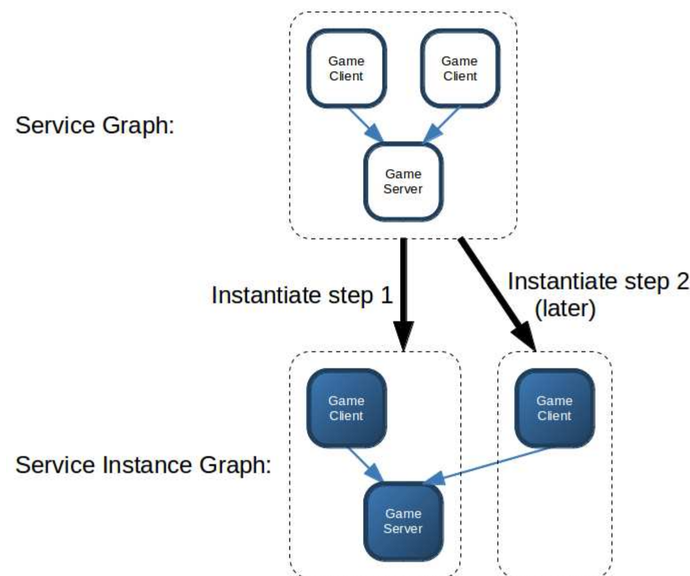

Figure 19: Variant C, stepwise instantiation and integration of service graphs

The advantages are therefore:

- This approach provides flexibility for optional future instantiation, e.g. the option to add another player for games working both with three or four players.
- The orchestrator not only supports, but also knows about planned future instantiations.
- There is no need to instantiate services that are not needed at the moment or that may be not needed in the future at all.

However, the disadvantages are:

- The instantiation of service graphs is restricted to a pre-defined set of service graphs. There is no flexibility to dynamically add any kind of service on user-demand. For example you cannot connect any kind of VoD server or a new game client to an EPG dashboard if this was not foreseen in the static service graph, or connect more game clients service instance to an running game server service instance.
- The orchestrator may always lack crucial information about future instantiations. For example the service graph may tell the orchestration that four game client services will be likely instantiated and connect to the game server service instances, but the graph does not contain information about the time and location of the players who will connect to the game client service. Similarly, even if the service graph informs the orchestrator that the EPG dashboard service instance will possibly connect to a VOD server service in the future, it may not know

Copyright © FUSION Consortium, January 2014

which VOD contains the desired movie and therefore to which VOD it will have to connect in the future. Therefore the orchestrator cannot make the best decision about future instantiations.

4.1.1.4 Variant D: Instantiation of individual services

The other extreme is a purely dynamic approach: There is no concept of service graphs. Instead, the orchestrator instantiates each individual service alone, taking possibly all information of all running service instances into account, plus the requirements of the single new service to instantiate. The approach is very similar to object oriented programming: each service knows to which other services it can connect or which other services it may want to instantiate in principle, but without restrictions as to whether and when such connections will be established or instantiations will be requested; this is similar for example to Java where a Java class knows which references to other Java objects of which types it supports, but without defining at compile time when and which Java classes are instantiated and how they are connected at run-time.

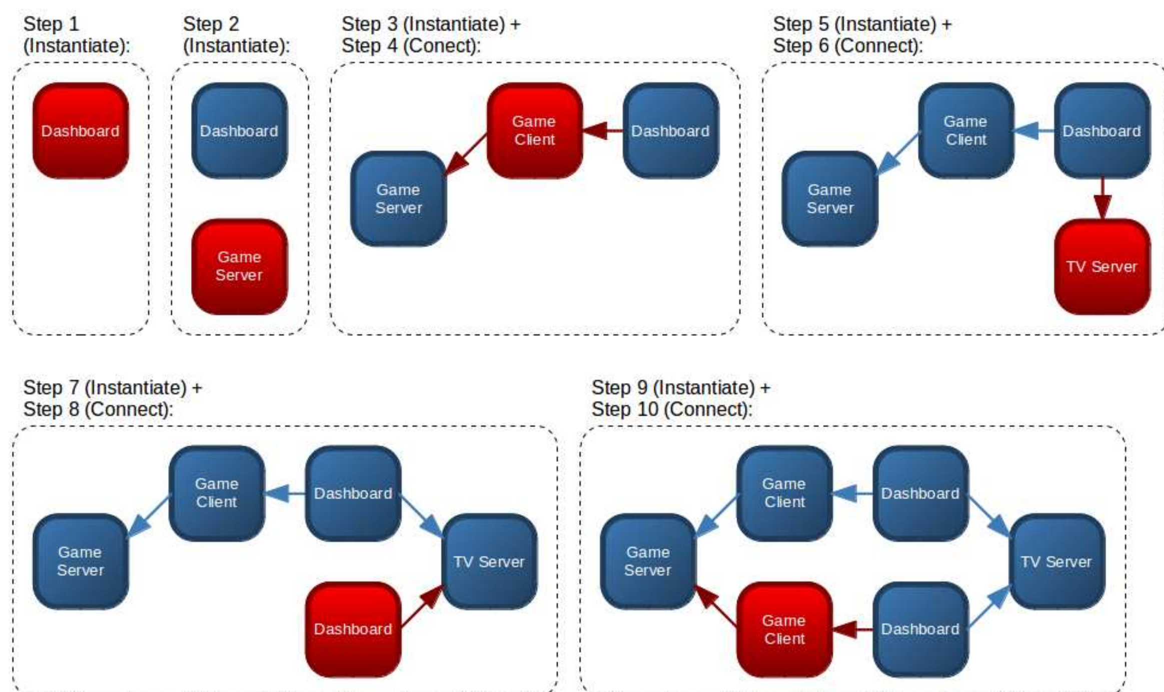

Figure 20: Variant D, instantiating of individual services

In practice this also means that service instantiation and establishing connections between services may be considered as separate steps, which may be invoked at any time at run-time separately:

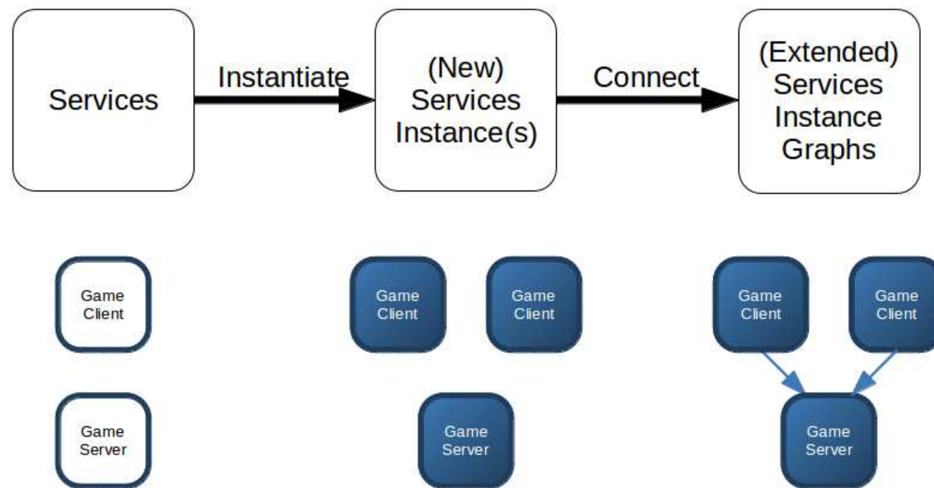

Figure 21: Three-step dynamic service instance graph construction

The advantages of this approach are:

- This is by far the most flexible variant: there are no restrictions on the service instance graph, the service instance graph can be extended by new service instances, and new connections can be created between service instances as needed. For example, a running EPG dashboard service instance can connect at any time to new VOD server service instances, it can request the instantiation of new game server instances or game clients instances, it can create connections to other dashboard instances for screen sharing, etc. Some of these use cases are described in Deliverable D5.1.
- The approach is very simple: just add a new instance based on the current situation.

Disadvantages are:

- The main disadvantage of that approach is that the orchestrator never has a global picture available. While the orchestrator has a complete picture of the current situation, it cannot consider more than one future services at once for optimal mapping to execution points. For example, when instantiating a game server service it cannot take into account upcoming instantiations of game client services. Even an optimal ordering of service instantiations can lead to sub-optimal results, as analysed in Section 2.8.

4.1.1.5 Variant E: Future-looking instantiation of individual services

The restrictions of the previous purely dynamic approach can be lifted by providing the orchestrator with additional information about upcoming service instantiation requests (or access to pre-instantiated services). For example, if a game lobby software wants to instantiate a game server service, it may tell FUSION about all parameters required for optimal placement, for example about the game client services, the locations of the players to connect to and the hardware requirements of the game clients, see Figure 22. To avoid that FUSION has to know service specific information, a smart communication between the FUSION orchestrator and services may be necessary.

Copyright © FUSION Consortium, January 2014

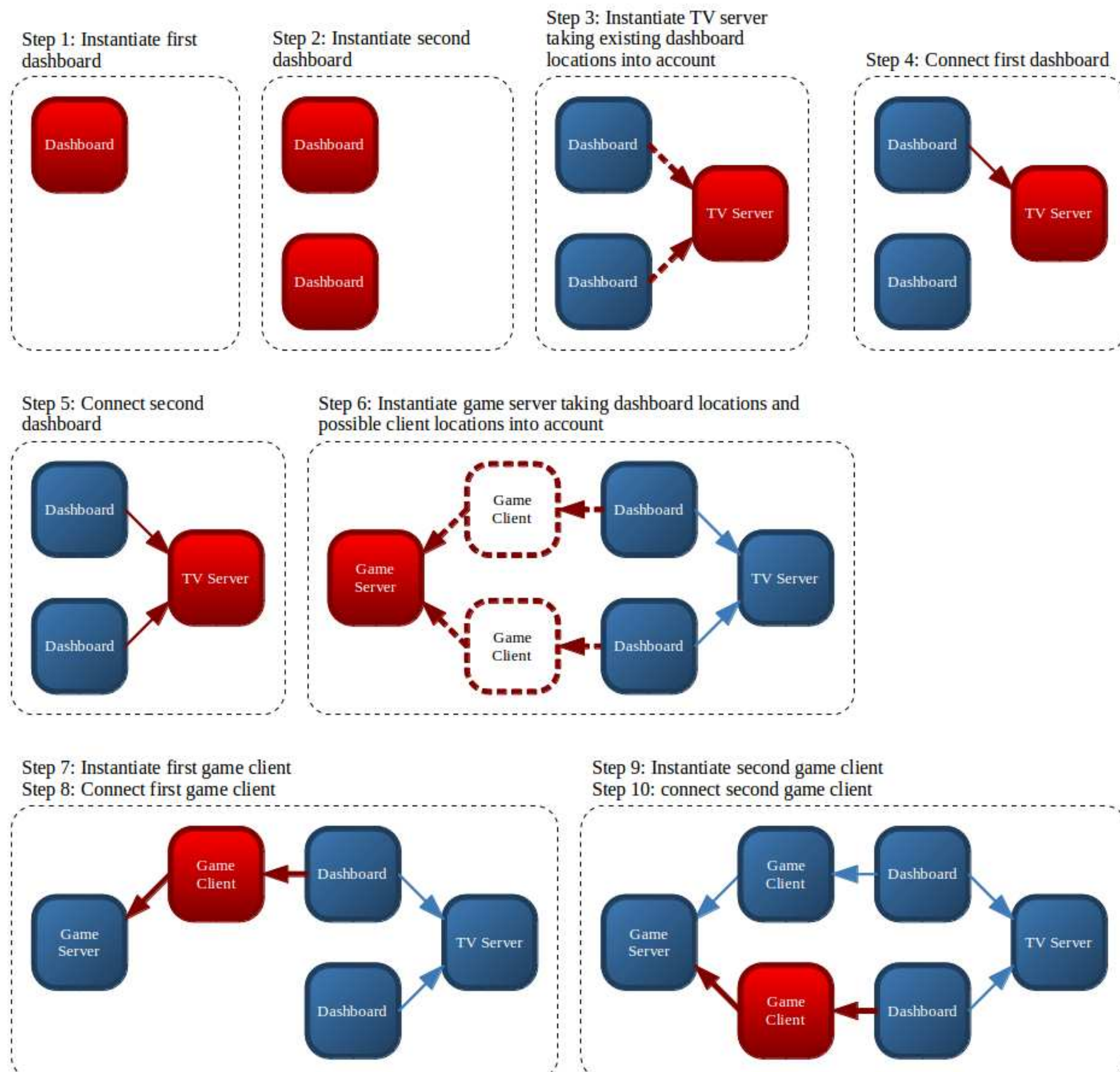

Figure 22: Variant E, future-looking instantiation of individual services

Therefore this approach is not a pure dynamic approach as the previous one, but also has static elements. However, in contrast to the service graph approach, static information is stored and used in different ways:

- While service graphs contain static connection information explicitly in one place (e.g., in a manifest), in this variant the static information is contained indirectly in the service probe implementations.
- Since the information is contained in probe code instead of static descriptions, it can contain more information, for example proprietary hardware probing FUSION cannot know about current load information.

The advantages are:

- This approach combines the flexibility of the purely dynamic approach with advantages of the static approach.

Copyright © FUSION Consortium, January 2014

- The orchestrator has a better picture of planned instantiations in the future (as long as it is known and communicated by someone, for example by the application provider or application itself).
- New instantiations and connections are possible in the future on user-demand, even if they are not known before (e.g., connecting new game clients, new connections to VoD servers, etc.)

The disadvantages are:

- Due to the mixture of static and dynamic elements it is more complicated to implement than a purely static or dynamic approach.
- Due to the flexibility of having service specific information stored in the probes, the implementation of the FUSION orchestrator and its communication with the probes may be more complex than just using a static manifest.

4.1.2 Granularity

In FUSION, we allow for flexibility regarding the granularity of individual service atoms within a FUSION service graph. To be able to better distribute and optimize individual service atoms across one or more execution zones, the composite service should be described as a set of service atoms, according to the need for distribution and reuse. If a FUSION service is described as one monolithic service, then that service obviously cannot take advantage of any distributed placement across the network to take full advantage of the network and compute resources. Consequently, within FUSION, we are looking into light-weight virtualization and deployment models and late binding that allow to also efficiently deploy light-weight service atoms across one or more execution zones and have efficient inter-service communication when deployed in the same execution zone.

4.1.3 Deployment strategies

We have identified basically two deployment philosophies:

1) Static deployment: The FUSION orchestration gets a problem description like a request of deploying or instantiating a set of related FUSION services, described for example by manifests, service graphs and parameters like the number of desired instances, distance and QoS requirements. Then FUSION can calculate the best graph of instantiated services on specified execution points. Applications using FUSION services then connect to these existing service instances using FUSION routing, which is basically independent from the orchestration step. Core properties of this approach are:
    - Developers registering services basically only have to deal with the FUSION orchestrator by telling the orchestrator about the new service. Software using FUSION basically only has to deal with the routing part of FUSION.
    - Orchestration is technically decoupled from routing in that sense that both algorithms do not have to work together handling a given service instance and that they are invoked at different times: the orchestration module of FUSION just informs the routing module of FUSION about the locations of all service instances, and the routing module uses that information to do the routing.
    - The orchestrator has the perfect view and theoretically can make the perfect decision about service instantiation.
2) Dynamic deployment: basically each time a software requests a FUSION service, not only FUSION routing to a given service instance is invoked, but the FUSION orchestrator also decides about deploying and instantiating new services at the best location as necessary. Core properties of that approach are:

Copyright © FUSION Consortium, January 2014

- Developers registering a service just upload information to a registry, but do not trigger orchestration or routing. When a software requests a service, then both orchestration and routing is activated.
- Orchestration and routing are more closely related: routing decisions are not decoupled from orchestration, because a service request does not only require the intervention of routing, but also that of orchestration.
- The orchestrator gets information only incrementally by each service request and never has the full picture. Therefore the orchestrator has to make decisions by taking assumptions and using available and predicted information.

These philosophies are not mutual exclusive, but can be mixed in various different variants. The following sections discuss some exemplary variants.

4.1.4 Service binding & lifecycle management

With composite services, the individual service atoms in the service graph can be loosely or tightly bound to the service graph. With loose binding, we mean that the lifecycle of the service atom instances can be completely independent from the lifecycle of the overall composite services. This means that instantiating or terminating a composite service does not necessarily imply the automatic instantiation or termination of every individual service atom. Loosely coupled service atoms enable to easily reuse existing instances across multiple service graphs or composite service instances, dynamically linking multiple service graphs, or simply to reuse existing instances and session slots across multiple composite instances for higher efficiency.

With tight binding on the other hand, the lifecycle of the individual service atoms are tightly coupled to the lifecycle of the composite service: due to their inherent function, the service atom instances have no reason for existence outside the scope of the composite service. The justification for having these individual atoms instead of just defining the composite service as one monolithic piece is for example because of the distribution of the service graph across multiple execution zones for latency or bandwidth reasons.

Although we envision typically more loosely bound service graphs, where particular atoms can be shared among multiple composite service instances and types, FUSION should also support the more tightly bound composite services, in which case the lifecycles of the individual atomic service instances are more closely linked to each other. This has its implications on the overall domain-level orchestration and lifecycle management of distributed composite services, as in case of tightly bound service atoms, FUSION needs to keep track of the overall graph, and coordinate appropriate actions and state changes across multiple execution zones. In case of loosely bound service graphs, each atomic service instance can be managed separately, and FUSION does not need to keep track as much of the graph itself, but rather of the individual atomic instances. We refer to Section 4.1.6 for more information regarding the addressing of composite services. In the course of the project, we will elaborate on these options in more detail an study their implications on the overall management within FUSION.

4.1.5 Distribution

In FUSION, we do not only want to deploy multiple instances of a service type in a distributed manner across multiple execution zones, we also want to be able to deploy the service atoms of a composite service in a distributed manner, both inside an execution zone as well as across multiple execution zones. As discussed in the previous section, depending on the binding and role of the individual service atoms within a service graph, FUSION will need to orchestrate the entire service graph, only portions of the graph or just the individual atomic service instances. This has a significant impact on the overall scalability and complexity of the FUSION orchestration and lifecycle services.

Copyright © FUSION Consortium, January 2014

A second key aspect related to service distribution is the networking and routing in between individual service instances. In case a composite service is deployed in a distributed manner, or constructed by connecting existing service atoms, FUSION needs to make sure that the network characteristics in between these distributed zones meets the overall service constraints. Consequently, this will play a key role in the placement or service selection operation.

4.1.6 Addressing composite services

Composite services by definition consist of multiple service atoms, each of which may or may not be directly addressable. For loosely bound service atoms that may be reused outside one specific service composition, FUSION should support addressing both the individual service atoms as well as the composite service. In case service graphs consists of tightly bound service atoms, the individual service atoms in many cases should not be publicly visible or addressable; instead, all service requests should be made using the name of the composite service. In both cases however, the service request for the composite services will need to be forwarded or passed through to the corresponding service atom that will accept and address the service request. This can be done in a number of ways, both of which are depicted in Figure 23:

- One option is to allow FUSION services (service atoms in particular) to implicitly or explicitly inherit the name of the enclosing composite service. In case when such "renaming" is explicit, the service atom is aware of the alternative name, whereas in case of an implicit "renaming", this can be done completely transparent from the service atom, for example by the service gateway function in the execution zone during service deployment. The latter case requires NAT-like capabilities for FUSION names, allowing to automatically translate incoming requests for composite service C to be translated into requests for atomic service A. In both cases, the original name of the service atom may still be publicly addressable or not, enabling or disabling direct access to the service atom, independently from the composite service.
- Another option is to insert tiny glue services that explicitly forward the service requests for the composite services towards the appropriate service atom that is part of the service graph. In this case, it is the responsibility of the service provider for adding the necessary glue services and to ensure that these glue services are collocated with the service atom that is responsible for handling the service request. These glue services may involve some overhead, as the request (and perhaps also the data in a later stage) need to pass through such glue service.

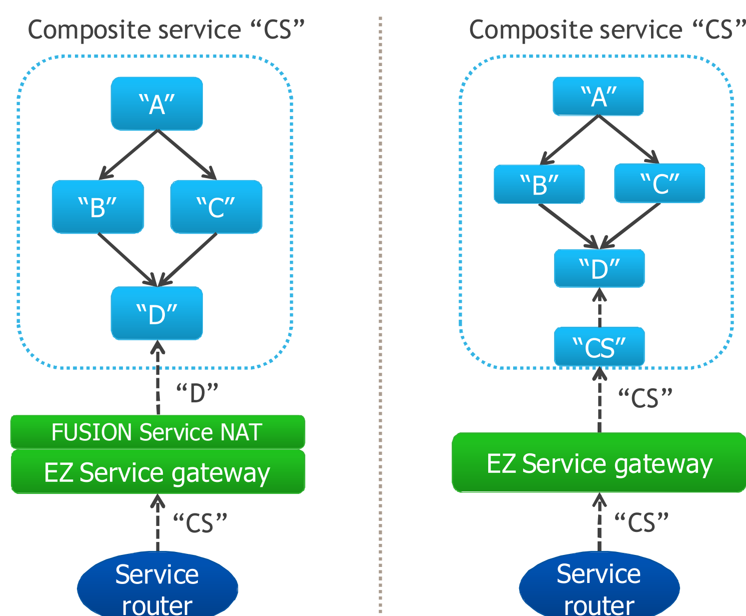

Figure 23: Two approaches for forwarding service requests to composite services.

In case the composite service only exposes a single service endpoint (meaning that each service only exposes a single function), the service requests should be forwarded towards the location of the corresponding service atom that will eventually handle these requests, both in case of the FUSION NAT approach as well as glue services. In the latter case, the glue service should always be collocated with the service atom to minimize the overhead and latency.

FUSION services however can also have multiple endpoints (meaning that different functions provided by a single service may be accessed by one or more client applications or other services, for example using different IP addresses or ports). In this case, the situation becomes slightly more complex, as different endpoints may be deployed on different execution zones. This means that the FUSION service routers need to be able to route service requests with a similar service address but different ports to different execution zones. The overall mechanism is shown in Figure 24.

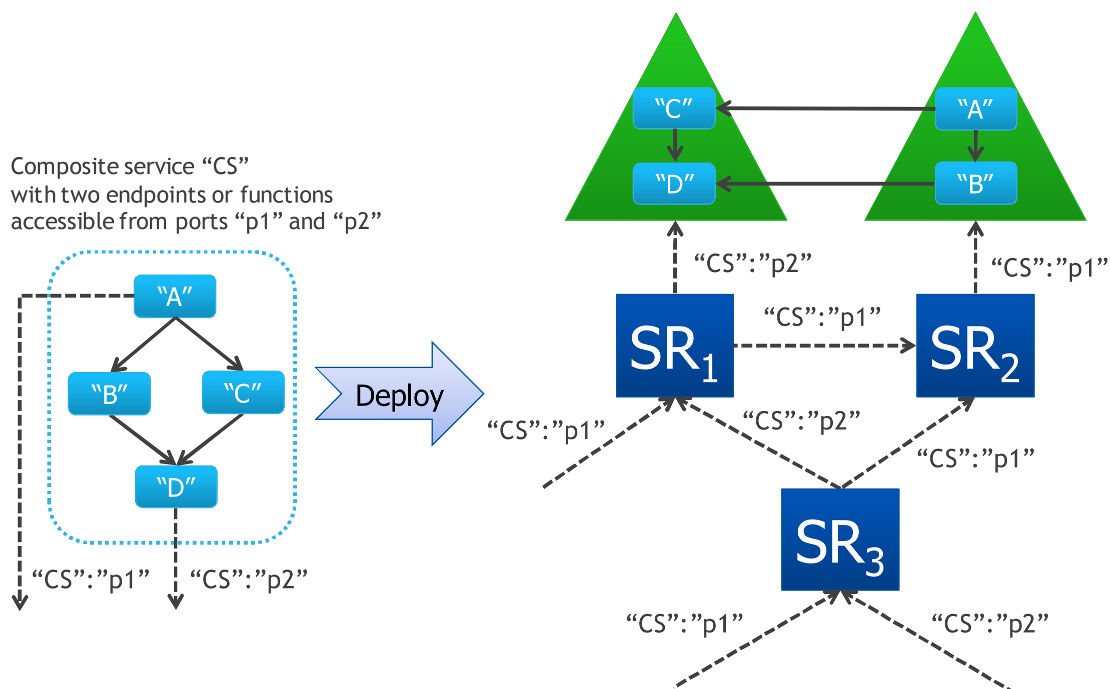

Figure 24: Service routing for distributed composite services with multiple endpoints

4.1.7 Level of interactivity

Different services in a FUSION service graph can support different types of interactivity with respect to the other services in the graph. On the one hand, there are the more static services or content produces where other services connected to that service do not control the content it is producing or streaming. Example communication protocols include streaming, pub/sub, etc. Other services are more interactive and allow other services to directly impact the operation and output of these services. Example communication protocols that support this mode are request/reply for short-lived requests, or a special feedback channel to forward input events, video input or other commands back to that service. This may impact the deployment and communication model when deploying such a service graph into a FUSION domain.

4.1.8 Constraints

In case of composite services, many of the service constraints (functional constraints and non-functional constraints) do not only apply to the composite service as a whole. There may also be specific constraints related to the individual service atoms in the service graph as well as for the individual connections in between these service atoms. Examples, include a latency and bandwidth

Copyright © FUSION Consortium, January 2014

requirement in between specific service atoms, runtime constraints for individual atoms (e.g., one atom requires a lot of CPU resources, another requires a lot of disk I/O), etc. The service developer and/or service provider needs to be able to specify this, and FUSION needs to be able to take these sets of constraints (both at the atomic level as well as the composite level) into account when deploying and managing the composite service.

4.2 Service description

A service manifest is a static description of a service, including all its dependencies, requirements, constraints, business aspects (cost, etc.), parameters, etc. It contains all functional and non-functional requirements that are required to be able to automatically and optimally deploy and create instances of this service in execution zones in FUSION domains. The service manifest should be described in a format that is understandable by FUSION to enable fully automatic deployment and instantiation.

The manifest likely consists of several parts, each of which could be provided in separate documents, and each of which is only relevant (or perhaps even visible) to the relevant parts of FUSION or its stakeholders. For example, there will be sections about deployment, how many resources to allocate, how to monitor, etc. Other parts of the manifest should also be made public so that other services, potentially from other service providers, can communicate with that service or even integrate that service into a more complex composite service. For example, sections that describe the public API of that service, including the communication API, the service functionality, etc. should be visible to all other services or providers who are authorised to interact with this service type.

The service manifest can only contain the static aspects that are known in advance, and that can easily be described in a manifest file. In many cases however, this is insufficient or too complex. As a result, we also rely on the concept of evaluator services, which is a dynamic, executable version of a service manifest that can complement (or help interpreting) the static service manifest file. See Section 5.4.2.2 for more details on these evaluator services.

Below a non-exhaustive list of categories.

4.2.1 General information

This section of a service description captures the basic information of a service, including its name, a description of what the service does, keywords, version information, etc. It can also contain information concerning for example the service type and statefulness.

4.2.2 Service graph

This section of a service description contains the key information about how services are connected and how they communicate. Key elements in this section include the overall composition of the service (is it a composite or an atomic service), the (known) dependencies with respect to other external (FUSION or non-FUSION) services, etc. It also contains a detailed description of all (known) input, output and control channels. Each control channel itself is described according to a number of key elements. These include the channel type (input, output or control), the service identification (service ID, URI, URL, etc.), a communication path description (URL), a transport protocol description (FUSION or other, e.g., RTSP, HTTP, etc.), a description of the data format (e.g., an MPEG-TS media container), and a description of the resource utilization and constraints (bandwidth, maximum latency, etc.)

4.2.3 Deployment and platform dependencies

In this section of a service description, all static/known dependencies with respect to the execution layer are described. This includes the software, hardware and functional dependencies. The software dependency section details what software packages it needs to deploy, and where to find it (possibly

Copyright © FUSION Consortium, January 2014

including kernel version, etc.). The hardware dependency section describes what hardware resources this service may want to use. This can be either a hard constraint, or merely describing a particular affinity towards a particular piece of hardware, including its type, model, version, capabilities, for both CPU and other accelerator hardware. Third, this section may also expose dependencies regarding some key high-level functions this service might use. A typical example is video encoding or decoding, for which a particular execution zone may have specific accelerated support for. This section may also include specific requirements concerning each of these functions (e.g., specific encoding capabilities, like the codec, encoding parameters, etc.).

Note that we don't expect FUSION to understand all these parameters and constraints of all these dependencies. However, they may be key input for the evaluator services when evaluating the effectiveness of a particular execution zone for a particular service type.

4.2.4 Lifecycle management

This section of a service description includes information that can help the lifecycle manager, including the expected duration of the service, start/shutdown scripts, etc.

4.2.5 Resource usage

This section of a service description contains (a link to) information concerning the resource utilization of the service type. This can range from a more high-level service profile label (cfr. the 'm1.large' instance type), a resource utilization profile (containing relative utilization information, broken down by each key hardware resource type, like CPU, memory bandwidth and footprint, disk, networking, GPU, etc.), towards historical monitoring data that is captured during previous deployments of the service type. Again, this information can be used by the evaluator service, both for evaluating the effectiveness of a particular execution zone as well as estimating how many parallel service sessions can be supported with a particular resource budget (which is related to a particular cost on a particular execution zone).

4.2.6 Monitoring

This section of a service description can contain a description of the service and resource parameters that should be monitored, and optionally the boundaries for each parameter within which the service should operate. An example service monitoring parameter is the FPS, and a possible boundary could be minimum 20 FPS upon which the system may need to act when the threshold is crossed.

4.2.7 Mapping

This section of a service description contains relevant information to enable service placement. This information can range from a static or fixed (set of) mapping location(s), to information that can help the dynamic evaluator service in evaluating the placement of the service in a particular execution zone. This can also include preferences and priorities for the orchestration to help deciding on the execution zone (e.g., choose the cheapest zone, or choose the zone that provides the lowest latency within a particular budget).

4.2.8 Policies

Due to the inherently dynamic nature of service deployment in FUSION, service providers registering new services in a FUSION domain should be able to specify in detail what the various policies are for that particular service type. This set of policies cover various aspects, including service deployment, availability, reliability and QoS, business and cost related policies, security and privacy related policies, etc., all of which FUSION needs to conform to and for which a service provider should be able to verify whether these policies are respected.

Copyright © FUSION Consortium, January 2014

4.2.9 Security and privacy

In this section, all security, privacy and visibility related aspects are described, including who can access what parts of the service, restrictions on the environment, etc. Things to consider here, include the following aspects:

- Geographical deployment constraints
- Legal constraints
- Isolation constraints, which may constrain the used virtualization technique
- Service content restriction description, e.g., an EPG may not be allowed to subscribe certain channels due to legal constraints in countries where the user accesses the EPG service

Poaching is the misuse of information provided for one purpose but used for another in a way that harms the client. This impacts the monitoring component of FUSION. Cloud vendors may also perform data mining in aggregate and learn characteristics of a clients' own customers, products.

4.2.10 Business aspects

As FUSION will deploy many of these services dynamically and fully automated, it is crucial that a service provider can constrain FUSION on how, where, when and how many instances it can deploy at all times. This section will typically also contain a cost model, so that the service provider can constrain the cost for deploying its services. Again, this could be augmented with dynamic information from an evaluator service that evaluates and calculates the cost on-the-fly.

The business aspects of a FUSION service deployment may relate to several aspects. A first aspect is regarding a billing service present in a FUSION orchestration domain. Billing has impact on resource consumption monitoring and reporting, which could be based on for example:

- A flat fee per session (requires per session reporting capabilities);
- A fixed cost for session setup, with an additional per-unit-time incremental cost;
- The actual resource consumption of a service onto a particular platform;
- Other possibilities

To enable accurate billing, relevant monitoring information needs to be provided by the execution zones regarding the resource utilisation of each service instance. Note that in case particular instances are shared by many services, more complex billing schemes may be required to be implemented by the service provider that is providing the shared service. This may result in additional constraints.

There may also be constraints with respect to the execution zone that are related to cost. This may be related to specific compute, storage, networking and accelerator capabilities that may exist in particular execution zones.

FUSION needs to know about these business constraints deployment (e.g., to setup monitoring in order to have correct input for billing, or to constrain the deployment across execution zones). Therefore, these business constraints must also be described in the service manifest and possibly partially also in the evaluator services. The manifest of a service could specify instructions on how the monitoring and billing should be configured.

Concerning licensing issues associated with a service, we assume this is the responsibility of the services to ensure all licensing is ensured. This may however also result in additional deployment constraints, for example with respect to the countries in which a service may be deployed.

Copyright © FUSION Consortium, January 2014

4.3 Service sessions

In this section, we will elaborate on the concept of a FUSION service session and FUSION session slots as a key enabler for efficiently handling many of the key functions of FUSION with respect to the resource utilisation of service instances.

Each service instance can typically handle a particular number of service requests in parallel. We define a FUSION service session as the period of time in which a (set of) communication channel(s) exists between a client and the service instance that handles the service request. Note that a service request could consist of multiple messages going back and forth between the client and the session instance. Note also that a FUSION session should not be confused with an application session, which may comprise one or more independent FUSION service sessions. Each FUSION session involved in the application session can be considered independent from the other sessions and thus could be handled by different service instances. Similarly to HTTP requests, the FUSION sessions could be linked at the application level by using a cookie or session identifier.

When a new service request is issued by a client (or another service), the service routers need to decide very quickly to what service instance to route that particular request. For this, it needs to take into account many factors, including the overall load of a particular service instance. Indeed, it is not useful to route a service request to a service instance that is already fully loaded, as this could have detrimental impact on all service sessions that it is already handling. However, for many reasons, including scalability and privacy, it may not be practical and probably also not feasible to send high-level or low-level resource utilization data from an execution zone to the domain-level service routers or orchestration.

To solve this issue, we propose the concept of FUSION session slots. The core idea is that during service instantiation, each service instance allocates enough resources for handling roughly *N* FUSION sessions in parallel. We thus say that the service instance has *N* available session slots. Each time a service instance is handling a service session, it takes up one of the available session slots for the duration of the session. This information (i.e., actual, average or predicated availability count) needs to be forwarded and distributed towards the domain-level service routers, which can use this information when forwarding a new service request towards a particular instance. Note that if during execution it seems that the actual resource utilization does not match the predicted resource utilization, then the available session slots could easily be updated on-the-fly using the same mechanism.

Session slots will be used in the service routing plane as an additional metric to be combined with network metrics for determining the most appropriate execution zone to forward service requests towards. Routing trade-offs between maximising network performance (in terms of delay, throughput, etc.) and available service processing resources (in terms of available session slots) may vary depending on service type, including the longevity of the service sessions. Approaches for handling these trade-offs as part of multi-metric routing decisions as well as techniques for efficiently dealing with routing decisions per service type are being developed as part of the anycast routing algorithms and protocols being studied in WP4, to be reported in deliverable D4.2.

4.3.1 Service Session Implementation

Two key aspects with the concept of session slots are (i) that during the resource allocation step when instantiating a new service instance, the amount of allocated resources need to be linked explicitly with a particular number of session slots, and (ii) that the number of available session slots needs to be monitored and distributed towards the domain-level service routers on a regular basis. To avoid flooding of these session slot update messages across the domain-level overlay network (for example in case of short-lived sessions), average numbers or even predicted availability numbers could be distributed, rather than an accurate actual figure, which may be already outdated when it reaches the closest service router.

Copyright © FUSION Consortium, January 2014

To keep track of the service session a particular client is connected to, a FUSION session identifier could be generated and associated with every new service request. Although the session identifier is probably not relevant for the service routing itself, it may be relevant for the service instance, to know what service session your service request belongs to, for example when linking multiple client requests to the same FUSION session. Remember that a FUSION session is different from an application session.

4.3.2 Benefits

The concept of service sessions and the corresponding session slots is applicable both for request-response type of services as well as streaming services, though their behaviour might be quite different. With the former type of services, service sessions could often be very short-lived, whereas with the streaming services, service sessions will typically last for several minutes, hours or even days. To manage these long-lived service sessions, service factories could be used, although this should be invisible from FUSION. FUSION could provide some API support however to restrict the amount of resources per session in a light-weight fashion.

Although this approach still needs to be worked out in detail, the notion of service sessions and session slots could significantly simplify many of the core FUSION operations, which is discussed in the following sections.

4.3.2.1 Light-weight service request routing

With this approach, service routers only need to be aware of the number of remaining available parallel sessions that each service instance still can handle, which boils down to a single value per instance. This information would be almost the only monitoring information coming from the execution zones that need to be broadcasted towards the domain-level service routers, making it very simple, concrete, and manageable.

4.3.2.2 Separation of resource allocation and service request routing

In this approach, the resource allocation is done upfront during service instantiation, which is coordinated by the orchestration layer. During service routing, the service routers do not have to take into account the available resources of each zone, the requirements of each service type, etc. This makes the service routers much faster and simpler. The major disadvantage of this approach however, is that the average amount of session slots that a service instance can provide for a particular amount of resources needs to be more or less predictable. In worst case however, a service instance could always start with a conservative estimation, and dynamically update this estimate along the way (for example, when the service instance detects that the current active sessions only use a portion of their estimated or assigned resources).

4.3.2.3 Hierarchical aggregation

The approach also allows to aggregate the available resources very easily at multiple levels: you just add them all up per unique service type. For example, if a zone has multiple instances of the same identical service, each supporting a particular number of sessions, then you can just add up each of their remaining available parallel sessions and present this aggregate number to the domain. The domain and the service routers do not even have to know that the available session slots come from multiple service instances within a particular zone. Even inside the overlay network of a (large) domain, service routers do not have to keep track of every single number from every single zone. Instead, for each link to another service router, they could keep track of the number of the number of available slots that are within range when forwarding the request along that link. In the inter-domain case, a domain could choose to only make a subset of all available session slots visible/accessible to another domain, for only a subset of all service types.

Copyright © FUSION Consortium, January 2014

4.3.2.4 Service-type neutral

As already discussed above, the concept works both for request-response as well as streaming services. The former type of service typically will be able to handle a large amount of parallel sessions with moderate resources, whereas the latter service type will typically only be able to handle a few sessions in parallel. Another key difference is the potential frequency with which the session slots are (de)allocated, as streaming services will likely be much longer-lived than request-response services. See the next section how this can be taken into account.

4.3.2.5 Stability

Service routers probably should implicitly or explicitly be aware of the duration and demand of each service session, to be able to estimate how accurate the number of available session slots is when an update reaches that service router. For example, it could keep track of the moving average and variance to estimate the probability that a particular service instance will still have available sessions when the request is forwarded to that instance. Indeed, the routers will typically always have outdated information, as in many cases, many requests can occur in parallel, each coming from different routers that are unaware of each other's choices. During periods of stability (i.e., low variance), it may be possible to use all available session slots. During periods of instability (i.e., high variance and low predictability), it might be better to choose a zone that has more available session slots available for that service type, rather than trying to use the last remaining session of the closest zone, as the session slot might already be taken by the time the request reaches the instance.

4.3.2.6 Auto-scaling

The decision of whether and when to request FUSION orchestration to start a new instance of a particular type also becomes easier, as one can simply keep track of the evolution and/or history of the available sessions (across the domain) and proactively start and/or stop new instances or even request particular service instances to increase or decrease their available session slots. In other words, the concept of session slots is a very simple metric to scale-up or scale-down the available session slots and service instances.

4.3.2.7 Billing

Formalizing the resource allocation and making it an explicit step upfront makes it easier to setup up appropriate billing, where the service provider knows how much he will have to pay for running the service instance. This is also applicable in case the service instance (or another service instance that monitors the instance) can request for more or less resources: by making the resource (re)allocation explicit, it becomes more transparent and easier to manage. If the routers would have the (sole) capability to decide to what instance to forward the request, this might lead to an aggressive over-allocation strategy, or prevent the application developer to have control on the resource utilization (and eventually the bill).

4.3.2.8 Trust

Another key benefit of this approach is that a zone never has to publish its available resources to the domain for handling the requests to other zones or domains to other domains. Instead, it can just broadcast for each public service how many sessions the zone or domain can provide, which could be a smaller number than in reality, for example triggered by SLA agreements.

4.3.3 Issues for further investigation

Although the concept of FUSION service sessions and session slots simplifies and solves a number of key issues, there are still a number of aspects which have to be solved based on further investigation and feedback from prototypes and first implementations.

Copyright © FUSION Consortium, January 2014

- Service instances, execution zones or a domain can at all times announce more available session slots than it can handle. This can be either a deliberate oversubscription strategy or also simply caused by the fact that the service has no complete and accurate view of the system it is deployed on, the interference caused by other applications, a partial or inaccurate view of the required resources per session. Though the root cause is different and thus the countermeasures may be slightly different, this issue is similar to the issue of obsolete FUSION routing tables. An example countermeasure might be to avoid "ghost" sessions slots from a particular service type from a particular execution zone or even domain.
- Another issue is who is responsible for determining the amount of sessions that a particular service type can provide for a particular amount of resources. There are a number of options. One option is to start with a particular amount of resources, for example captured by a resource profile (like many cloud providers do), and to associate a number of session slots with each profile. Another option is to start from a number of session slots and request a particular amount of resources accordingly. The advantage of the former option is that is simplifies resource allocation in the execution zones, as well as for the service types, which can determine the average number of session slots with each profile offline or online via monitoring and profiling. However, is does restrict overall flexibility. The latter option on the other hand provides optimal flexibility, however at the expense of more complex resource management inside the execution zones, as well as most likely more complex resource estimation for each service type.
- As with all cloud infrastructures, it is hard to estimate in advance what the interference of one type of application will be on another when they are co-located on the same physical infrastructure. This may impact the effective amount of session slots that can be hosted by a service instance with a particular set of resources. The application or an external load balancer could keep track of this by taking into account overall load and application health, and adjust the available session slots accordingly.
- More in general, there is the fundamental dilemma of who is responsible in case a particular performance target is not met. This question is related to whether FUSION (or any cloud provider) will ever be able to provide performance guarantees beyond best effort to the applications it is hosting. For example, via profiling, a particular service type expects to be able to run *N* sessions in parallel with a particular resource profile. However, when the instance is deployed inside a particular execution zone, only a fraction of these sessions can be achieved. Is this caused by the execution zone provider that is oversubscribing its resources too aggressively, is this accidentally due to an extreme interference between two random applications deployed on the same physical infrastructure, or is this because the application has an inaccurate view of the effectively required resources? If it is the responsibility of the execution zone to allocate enough resources for a particular application, then what prevents the application from lying about its internal health status in order to get more resources allocated to itself at the same cost?
- What entities are responsible for triggering such updates implicitly or explicitly? Does the service instance need to call a FUSION API function to inform the execution zone of a decreasing or increasing amount of available session slots, or can this be managed semi-automatically by an external load balancer that estimates the available session slots for a particular service instance based on monitoring information?
- How frequently does the available session slot information need to be updated in the overlay network, and what components are responsible for this? As there may be many service instances of many service types scattered across many execution zones and domains, there needs to be a good balance between accuracy and flooding. This needs to be studied, and may result into some interesting (potentially self-adapting) algorithms.

Copyright © FUSION Consortium, January 2014

- How to cope with many small service instances that typically only have a few available session slots available and that are distributed across zones and domains? How to prevent service routers to have to keep too much state, which is probably outdated anyhow?
- How to aggregate these available session slot counts in the overlay network without causing routing loops or double-counting? Perhaps it should also contain logical geo-information, or there needs to be a mechanism for uniquely identifying session slots without causing too much overhead.
- In general, what extra information is required to be distributed and/or stored inside the service routers, next to the available session slots per service type? This includes information for providing stability, preventing double counting and routing loops, service type parameterization, etc.

Most of these issues are no fundamental challenges, but often more related to implementation details requiring additional practical experiences and feedback from prototype implementations of FUSION features and the demonstrators. Therefore these issues will be addressed along the way when implementing FUSION and the demonstrators.

4.4 Inter-service communication and late binding

Due to the distributed nature of services, communication is inherently present between these services. Depending on the hosting platform and environment and its available communication interfaces, several interconnect options may become available.

In the simplest case the communication channel is chosen at design time (e.g. a TCP socket). Any other possible communication paths are not considered due to absence of knowledge about available communication paths. Therefore, the design time chosen communication channel can be sub-optimal compared to possible communication channels available on the hosting platform. The presence of specific adapter related technologies such as RDMA over Converged Ethernet (RoCE), Infiniband extend the portfolio of available possibilities. Choice of an optimal communication channel at deployment time can leverage an important benefit.

Due to the nature of FUSION with distributed execution zones and execution domains, two communicating services may have been deployed on the same (physical or virtual) machine or another, either inside the same execution zone or across multiple execution zones.

Moreover, in case of physically distributed services, it may be necessary to add one or more transformation functions, for example to save bandwidth (H.264 encoding) or to increase overall security (SSL encryption) at the expense of increased compute resources and additional end-to-end latency but on an overall increased efficiency and capacity.

Even at the application level itself, more optimal communication channels could be provided. One example is the efficient sharing of GPU buffer pointers across multiple service instances running on the same machine to avoid expensive copying and wasting huge amounts of bandwidth. As this is a very dedicated protocol, it is impossible for FUSION to support this out-of-the-box. However, FUSION should make it easy for services to detect these scenarios and deploy their own communication mechanisms via custom libraries in a reliable and secure way.

As a result, FUSION should support a late binding mechanism for services, to enable to use an optimized communication channel, based on the chosen service placement distribution and physical mapping. FUSION should allow services to implement their own communication channels, to be able to support custom and dedicated communication mechanisms.

Late binding from orchestration point of view can be realized from different angles, of which we describe two possible options:

Copyright © FUSION Consortium, January 2014

- Have orchestration perform the late-binding for the application so that at design time communication channels are available and try to optimize application placement using this knowledge;
- Allow an application or some intermediate layer to learn about available communication channels when an application is deployed and select the communication channel at placement/deploy time, for example using the evaluator services in combination with appropriate FUSION APIs.

From the above, selecting the 'optimal' communication path depends on two factors:

- placement algorithms
- knowledge of physical topology onto which the interconnected distributed service is deployed, together with the communication mechanism that is used.

In the second year of the project, we will work out the details, possible strategies as well as the mechanics in more detail, and define the benefits and constraints with respect to lately bound communication protocols in between related services. We will also implement several of these strategies within the scope of the demonstrator setups.

4.5 Service management interface

In this section, we discuss the initial key FUSION external interfaces for managing the services as well as internal interfaces for FUSION services to efficiently communicate and interact with other FUSION components. These interfaces can be divided into four key interfaces, as depicted in Figure 25:

- A lifecycle management interface
- An inter-service interface
- A data centre and resource management interface
- A FUSION routing and networking interface

Copyright © FUSION Consortium, January 2014

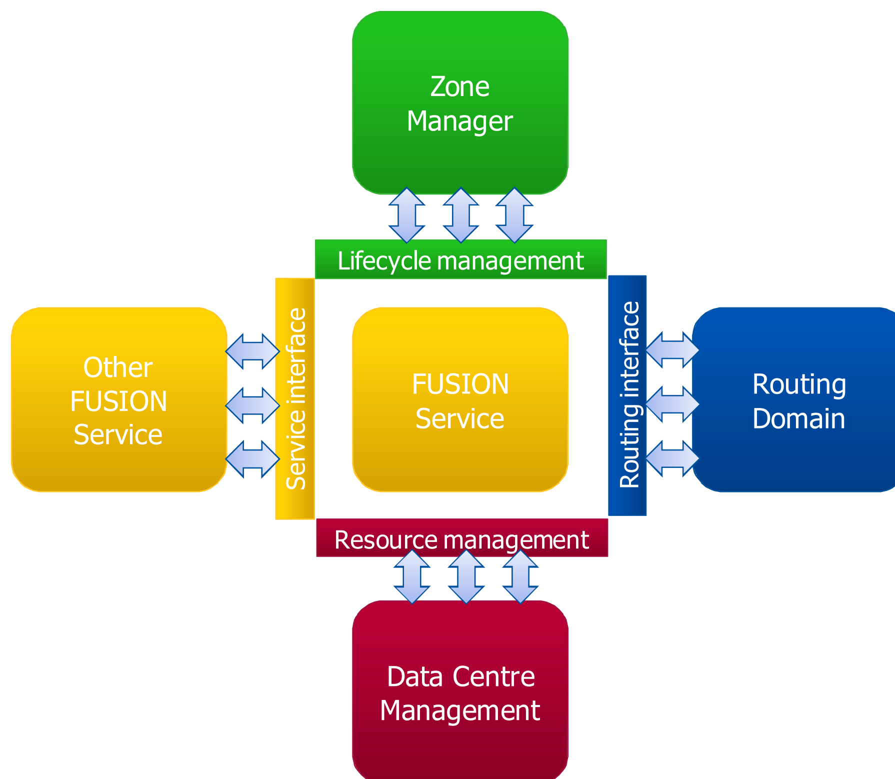

Figure 25: Service management interfaces.

Note that in FUSION, we envision that services can also be FUSION-agnostic, meaning that services do not have to be aware that they are being managed by FUSION. This enables legacy services to be deployed into FUSION. This can be accomplished for example by creating simple FUSION-aware wrappers that convert FUSION agnostic services into FUSION services. In the following sections, we will discuss each of these interfaces in more detail. Note that these (and the next) interfaces represent an initial set of envisioned interfaces. During the project, we will update and validate them as the design is being refined and based on input of the demonstrators.

4.5.1 Lifecycle management interface

This interface contains an initial set of key functions to control the lifecycle and overall health of a FUSION service, including functions to start and stop a FUSION service and for requesting overall monitoring information.

Copyright © FUSION Consortium, January 2014

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONServiceStart (zone manager → FUSION service)		
BEHAVIOR		
This function triggers the FUSION service to start running and start accepting service requests. This function could be implemented by a script that will start a (set of) application(s) with specific parameters.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceParameters	Blob	Instantiation parameters for the service instance
MaxSessions	Int	Number of session slots to prepare
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Returns whether the request was successful or not, or is pending

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONServiceStop (zone manager → FUSION service)		
BEHAVIOR		
This function triggers the FUSION service to stop all sessions and shut down.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Timeout	Time	Maximum amount of time stopping the service sessions
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Returns the status for stopping the service

Copyright © FUSION Consortium, January 2014

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONServiceGetAvailableSessions (zone manager → FUSION service)		
BEHAVIOR		
This function requests the current number of available session slots.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
RETURN VALUES		
Name	**Type**	**Description**
AvailableSessions	Int	The currently available number of session slots

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONServiceModifySessions (zone manager → FUSION service)		
BEHAVIOR		
This function requests the service instance to change the number of available sessions.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
MaxSessions	Int	The new maximum number of session slots
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Returns the status code

Copyright © FUSION Consortium, January 2014

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONServiceGetInformation (zone manager → FUSION service)		
BEHAVIOR		
Request the service instance to return health or monitoring information for a specific subset of categories. This could be service-specific monitoring information, or more general service information.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Category	Enum	Specific subset of information
RETURN VALUES		
Name	**Type**	**Description**
Information	Structured	The requested information, in a particular format.

4.5.2 Inter-service interface

Due to the communication intensive nature that many FUSION services will have, we plan to support optimized communication protocols using late binding for setting up low-overhead and/or high-bandwidth links between FUSION services. It is up to the execution zones to decide what late binding protocols they will support and provide to the services. It is important to note that these functions assume that the services involved in the inter-service communication have already been located, for example using the FUSION routing plane, but want to enable more efficient data-plane communication in between these instances.

Copyright © FUSION Consortium, January 2014

Inter-Service Interface		
FUNCTION NAME		
FUSIONServiceBind (FUSION service → FUSION service)		
BEHAVIOR		
This is the key function for doing the late binding in between different services		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	URI	Name of the other service to connect
BindType	Enum	What communication protocol to use for binding to that service (could be automatic)
BindParams	Structured	More details concerning the type of communication
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code for the request
Socket	Socket	An optimized FUSION socket connection, implementing an advanced communication protocol (e.g., shared memory across VMs)

4.5.3 Resource management interface

Because of the compute-intensive nature of FUSION services, these services may require specialized hardware. Via the interfaces below, FUSION services or evaluator services can query and request for (the availability of) specific hardware resources. As with the late binding protocol, it is up to the execution zone whether they want to support this interface or not.

Copyright © FUSION Consortium, January 2014

Resource Management Interface		
FUNCTION NAME		
FUSIONServiceQueryResources (FUSION service → zone manager DC mgmt)		
BEHAVIOR		
Query the execution environment for the existence of specific hardware resources.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	URI	Name of the service
DeployParameters	Blob	Deployment parameters for the service
Timeout	Time	Maximum amount of time for making the evaluation
RETURN VALUES		
Name	**Type**	**Description**
Score	KeyValueList	The multidimensional score

Resource Management Interface		
FUNCTION NAME		
FUSIONServiceRequestResource (FUSION service → zone manager DC mgmt)		
BEHAVIOR		
Requests a specific resource or accelerator to be allocated and bound (dynamically) to the service.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ResourceType	String	Name of the resource
ResourceParams	Complex	More details about how much resources are requested
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
ResourceHandle	Handle	A handle to the resource

4.5.4 Routing and networking interface

The routing and networking interface covers incoming service requests and data-plane communications from users invoking the service and, in the case of composite services distributed over multiple execution zones, either incoming or outgoing service requests and data plane communications from/to service instances in remote execution zones. This interface and the associated protocols are part of the service networking work specified in deliverable D4.1.

Copyright © FUSION Consortium, January 2014

5. DISTRIBUTED SERVICE ORCHESTRATION

In this section, we will discuss in detail the key FUSION domain-level orchestration functions for managing service instances across a distributed set of execution zones.

5.1 Functional design

A FUSION domain is managed by an orchestration layer comprising several key functions for managing both the services as well as the execution resources. This section will provide details on the overall design decisions as well as the key functional blocks for implementing these functions. In this discussion, we will assume a single logical orchestrator per domain, although obviously, for scalability reasons, each function could be implemented in a distributed way by multiple entities.

5.1.1 Overall design decisions

For the design of the FUSION domain-level orchestrator, we envision the orchestrator to consist of a modular set of functional components, each of which allowed to be a (FUSION) service on its own. Each orchestration function is a service that may have multiple instances that can be deployed on one or more execution zones within the orchestration domain (though this is not a key requirement, see section 5.3 on lifecycle management of the FUSION orchestration functions).

For the communication protocol in between the various orchestration services, we will use a set of RESTful APIs. In the future, we may even envision an OpenStack compatible API on top of a FUSION orchestration zone for managing and deploying services in FUSION, including necessary hooks and support for efficient deployment of FUSION services within a heterogeneous data centre.

5.1.2 High-level design

Figure 26 depicts a high-level view of the key FUSION domain orchestration functions and their key interactions. For clarity, we did not draw any arrows between the various functional blocks and the security block. The dotted lines represent public interfaces, whereas the solid lines represent internal interfaces in between various orchestration services. In this section, we give a brief overview of the key functions and their key role in the overall orchestration. Each function will then be discussed in more detail in separate sections.

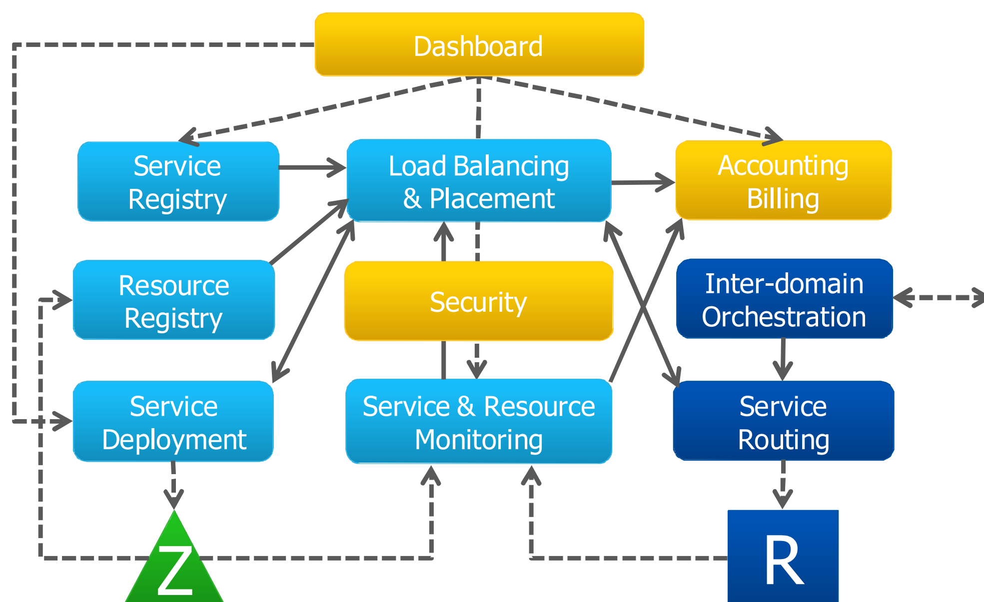

Figure 26: High-level design of a FUSION domain orchestrator

Copyright © FUSION Consortium, January 2014

- Dashboard

  At the top, we envision a FUSION dashboard web service acting as a graphical frontend for managing a FUSION domain, registering and deploying FUSION services, billing, etc. This dashboard internally interacts with the various orchestration functions via their respective APIs.

- Security

  This component is responsible for all authentication and authorization functionality whenever a particular orchestration API is used. Whenever an unauthorized request is made to one of the orchestration components, they should be automatically forwarded towards the security component, which will enforce service authentication, and which will ultimately also decide whether a particular user of a component is authorized to make this request. This component will also contain all domain-level policies.

- Accounting & billing

  The FUSION orchestrator needs to ensure that all entities are being billed and paid appropriately according to provided or consumed resources and offered services.

- Service registration

  A key orchestration function is the service registration function, which will handle all registrations of new service types, updates with respect to their deployment parameters, and ultimately decommissioning of a particular service in the FUSION domain. Registering a new service may involve a subsequent automatic deployment of a number of instances within the orchestration domain, as described in the service manifest.

- Resource registration

  Execution zones, service routers and other resources also need be registered to or unregistered from the domain orchestrator. This enables the orchestration domain to grow or shrink dynamically, and also enables execution zones or routing domains from external parties to be attached to a particular orchestration domain. A domain orchestrator may for example also decide to create temporary execution zones on remote locations (e.g., on Amazon EC2) and attach them to its orchestration domain to handle unexpected peak loads.

- Service deployment

  Another key function is the service deployment function. This function is responsible for interacting with the zone manager of an execution zone for deploying new instances in that execution zone. This involves executing and coordinating the 4-step placement strategy discussed in Section 5.4.2.1. Service instances can also be deployed directly by authorized entities, possibly on targeted locations. The load balancer may also directly instruct the service deployment function to start deploying new instances in particular zones based on (changing) predicted demands, prices, etc.

- Load balancing and placement

  The load balancer and placement function has multiple purposes. First, it is responsible for guaranteeing proper QoS by making sure the load of all service instances is on par with what was requested by the service provider; for example through an SLA. This includes taking into account load forecasting or load patterns provided by a service provider. Next, the placement function contains the inter-zone placement algorithms for optimally deploying all service instances across all available execution zones, taking into account the announced available resources from each execution zone, and the existing and predicted execution requirements from all current and future service instances deployed within the orchestration domain.

- Domain monitoring

Copyright © FUSION Consortium, January 2014

The monitoring block is responsible for aggregating and analyzing all available monitoring information coming from the execution zones, the service instances, the service routers and the network. This component provides essential input towards many other orchestration functions, like the load balancer and placement component, the accounting and billing component for processing the resource utilization per service instance and per execution zone. It also provides health and status information towards the domain orchestrator and the service providers regarding the overall operation of the domain and the individual services.

- Inter-domain orchestration

  To handle inter-orchestration communication and deployments, we envision a dedicated component that will interface with the other orchestration domains and that will make sure that all security issues and other requirements and agreements are met. Most likely, only limited information regarding service instances and resource availability will be shared with the peer orchestration domains. In case one orchestration domain exposes itself as an execution zone towards another orchestration domain, this block (or an extension) could also implement all necessary interfaces for an execution zone.

- Service routing

  In case the domain orchestrator controls and manages the underlying routing domain, this component is responsible for programming and managing all service routers, configure their FIBs, etc.

5.2 Key FUSION orchestration actors

Many different entities will need to interact with different aspects and functions of a FUSION orchestration domain. Each entity will typically only have access towards a subset of all functions and data available in that domain. Below an overview of some of the key entities and roles we envision that need direct access to particular FUSION orchestration functions. Note that these entities can involve either physical persons as well as software components.

- FUSION orchestration domain admin

  This entity is responsible for managing an orchestration domain. Its key role is to ensure proper operation of the entire orchestration domain. This means ensuring the orchestration services are up and running, doing overall management, accounting and billing, ensuring the necessary security policies are in place and are operational, and ensuring proper provisioning of resources (e.g., execution zones, service routers, network infrastructure, etc.) with respect to the deployed services to ensure all services can run properly in a healthy stable environment.

- FUSION execution zone admin

  This entity is responsible for managing an execution zone. This involves registering the available resources towards the orchestration domain, ensuring proper communication between the orchestration domain and the execution zone, enforcing all necessary policies dictated by the orchestration domain, as well as installing all appropriate policies of the execution zone with respect to the orchestration domain, etc.

- FUSION network domain admin

  This entity is responsible for managing the FUSION overlay network for routing FUSION service requests. The FUSION orchestration domain at the very least needs to be aware of some network health information coming from the network domain in order to appropriately balance and deploy services across the orchestration domain. In case the orchestration domain is also responsible for managing several aspects of the FUSION routing domain, this entity is also

Copyright © FUSION Consortium, January 2014

responsible for ensuring proper communication and execution of all requests coming from the orchestration domain.

- FUSION service provider

  This entity is responsible for registering and managing new service types within a FUSION domain. Some of its other key functions are to monitor the overall operation of its services within an orchestration domain. This includes monitoring the overall health of the services, the billing, as well as the set of policies that were enforced for each individual service.

- Zone manager

  The zone manager is the software component that will actually manage all communications between an execution zone and the FUSION domain orchestration. This includes service deployment, service monitoring, etc. These functions are described in detail later.

- Service router

  The service routers should forward relevant information towards the orchestration domain so that the latter can properly deploy and balance FUSION services across an execution domain. In case the orchestration domain also controls the networking domain, the orchestration domain needs to be able to manage the forwarding tables and/or service routing policies within these service routers.

5.3 Lifecycle management of the FUSION orchestration services

All these FUSION domain-level orchestration services also need to be hosted and managed somewhere in the network. We are currently assuming and exploring two main operational models. In the first model, the FUSION orchestration services are deployed on a non-FUSION infrastructure, which could be located inside or outside the geographical area that spans the FUSION orchestration domain. In this model, the orchestration services are managed externally and need to be made compliant with the underlying service management platform, which could have its own dedicated APIs, or which could use standard APIs (e.g., EC2, OpenStack API). In this case, there is a clear distinction between FUSION orchestration services and FUSION application services.

In the second model, all FUSION orchestration services are actually FUSION-compatible services themselves, which are deployed on FUSION execution zones and managed by the zone managers. This approach has a number of advantages. First, it results in a very symmetric model, where FUSION orchestration services are treated the same way as regular application services. This also means that the requirements for the orchestration services can be expressed in the same language as regular FUSION services. Second, as these services are running in one or more execution zones, the same physical infrastructure can be reused and managed via one common interface (i.e., the zone manager). One potential issue however is that it may complicate the order in which to bootstrap a FUSION orchestration domain and its initial execution zones: there needs to be at least one execution zone already up-and-running, on which an orchestration domain is deployed. This could be handled by incorporating a special function inside the zone manager to automatically deploy particular orchestration services inside its execution zone and automatically registering itself to that orchestration domain.

5.4 Key functions

In this section, we provide a detailed view of each key orchestration function.

5.4.1 Service registration

Any new service first needs to be registered to a FUSION domain before it can be deployed and instantiated in that FUSION domain. This can be either an application service or a FUSION core

Copyright © FUSION Consortium, January 2014

service. In case of application services, the service registration is typically triggered by the application provider that wants to deploy and start instances of the new service in that FUSION domain.

The service registration is handled by a FUSION domain registrar service, and involves sending (an URL of) a service manifest to the registrar. A link to the required software packages can be provided in the manifest. These software packages can be stored by FUSION in a repository. The registrar can accept or decline the registration. If the registration succeeds, a unique service identifier is generated and returned to the service provider, which can use this service identifier to refer to that newly created unique service type.

A possible sequence diagram is depicted in Figure 27. In this diagram, a service provider registers a new service, which is stored in a repository. This repository may as part of the registration start fetching the software packages and other information. Upon success, the service provider is returned a unique service identifier, which the service provider can then use when identifying the service when making a service request. In parallel, the domain orchestration may start analyzing the manifest and start pre-deploying instances across execution zones in the FUSION orchestration domain.

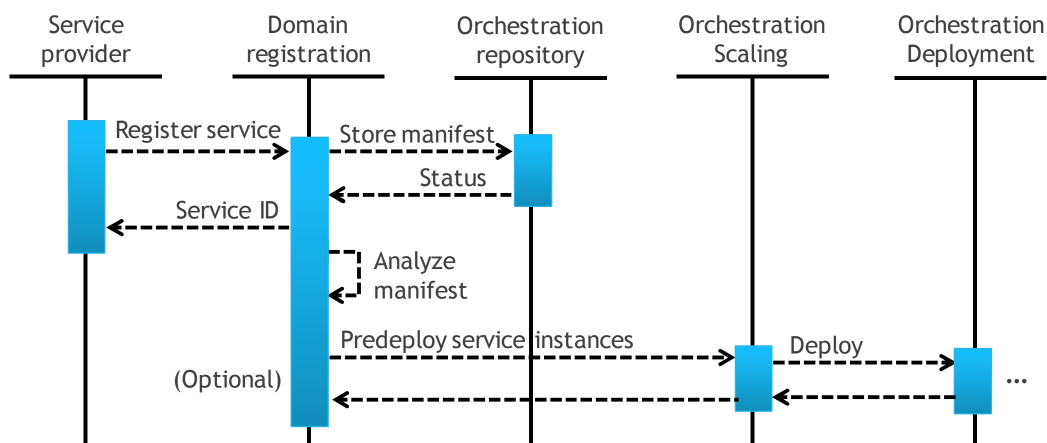

Figure 27: Sequence diagram for service registration and optional initial predeployment

The registrar can at any times be queried to search for particular service types, in which case all service types you are authorised to query for will be returned. The querying could be done based on name, keywords, functionality, etc., and could be provided by a dedicated (optional) FUSION core service.

5.4.2 Service placement

Service placement or service mapping is the act of selecting Execution Zones that will advertise service availability to the service routers. Service placement is closely related with resource monitoring and service deployment, as services should only be mapped onto execution zones or hosts with enough available and appropriate resources (see also Section 5.4.2.3 on execution zone resource allocation). The input for the service placement operation consists of a number of constraints coming from multiple sources, including the service manifest, zone resource availability, cost constraints with zone providers, current and forecasted user demand, etc. The output of the operation is the set of execution zones that subsequently will advertise the service through the service routing function.

In some scenarios, extremely service-specific requirements must be taken into account during service placement. To keep the FUSION orchestration layer scalable, these specific requirements must be abstracted to the orchestration layer and the involved service must aid in the placement process. Notable examples are GPU features like shader models and available GPU memory. On the

Copyright © FUSION Consortium, January 2014

one hand, these parameters are too detailed for evaluation at the orchestration layer. On the other hand, it is unlikely that zones will advertise this to the orchestration layer. Hence, we need to deploy evaluator services to support the service placement.

Therefore we plan two different roles working together via the FUSION API:

- FUSION has the master control over evaluating different possible locations.
- Service-specific service evaluators help FUSION by scoring possible locations based on service-specific criteria.

A software requesting a service instance from FUSION should be able to pass service-specific description parameters, which are not understood by FUSION, but handed over to corresponding service evaluators that understand the description. FUSION only understands the scoring returned by the evaluators and uses it to determine the best place for the new service instance.

5.4.2.1 Four-step service placement

In this document, we propose a four-step divide-and-conquer approach for efficiently handling the service placement operation at the orchestration layer in an efficient, scalable and feasible manner. The sequence diagram is shown in Figure 28.

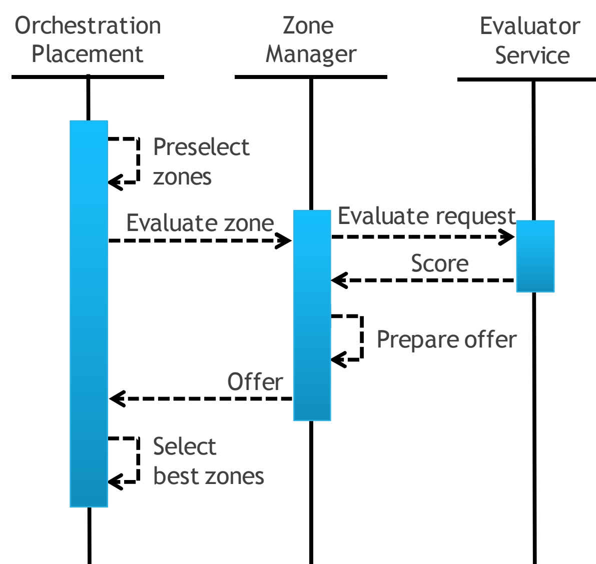

Figure 28: Sequence diagram for the four-step placement algorithm

1) The orchestration domain first selects a number of feasible execution zones based on high-level attributes and requirements (distance, policy, application and zone profile, availability, etc.).
2) The orchestration domain then forwards the service mapping request to these selected zones in parallel, and requests to evaluate whether it can provision that service and at what cost. In other words, the selected zones analyse the request and make an offer to the orchestration service. That offer may even be acting as a binding contract between a zone and a domain, and once a particular zone is selected, it has to make sure it can deliver the QoS as described in that offer. Note that the orchestrator may request a zone to give an answer within a particular time frame, for example to have fast responses in the on-demand service deployment scenario.
3) The selected execution zones each analyse the deployment or instantiation request and provide an offer within the allocated time frame. Each zone manager may trigger a generic or service-specific evaluator service for doing the analysis and for providing a multi-dimensional score, which is then returned towards the orchestration domain.

Copyright © FUSION Consortium, January 2014

4) In the last step, the orchestration receives all the offers and decides on the optimal execution zone, based on the offers from each zone. The offer consists of a set of scores on how effective and costly it would be to create a new instance in the corresponding execution zone. The domain-level orchestrator service can then make a decision on what execution zone fits the needs of the client best, based on priority metrics provided by either the manifest or client (e.g., lowest cost, or lowest latency within a particular budget, etc.) as well as FUSION orchestration domain policies.

There are a number of advantages to this approach. First, the zones do not have to expose their internal resources (a similar argument can be made for the inter-domain scenario). Second, the complexity at the domain-level is lowered and the overall scalability is increased, as the service placement operation now becomes a more local optimization process with only a global decision at the domain level. Third, the offer as described above could be used as a binding contract, and fourth, this approach makes it easy for zones to attract or repel new services (or types), and perhaps could result in competition between multiple zones, resulting in lower costs for the service providers in case of independent execution zones managed by different providers.

5.4.2.2 Evaluator services

This section discusses the concept and implementation of evaluator services as an enabler for deploying complex services with specific requirements onto distributed heterogeneous execution environments, focussing on the scoring aspect when evaluating possible locations.

5.4.2.2.1 Scoring of possible locations

Scoring may take into account for example:

- Requirement for special hardware, e.g. GPUs with specific shader models.
- Load and current QoS, for example network, CPU or GPU loads relevant for the specific service.
- Custom-weighting of network connection quality parameters (delay, jitter, packet loss etc) to other services. For example, video may tolerate more packet loss and larger latency than game servers.
- Possible locations of other related services that also need to be instantiated. For example, the description may contain a list of the locations of game players, plus a flag for each player whether the player only has a thin client application, requiring a client to be instantiated in the cloud, which should be taken into account when choosing the location for the server. See further for more information about related services.
- If the service needs to access another already existing service instance, it may ask specific aspects about that service instance, e.g. load, distance, or service quality parameters (e.g. supported video resolutions).

FUSION understands the scoring returned by the evaluators. In the simplest case, the score is a single float value containing the overall score, meaning that a higher score value represents a better possible location. However, the score value may be more fine-grained and could be represented by a dictionary to translate standard score component name strings into score values. For deciding about the kinds of score components, there are two opposite, possibly contradicting approaches:

- The evaluator may combine multiple internal score values (e.g. hardware capacity, network distance to other services, current load) into one single score value and return that single score value to FUSION.
    - Advantage: The combination of multiple aspects into a single score can be service dependent. For example, one service may be more interested into good hardware and less

into networking bandwidth and response time, while another service may care very much about low network latency.

- Disadvantage: FUSION cannot manage the individual scoring details.

- The evaluator provides the individual scores (like separate scores for hardware capacity, network distances to other services, current load) to FUSION, so that FUSION can use different policies for using these values.
  - Advantage: FUSION gets more fine-grained information useful for orchestration decisions and can implement its own method/algorithm..
  - Disadvantage: FUSION may have not the knowledge about the preference of different services, or how to appropriately scale and aggregate the various scores, each of which may have completely different dimensions.

In practice a mixture of both approaches may be possible. For example, a service may return always an overall score value, which is used by FUSION in the simplest case. Additionally, services may return additional values for more refined information which may be optionally used by FUSION.

We suggest the following approach: evaluators should combine scores into one single or only very limited scores as much as possible before reporting them to FUSION. FUSION should only handle different kinds of scores if necessary.

Standard score component names may be for example:

- "total" into float: a total score which takes all aspects into account the service cares about, for example hardware, network distance to other services and current load. 0.0 means that the location is not suited at all and 1.0 means the location is perfectly suited.
- "hardware" into float: suitability of the hardware, where 0.0 means that the hardware does not support that service, and 1.0 means the hardware perfectly support that service.
- "capacity" into float: available CPU and GPU capacity, for example 0.0 means no available capacity and 1.0 means full available capacity.

This scoring is implemented in an evaluator service and not only in components running on the execution points itself for the following reasons:

- One evaluator service instance may calculate scores for a large number of execution points. For example one evaluator answering for many cloud servers instead of needing a evaluator running on each execution point.
- Caching of scoring answers.
- Evaluation mechanisms not specific for execution points, but for groups of execution points.

5.4.2.2.2 Scoring locations of related services

As result of the discussion on related services above, in case of related services the scoring of possible locations for one service may depend on the location of related services. This means that an evaluator service may also inform about possible locations of related services.

A possible approach – but with bad scaling behaviour without further measures – could be the following. A service evaluator may recursively ask FUSION about scoring locations of other related services. Formally the service instance request scores only the possible instance locations for the single requested service. For example, in the diagram below, request (1) asks only for an instance of service A, which is for example a game server. However, to satisfy this request, FUSION effectively scores possible locations for complete service instantiations, including not only possible locations for instances of service A, but also for instances of services B and C, which happen to be two game clients in this example. This total score is then only used in this first step to instantiate the service A

Copyright © FUSION Consortium, January 2014

at the determined best location. Then, if the user application requests the related service instances B and C at steps (13) and (17), FUSION can determine the best location for these instances based on the location of the already instantiated service instance A.

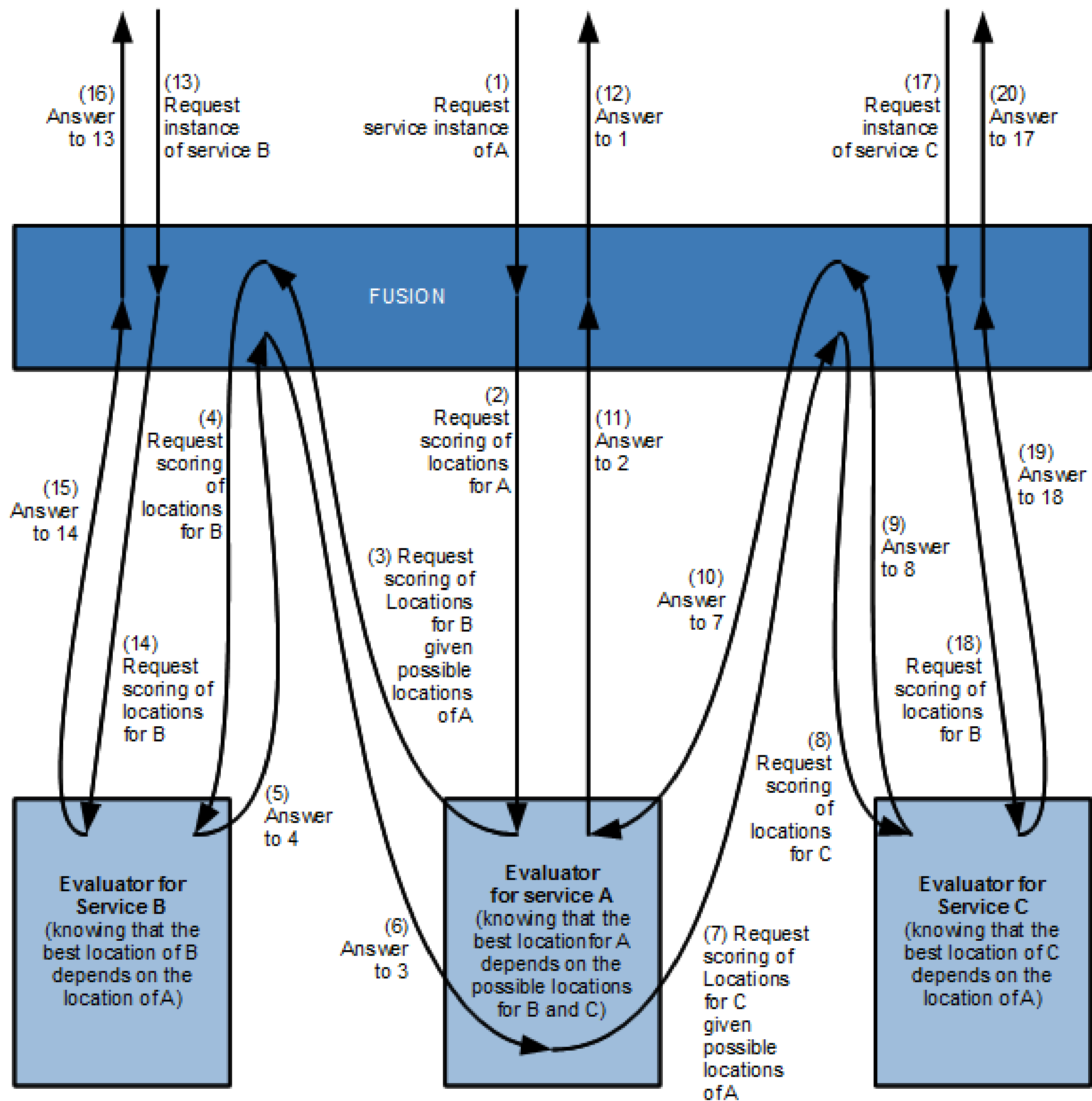

Figure 29: sequence flow of a composite service evaluation process.

The time diagrams for this approach is the following:

Copyright © FUSION Consortium, January 2014

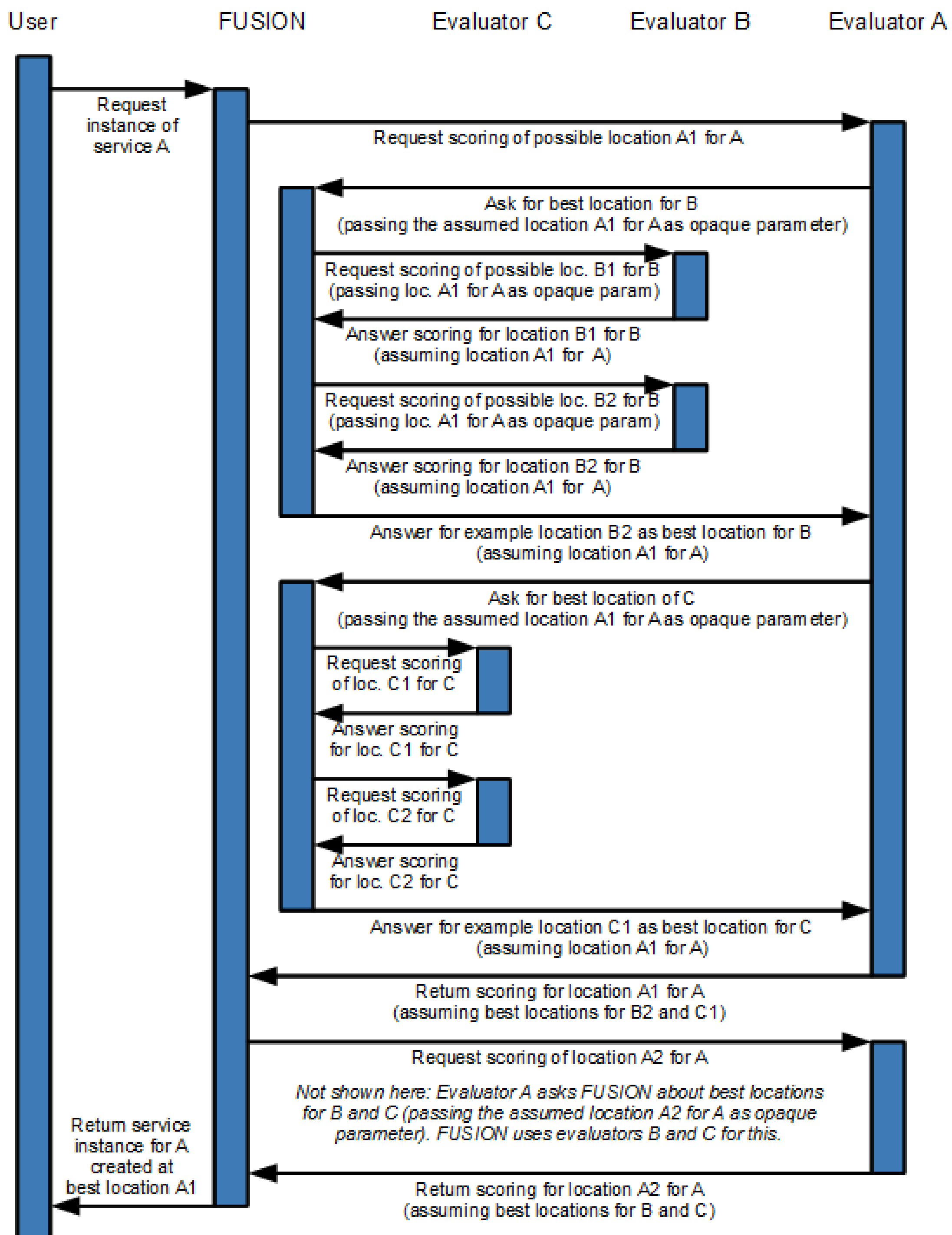

Figure 30: Detailed sequence diagram for complex service scoring and evaluation of service A

Copyright © FUSION Consortium, January 2014

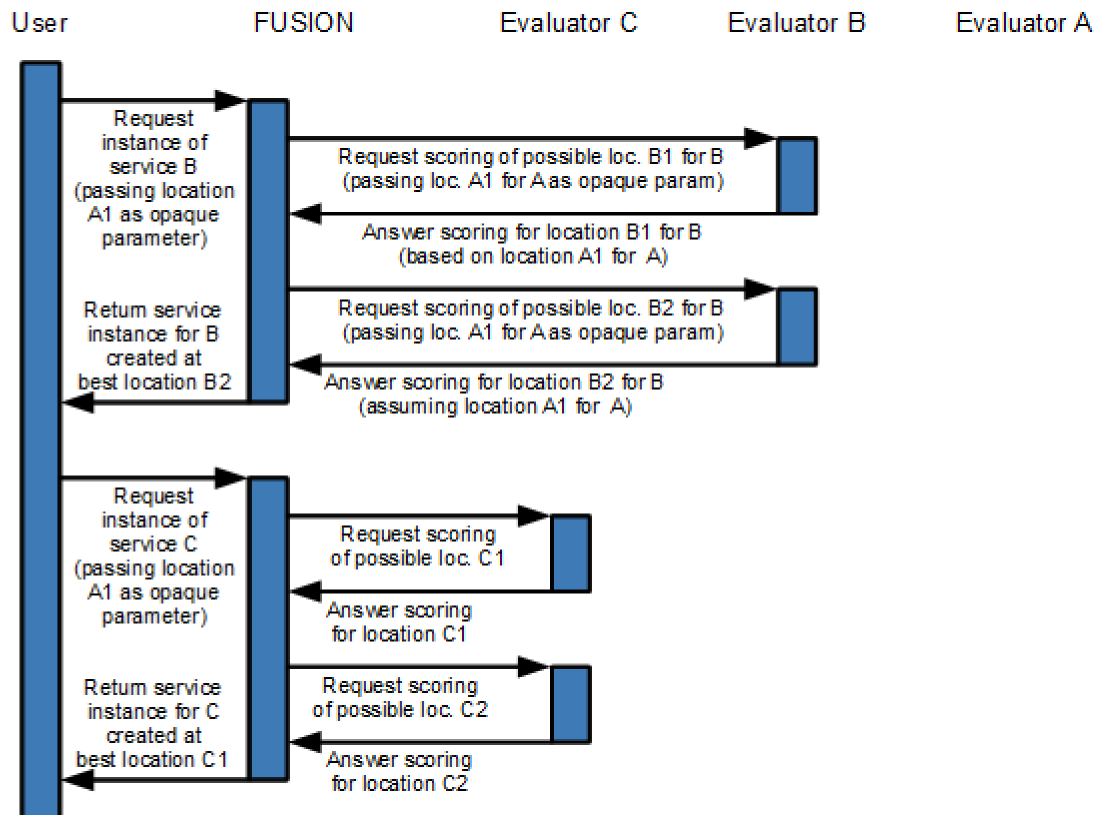

Figure 31: Detailed sequence diagram for complex service scoring and evaluation of service B and C

This approach satisfies both needs discussed above:

- Related services can be placed optimally taking the whole picture into account
- For each service, FUSION can use (extensible) service specific knowledge to rank locations for service placement, including ranking for placement of relative services.

5.4.2.2.3 Optimizing scoring

In the mechanism above, FUSION may calculate the same or similar scorings multiple times. For example, the request (14) in the diagrams above may be similar to one of the requests (4). Therefore the request (14) may use the results of requests (4), possibly avoiding the requests (14), basically caching the scoring results. However, this may be difficult to implement, because FUSION does not understand the request description, and therefore may not be able to easily identify equal or similar requests. A simple approach may be just a byte-wise comparison of the description parameter. Alternatively, the users and services may provide FUSION an additional identification key in addition to the description parameter, which FUSION can use to identify equivalent requests. However, this additional complexity may not be worthwhile, for example because request (1) anyway requires much more requests than (3), because at (1) the server location is not known yet. Furthermore, evaluators may cache scoring, see above.

Another aspect is that the above algorithm would scale exponentially with the number of related services, because the algorithm effectively scores all possible combination of locations of the services. This is of course not acceptable in practice. Therefore smarter approaches are required. possible approaches may be:

Copyright © FUSION Consortium, January 2014

- FUSION may use some heuristics for placing the service components, and so can rule out many placement combinations. For example, FUSION may assume to place all related service components in the same zone and not score all possible combinations of the related services placed in different zones.
- FUSION knows that many EPs result in exactly the same scoring. For example, all EPs of one data centre may be equivalent, so the evaluator must be called only once for one of them, and then FUSION can choose the location efficiently based on additional other data, for example the current load of individual EPs.

5.4.2.3 Execution zone resource allocation

The task of reserving and allocating resources within execution zones for service instances can be seen from two perspectives. Firstly there is the perspective of the orchestrator who must select execution zones that 1) are in appropriate locations to meet the network aspects (latency, throughput, etc.) of service performance constraints as defined in the service manifest, given the predicted demand in terms of the geographical distribution of users and the volume of service requests from those locations, and 2) have the hardware capabilities required by the service instances, including specialized resources such as GPUs supporting specific shading algorithms, for example. These aspects are discussed in the preceding section on evaluator services and will be refined further in the specific server placement algorithmic work to be documented in deliverable D3.2.

The second perspective of resource allocation is from the point of view of the execution zone. Whether they are large or small the available computational resources in a data centre are finite. Requests for resources may exceed capacity, especially for smaller execution zones. This is the case whether we consider multiple services being deployed by a single orchestrator within an execution zone (case a) or the related case of multiple execution zones belonging to different orchestration domains running in the same data centre where the different orchestrators may compete for the data centre resources to be allocated to their execution zone (case b). An initial algorithm for this resource allocation problem for competing requests for finite execution zone resources is presented in Section 7.2 - where an auction-based mechanism is applied to services (case a) or orchestrators (case b) bidding for resources to run service instances.

As discussed in Section 2.7 FUSION services can either be pre-deployed in execution zones, prior to requests being issued by users, or deployed at invocation time. These two modes of operation require radically different response times for the allocation of resources as discussed above. Some degree of resource pre-allocation within execution zones may be required to reduce invocation latency for on-demand deployment. An example of this might be that a data-centre’s computational resources are reserved for an execution zone but not allocated to any specific service. These spare resources are made available to the orchestrator and could be used by any of the service instances the orchestrator manages to speed up on-demand deployment. The quantity of spare resources could then be managed by domain scaling algorithms as described in the second scenario in the following section.

5.4.3 Service scaling

Service scaling goes beyond service placement. Whereas service placement mainly involves the selection of what execution zones should host a service, service scaling refers to the number of instances that should actually be deployed in each of the selected zones.

Instance scaling algorithms typically balance resource usage with application performance. Upscaling the number of service instances will improve application-level performance metrics such as response delay, but incurs additional costs (energy, renting). Downscaling is needed to avoid unnecessary capacity costs when the service demand is low.

Copyright © FUSION Consortium, January 2014

While the decision authority for inter-zone scaling is the FUSION orchestration layer, there are different options for the intra-zone decision authority.

A first option is to leave the intra-zone scaling decision authority at the service itself. In this case, the intra-zone scaling mechanism is completely opaque to the FUSION orchestration layer. Instances could autonomously request duplication or shutdown by contacting the zone manager. Note that Domain Scaling (i.e., zone selection) could configure a limited set of properties, such as the maximum deployment cost, or a minimal number of instances. These property configurations could be provided by the service provider upon registration of his service with the FUSION orchestration layer. This opaque scenario is illustrated in Figure 32:

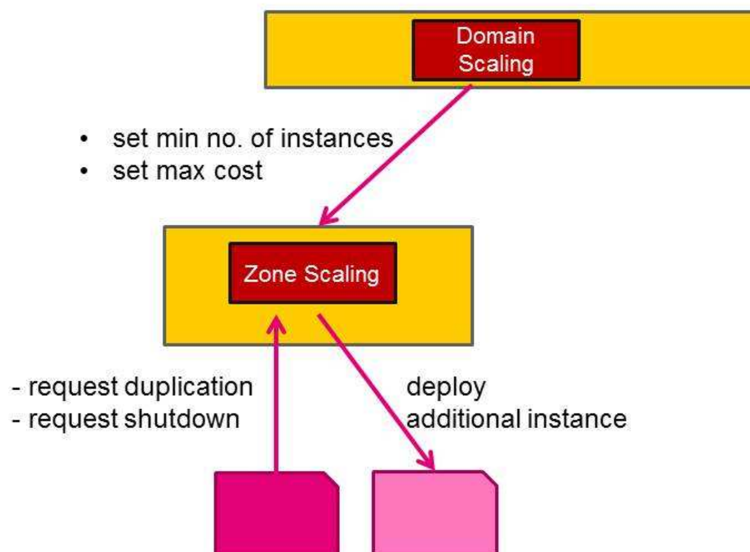

Figure 32: Opaque scaling: scaling decisions are taken by the service instance.

The advantage of this scenario is that very application-specific elasticity rules can be applied. Each service instance could implement its own elasticity, without exposing Key Performance Indicators to other components. The biggest drawback is that a standardized scaling interface to the zone scaling component is needed. This may be difficult to enforce in practice, as it mandates the use of standardized interfaces.

In the second scenario, the scaling authority is situated at the Zone Manager. Service instances report Key Performance Indicators to the Zone Manager. These values are evaluated against the elasticity rules configured by the FUSION orchestration layer and may result in up- or downscaling of the number of instances. If a particular service is overloaded and the zone manager cannot further upscale, the zone manager may send a notification to the Domain Scaling component situated in the FUSION orchestration layer. The zone-orchestrated scenario is illustrated in Figure 33.

Copyright © FUSION Consortium, January 2014

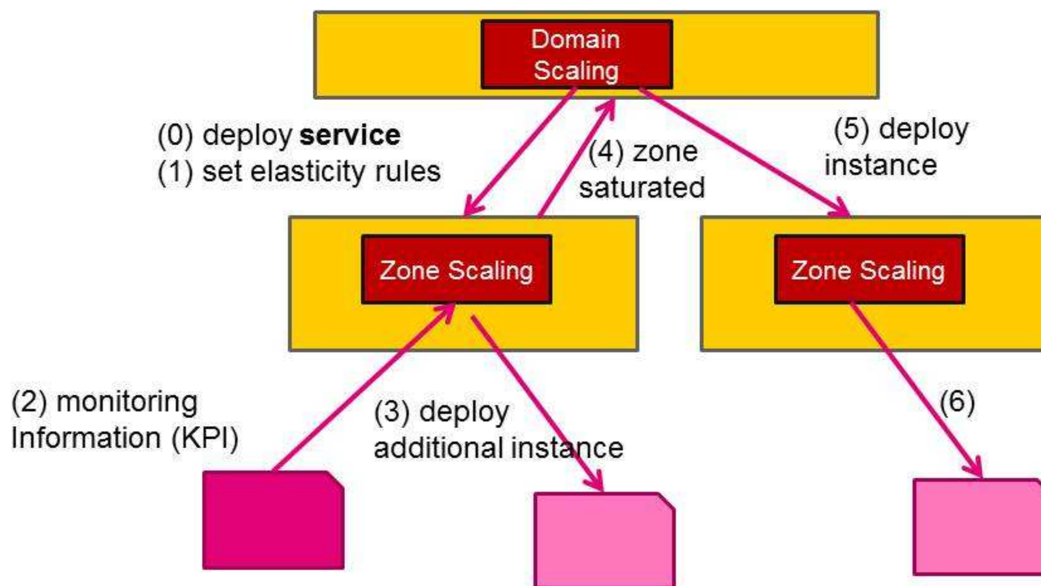

Figure 33: Decision authority situated at the Zone Scaling component of the Zone Manager.

In this scenario, the interface between service instances and management components of each zone could be very narrow: only KPIs must be reported. Furthermore, there is an interface needed for configuration of a basic set of elasticity rules.

This scenario can be further refined, according to the type, granularity and authority of the elasticity rules that are being configured by the orchestration layer. One case could be that the configured rules should only be seen as a guideline for the Zone Scaling component: the zone manager keeps full authority on scaling decisions.

In another case, the rules configured by the Domain Scaling component are entirely adopted and adhered to by the Zone Scaling mechanism. Service providers could register their elasticity rules with their service to the FUSION orchestration layer, which translates these to specific scaling rules per zone.

5.4.4 Service deployment

Given the fact that the service placement was already done, service deployment at the orchestration layer simply involves triggering the selected execution zone(s) to start deploying new service instances with particular parameters, enabling a particular amount of session slots for that service. A zone may accept or refuse the service deployment request. All details of how an execution zone will implement the deployment is completely hidden from the point of view of the orchestration domain. Information on the status of the deployment and the service instances will be returned to the orchestration domain.

In case of a distributed deployment of a composite service across multiple execution zones, the orchestration domain needs to coordinate the deployment of each service atom, and ensure that the individual service instance are able to connect appropriately to each other. This could be achieved for example by deploying the service atoms one at the time, and providing the locator for each service atom as a parameter for the connected service atoms during deployment. More flexible and dynamic schemes are also possible.

5.4.5 Service and resource monitoring

The service and resource monitoring described in this paragraph relate to key functionalities in distributed service orchestration.

Copyright © FUSION Consortium, January 2014

As a concept, monitoring for orchestration serves the purpose of learning the state of the execution layer so that orchestration can take into account the current execution layer state during service scaling and placement as well as serve as a reporting function towards the key actors that interact with a FUSION orchestration domain: the domain orchestrator for monitoring the health of its orchestration domain, service providers to receive feedback concerning service health, deployment and scaling information, billing based on resource and service consumption, etc.

The monitoring state can encompass the following topics (exemplary and not limiting to these):

- Data centre related metrics

  Depending on the execution zones exposing overall resource utilization and capacity information, an orchestration domain can leverage this information for improving service placement and service scaling, preselecting execution zones based on availability of particular resources, etc.

- Service execution related metrics

  This set of metrics comprises service execution and utilization metrics. This includes the overall number of service types, instances and available session slots in various execution zones. It may also include the resource utilization (e.g., CPU, memory, network or disk I/O, etc.) of specific service instances running on physical machines. The latter may be useful for service providers or automatic scaling components for measuring service load or triggering the scaling of service instances.

- Networking related metrics

  During service placement and deployment, a FUSION domain orchestrator should take into account available network information coming from the FUSION routing plane as well as the underlying IP infrastructure. Although we do not intend to use strict network resource allocation schemes (as not only FUSION-related traffic will be transported over the network infrastructure, and as FUSION has no direct control over the data plane), we do intend to take networking related metrics like latency, jitter and bandwidth into account when making the final placement and deployment decisions.

- Application related metrics

  These metrics consist of application-specific monitoring data coming from the application services, like for example the actual processing or rendering frame rate. These metrics are relevant for the service providers that may want to measure the overall QoE of the deployed service instances, as well as for application-specific scaling components, which may decide to trigger FUSION orchestration or routing based on these metrics to for example scale instances up or down, or to reduce the amount of service requests towards particular instances.

- FUSION platform related metrics

  These monitoring metrics relate to the state and health of key FUSION architectural components, including the zone managers controlling an execution zone, the service routers, etc.

Given the scale at which FUSION is intended for, an important aspect to consider is the amount of information that is available at lowest level and how this information is propagated and aggregated so that the amount of monitoring info is kept to a minimum towards the orchestration layer. Also for security or business reasons, execution zones and services may want to restrict the amount and type of information they want to propagate and publish via the FUSION monitoring interface. Some of the service monitoring information for example may be passed on to the FUSION monitoring function as a blob of data that can only be understood by the service provider.

5.4.6 Inter-domain orchestration

Figure 34 shows the scenarios of single and multi-domain orchestration considered in FUSION.

Copyright © FUSION Consortium, January 2014

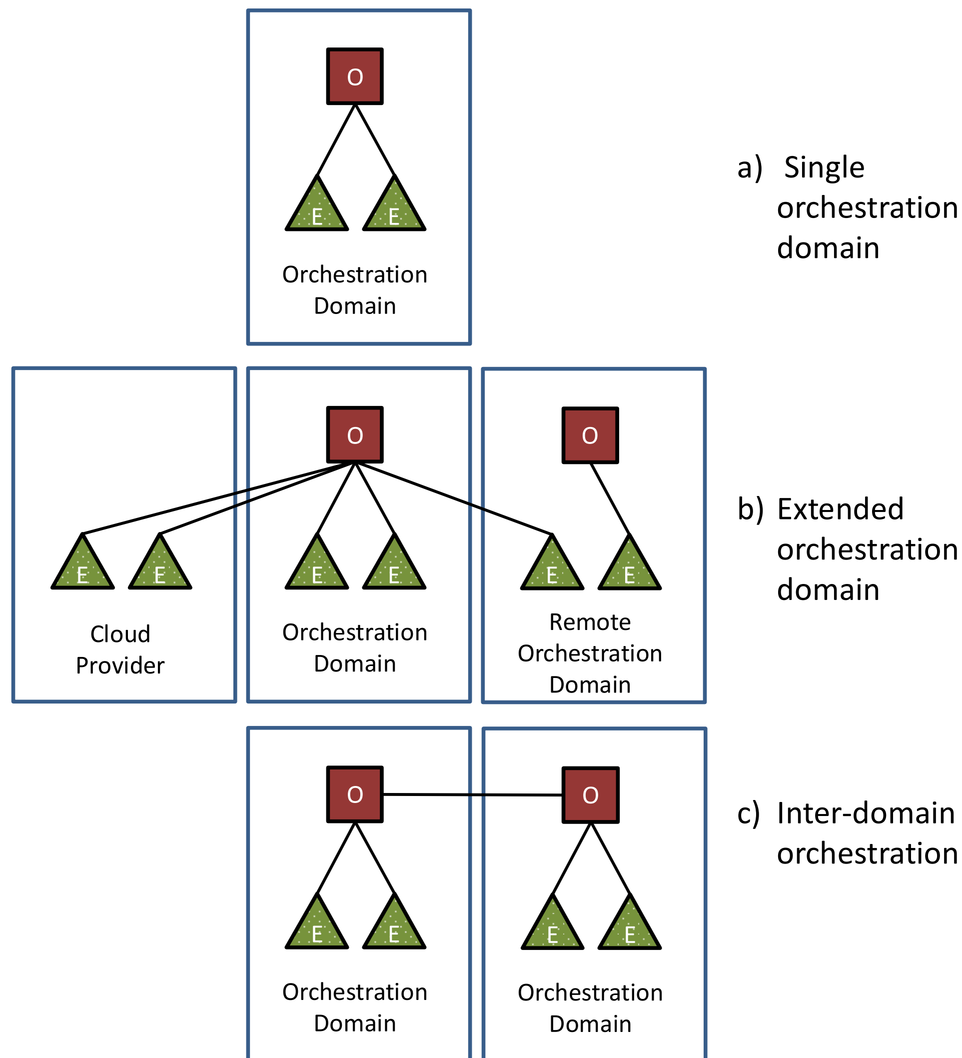

Figure 34: Single, extended and inter-domain orchestration.

Option (a) in the figure illustrates the situation where there is a single domain of ownership: a single organisation - such as an ISP deploying and operating FUSION services - owns execution zones running in its data centres, small servers attached to routers, access nodes or other equipment. In this case the ISP's orchestrator is constrained to deploying service instances in locations within its boundary of ownership.

In option (b) the original domain, shown in the centre, may extend the scope of its operation to remote execution zones which it contracts from other organisations. The example on the left of figure (b) is a cloud provider who offers computation resources, but does not participate in the orchestration of FUSION-compatible services. The example on the right of figure (b) shows a remote orchestration domain, e.g. another ISP offering FUSION-compatible services to its own customers within its own domain who additionally acts in the role of a cloud provider offering computational resources to the original orchestration domain. The set of the two execution zones in the cloud provider, the two execution zones belonging to the original orchestration domain and the left-hand execution zone within the remote orchestration domain become part of an *extended orchestration domain* under the control of the original orchestrator. The orchestrator is free to deploy, modify, remove any services it wishes within the constraints of the quantity of computational resources it has contracted, and is paying for, from the other two domains. In option (b) there is no inter-domain orchestration involved – the only inter-domain interactions in this option are those related to the contracting of computation resources prior to any subsequent orchestration actions to deploy

Copyright © FUSION Consortium, January 2014

services. The contractual interactions to negotiate the procurement of computational resources with remote data centres are considered outside the scope of the FUSION protocols.

Although the remote orchestration domain in figure (b) is running its own FUSION services it may not use the left-hand execution zone to deploy its own service instances, as these are resources it has contracted to its customer – the orchestration domain in the centre. It should be noted that the two execution zones shown in the remote orchestration domain may (or may not) coexist on the same physical data centre.

Option (c) depicts *inter-domain orchestration* where there are interactions between two orchestrators who cooperate in the deployment of FUSION services across the set of execution zones belonging to the two orchestration domains. There are three modes of inter-domain orchestration:

- Inter-domain orchestration mode 1: Orchestrator A is the primary orchestrator contracted by the application developer/service provider. Orchestrator A will receive the full service manifest from the app developer/service provider and may subcontract a portion of that manifest (e.g. to instantiate service instances in remote geographical regions) to orchestrator B. Orchestrator B will invoke its own service placement and scaling algorithms to deploy and manage instances within its domain according to the contracted manifest. In this mode the interactions between orchestrators A and B are equivalent to those between a service provider and an orchestrator.
- Inter-domain orchestration mode 2: As in mode 1 orchestrator A is the primary orchestrator but in this case orchestrator B acts as an aggregator, or one-stop shop, for all (or a subset) of its execution zones. This mode of operation is similar to Figure 34(b) but rather than individual execution zones being contracted by orchestrator A from orchestration domain B a set of execution zones are made available. In this mode the interactions between orchestrators A and B are functionally equivalent to the interactions between an orchestrator and its execution zones in the single or extended orchestration domain cases.
- Inter-domain orchestration mode 3: Orchestrators A and B operate independent services but establish contracts to allow their users to roam between the geographical regions covered by their respective orchestration domains. For example orchestrator A could operate in Europe, orchestrator B in South America. A customer of domain A from Belgium might be visiting Brazil for the World Cup and would like to make use of her interactive EPG or real-world tagging services which require local instances in Brazil to be made available for invocation to avoid the excessive latency or reduced bandwidth that would be caused by accessing instances back in Europe in the execution zones contracted to her home orchestration domain A.

5.4.7 Service routing

Service routers may run a distributed algorithm to establish their forwarding tables, but orchestration layer metrics may be taken into account. Examples include traffic distribution that takes into account detailed forecasts of user demand, or service-specific contracts whereby only particular regions of users may access the service.

As shown in Figure 35, one option for a routing management interface between the domain orchestration and the service routers. Although in this figure, the orchestration domain coincides with the routing domain, we keep the option open that a single orchestration domain spans multiple routing domains.

Copyright © FUSION Consortium, January 2014

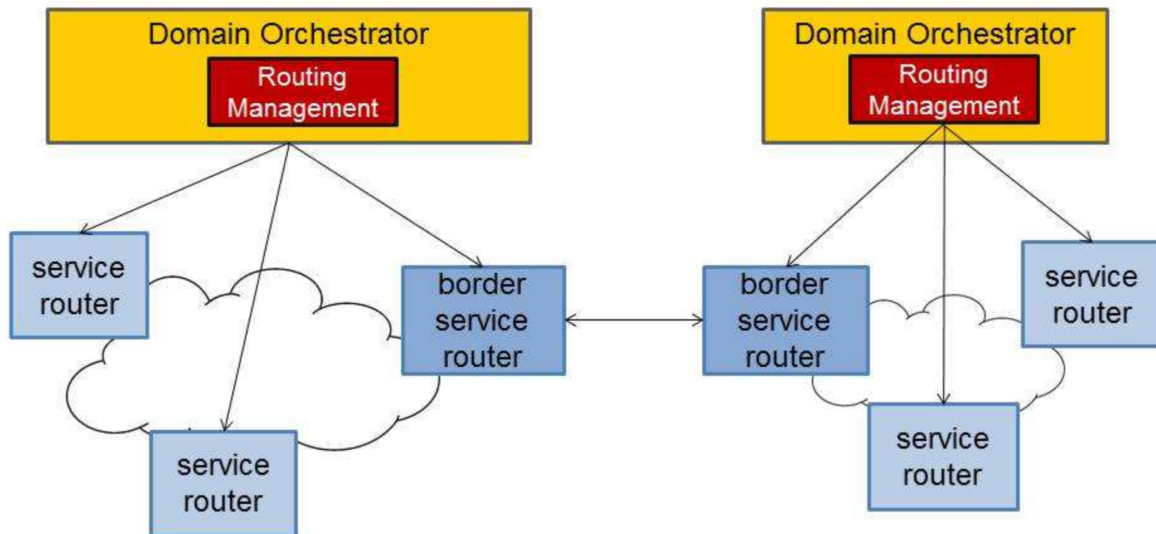

Figure 35: The Orchestration Layer may impose policies to the service routing layer.

The FUSION service routing and forwarding plane is specified in more detail in deliverable D4.1, where a distributed routing protocol is also developed as an alternative to the centralised routing management function collocated with the domain orchestrators as depicted in the above figure.

5.5 Management interfaces

In Figure 36, the initial key management interfaces of an orchestration domain are presented, clustered into four key interfaces:

- The orchestration management interface
- The execution zone management interface
- The inter-orchestration domain management interface
- The routing domain management interface

Copyright © FUSION Consortium, January 2014

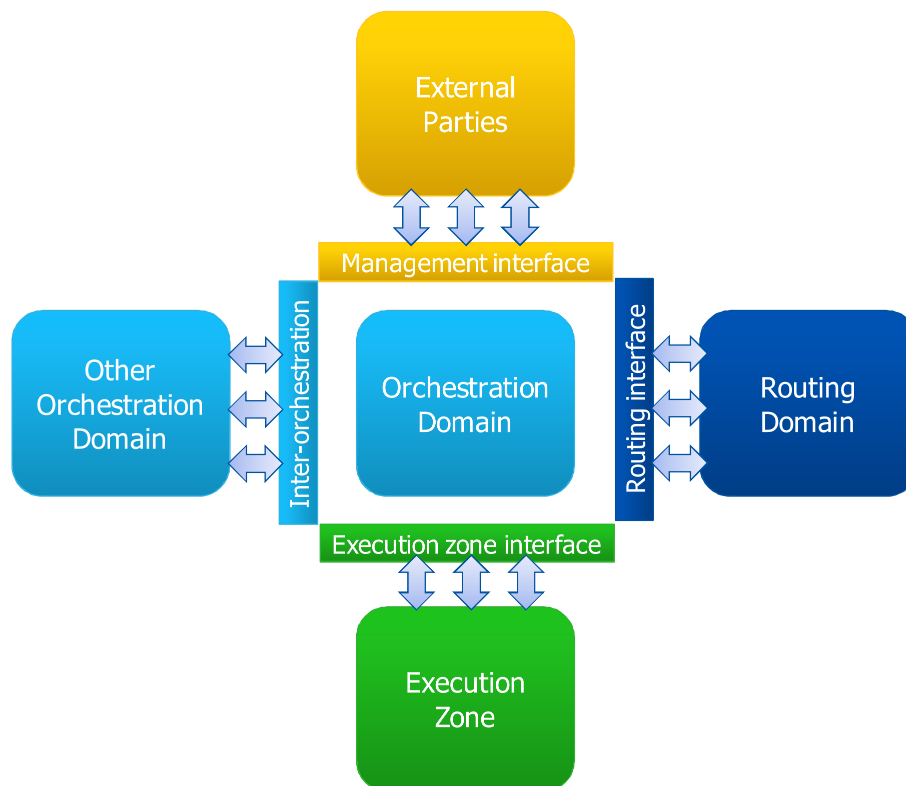

Figure 36: Key FUSION orchestration domain interfaces.

In the following sections, we will elaborate on each interface and provide details in the key functions, behaviour and initial parameters for each function.

5.5.1 Orchestration management interface

This interface represents the public interface towards external parties for accessing, managing and monitoring services and resources deployed and managed by an orchestration domain. It comprises a set of specialised interfaces as specified in the following subsections.

5.5.1.1 Service registration interface

Below an overview of the key functions related to service registration.

Copyright © FUSION Consortium, January 2014

Orchestration Management Interface		
FUNCTION NAME		
FUSIONRegisterService (service provider → domain orchestration)		
BEHAVIOR		
Register a new service type to an orchestration domain. In the current model, the manifest should contain all necessary information regarding the service, its policies and constraints and where to find the software packages and/or images.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceManifest	Manifest/URL	Service manifest (or URL to manifest) containing all information
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
ServiceID	FUSIONID	The registered FUSION service name

Orchestration Management Interface		
FUNCTION NAME		
FUSIONUnregisterService (service provider → domain orchestration)		
BEHAVIOR		
With this function, authorized users can remove an existing service type from an orchestration domain. This may include stopping all active service instances first.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceID	FUSIONID	Name of the service to unregister
StopServices	Bool	Immediately stop all existing instances
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Copyright © FUSION Consortium, January 2014

Orchestration Management Interface		
FUNCTION NAME		
FUSIONUpdateService (service provider → domain orchestration)		
BEHAVIOR		
This function can be used to provide updates to an existing service type. This may include updated versions of the software packages, changed requirements, policies or cost models, load patterns, etc.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	ServiceName	Name of the service to update
Manifest	Manifest/URL	New/updated manifest for the service
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

5.5.1.2 *Service querying interface*

Below an overview of the key functions related to the capability of querying the orchestration domain about particular services.

Orchestration Management Interface		
FUNCTION NAME		
FUSIONQueryServiceByName (service provider → domain orchestration)		
BEHAVIOR		
This function allows authorized users to query for particular information of a specific service type.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceID	FUSIONID	Name of the service under query
InformationType	Enum	Subset of what kind of information needs to be retrieved
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
Description	Structured	Structured description of the requested information (XML, manifest, etc.)

Copyright © FUSION Consortium, January 2014

Orchestration Management Interface		
FUNCTION NAME		
FUSIONQueryServiceByCategory (service provider → domain orchestration)		
BEHAVIOR		
This function allows authorized users to query for particular services providing particular functionality or have particular requirements or features.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Categories	List	List of features to look for
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
ServiceNames	[ServiceName]	A list of all registered services that match the query

5.5.1.3 *Service deployment interface*

Below an overview of the key functions related to service deployment, triggered externally by users.

Orchestration Management Interface		
FUNCTION NAME		
FUSIONDeployService (service provider → domain orchestration)		
BEHAVIOR		
This function allows authorized users to deploy extra instances of a particular service type		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceID	FUSIONID	Name of the service to deploy
ServiceParameters	Blob	Service-specific deployment parameters
DeployParameters	Structured	FUSION-specific deployment parameters (number of sessions, location, cost, etc.)
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
InstanceIDs	[FUSIONID]	List of all instances that have been created

5.5.1.4 Accounting/billing interface

We will not focus on these aspects within the FUSION project.

5.5.1.5 Security interface

The security aspects of all these interfaces and components will be tackled in Deliverable D3.2.

5.5.2 Execution zone management interface

This public interface serves execution zones to register itself and communicate with an orchestration domain.

5.5.2.1 Zone registration interface

Given below is an overview of the key functions for registering and managing execution zones in a domain.

Zone Management Interface		
FUNCTION NAME		
FUSIONRegisterZone (zone administrator → domain orchestration)		
BEHAVIOR		
With this function, a new execution zone can announce and register		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Zone	URI	Name of the execution zone
Parameters	Complex	A detailed description of the zone, its location and its capabilities
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
ZoneID	FUSIONID	A unique zone ID for this execution zone in the domain

Copyright © FUSION Consortium, January 2014

Zone Management Interface		
FUNCTION NAME		
FUSIONUpdateZone (zone administrator → domain orchestration)		
BEHAVIOR		
With this function, an execution zone can modify its registration parameters to the domain (e.g., change key capabilities, etc.)		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ZoneID	FUSIONID	Execution zone identifier
Parameters	Structured	An updated version of the execution zone specifications
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Zone Management Interface		
FUNCTION NAME		
FUSIONUnregisterZone (zone administrator → domain orchestration)		
BEHAVIOR		
With this function, an execution zone can unregister itself from a domain		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ZoneID	FUSIONID	Execution zone identifier
StopServices	Bool	This flag indicated whether it will immediately shut down existing services (this request can be refused by the orchestrator).
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Copyright © FUSION Consortium, January 2014

5.5.2.2 Monitoring interface

Given below is an overview of the key functions for feeding monitoring information from an execution zone to the orchestration domain. Currently, we are still considering both the push and pull based models for sharing monitoring information in between FUSION entities. It is still under investigation which model or what combination is optimal for distributed service management within FUSION.

Zone Management Interface		
FUNCTION NAME		
FUSIONUpdateServiceInformation (zone manager → domain orchestration)		
BEHAVIOR		
With this function, an execution zone provides the orchestration domain with up-to-date information concerning FUSION services running in the execution zone		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ZoneID	FUSIONID	Execution zone identifier
Information	Structured	Service monitoring information
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Zone Management Interface		
FUNCTION NAME		
FUSIONUpdateResourceInformation (zone manager → domain orchestration)		
BEHAVIOR		
With this function, an execution zone provides the orchestration domain with up-to-date information concerning the available resources and overall health of the execution zone		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ZoneID	FUSIONID	Execution zone identifier
Information	Structured	Resource monitoring information
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Copyright © FUSION Consortium, January 2014

5.5.3 Inter-orchestration management interface

The interactions between domains envision in FUSION were introduced in Section 5.4.6. The extended orchestration domain option in Figure 34 (b) does not make use of an inter-orchestration interface and is not considered further here.

Inter-domain orchestration mode 1 has orchestrator A acting in the application developer/service provider role and orchestrator B acting in the orchestrator role and hence the interfaces are equivalent to the Orchestration Management Interface specified in Section 5.5.1 covering service registration, service querying, service deployment, accounting & billing and security.

Inter-domain orchestration mode 2 has orchestrator A acting in the orchestrator role and orchestrator B acting in the execution zone role and hence the interfaces are equivalent to the Execution Zone Management Interface specified in Section 5.5.2 covering zone registration and monitoring and the Orchestration Management Interface specified in Section 6.4.1 covering zone selection, service lifecycle and monitoring. An additional parameter is required in the functions specified as part of the Orchestration Management Interface to identify the specific execution zone in the remote orchestration domain that is the target of the management action:

PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ZoneID	FUSIONID	Execution zone identifier

Inter-domain orchestration mode 3 has roaming between orchestration domains. Interactions are required to map equivalent service capabilities between domains, to authenticate users and their service requests and to perform accounting and billing transactions. A deeper investigation of the interactions will be undertaken in the second year of the project and will be reported in D3.2.

5.5.3.1 Inter-domain routing management interface

In all three inter-domain orchestration modes routing information needs to be exchanged between domains. In the case of centralised routing management within orchestration domains (see Figure 35) this will be achieved with the interface specified below. In the case of a distributed routing protocol the inter-domain routing protocol requirements are part of the service routing plane and are introduced in deliverable D4.1 with a full specification to be produced in deliverable D4.2 at the end of the second project year.

Copyright © FUSION Consortium, January 2014

Inter-domain Routing Management Interface		
FUNCTION NAME		
FUSIONAnnounceServiceID (domain orchestrator A → domain orchestrator B)		
BEHAVIOUR		
Orchestration domain A announces the existence of serviceIDs available through its border gateway service router. These are either running in execution zones within the domain or in the case of transit domains they are available in remote domains.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceID	FUSIONID	The FUSION service name.
ServiceSlots	Int	Available service slots.
BorderRouterID	Locator	Locator of next hop border gateway service router that the border gateway service router in domain B should use.
RouteInformation	Structured	Route characteristics, hop count, domain path (in the case of transit, equivalent of AS path/set in BGP).
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Inter-domain Routing Management Interface		
FUNCTION NAME		
FUSIONUpdateRoute (domain orchestrator A → domain orchestrator B)		
BEHAVIOUR		
Orchestration domain A updates the routing information for a serviceIDs when the number of session slots or route information changes.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceID	FUSIONID	The FUSION service name.
ServiceSlots	Int	Available service slots.
BorderRouterID	Locator	Locator of next hop border gateway service router associated with this update.
RouteInformation	Structured	Route characteristics, hop count, domain path (in the case of transit, equivalent of AS path/set in BGP).

RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Inter-domain Routing Management Interface		
FUNCTION NAME		
FUSIONWithdrawServiceID (domain orchestrator A → domain orchestrator B)		
BEHAVIOUR		
Orchestration domain A announces that a ServiceID or its route is no longer available through the specified border gateway service router.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceID	FUSIONID	The FUSION service name.
BorderRouterID	Locator	Locator of next hop border gateway service router.
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

5.5.4 Routing and networking management interface

There are two options for managing service routing discussed in D2.1 and in more detail in D4.1: 1) centralised routing decisions made within a domain with the routing management algorithm being collocated with service orchestration functions; and 2) distributed routing algorithms collocated with the forwarding functions of service routers. In the first option the routing and forwarding functions are in separate entities (see Figure 35) and the interfaces for configuring the forwarding tables are specified below. In the second option routing updates are exchanged through a distributed routing protocol and the configuration of forwarding tables is an internal interface within the service routers. For the second option the routing protocol and internal interface to forwarding tables is part of the FUSION service routing plane and, as such, is discussed in deliverable D4.1.

5.5.4.1 Router configuration interface

Routing Management Interface
FUNCTION NAME
FUSIONCreateForwardingEntry (domain orchestrator → service router)
BEHAVIOUR
Forwarding entries are created in the service router according to the routing decisions made by the routing management functions collocated with the domain orchestrator.

PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
RouterID	FUSIONID	Service router identifier
EntryID	EntryID	Forwarding table entry identifier
Information	Structured	Forwarding entry, containing ServiceID(s), load balancing parameters, locator(s) of next hop
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Routing Management Interface		
FUNCTION NAME		
FUSIONUpdateForwardingEntry (domain orchestrator → service router)		
BEHAVIOUR		
Forwarding entries are updated in the service router according to the routing decisions made by the routing management functions collocated with the domain orchestrator – to modify the set of ServiceIDs, load balancing parameters or net hop locator(s).		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
RouterID	FUSIONID	Service router identifier
EntryID	EntryID	Forwarding table entry identifier
Information	Structured	Forwarding entry, containing ServiceID(s), load balancing parameters, locator(s) of next hop
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Routing Management Interface
FUNCTION NAME
FUSIONDeleteForwardingEntry (domain orchestrator → service router)
BEHAVIOUR
Forwarding entries are deleted from the service router according to the routing decisions made by the routing management functions collocated with the domain orchestrator.

Copyright © FUSION Consortium, January 2014

PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
RouterID	FUSIONID	Service router identifier
EntryID	EntryID	Forwarding table entry identifier
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

5.5.4.2 Monitoring interface

Routing Management Interface		
FUNCTION NAME		
FUSIONProvideRoutingInformation (service router → domain orchestration)		
BEHAVIOUR		
Service routers provide monitoring data to the routing management functions in the orchestrator concerning historical service request and network performance statistics collected from the perspective of that service router.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
RouterID	FUSIONID	Service router identifier
Information	Structured	Request and network resource monitoring information
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Copyright © FUSION Consortium, January 2014

6. DISTRIBUTED SERVICE EXECUTION MANAGEMENT

In this section, we discuss in detail the functional design of a FUSION execution zone for deploying and managing FUSION services on top of an existing data centre or physical infrastructure.

6.1 Functional design

The execution zones are very closely coupled to the physical execution platforms and resources on which FUSION services will be deployed and managed. Similar to an orchestration domain, an execution zone consists of a number of key functional components each dedicated to a very specific set of tasks.

6.1.1 Overall design decisions

For designing a FUSION execution zone, we propose a PaaS approach with the following key characteristics. First, for compatibility reasons, we will initially deploy the FUSION execution zone services on top of existing data centre or hardware infrastructure management systems, with optional hooks deep into the infrastructure and management layers for extra efficiency or reliability. This approach allows to quickly enable existing (cloud, grid, etc.) centralized and distributed infrastructures to become FUSION-ready to start deploying and managing FUSION services on top of these infrastructures. This also enables a potentially faster adoption of the FUSION APIs in existing environments. For designing the overall orchestration and management of an execution zone, we will also use the same modular approach as for the FUSION domain-level orchestration services, with REST APIs in between the key execution zone components. This will result in a scalable solution that can easily be deployed and managed on potentially large data centres.

In order to minimize the amount of functionality and components that must be ported or rewritten for different underlying infrastructures or management systems, we envision a DC abstraction layer for all critical interactions with the underlying infrastructure, as we will discuss later. In such a way, most functions of the execution zone manager can be reused across different implementations, and only the DC abstraction layer needs to be implemented for a specific environment. This is shown in Figure 37 below. The FUSION zone manager runs on top of a FUSION DC management backend that is optimized for the existing DC management layer currently running on that infrastructure. Note that a bare metal environment is just a special case. In some implementations, we envision that part of the FUSION DC abstraction layer could have direct access to the underlying hardware, for example to enable higher efficiency. The FUSION service instances are automatically wrapped by the DC abstraction layer into whatever execution environment the underlying data centre management layer supports. This could be a native environment (e.g., grid cluster), or a virtualized environment (e.g., an OpenStack or EC2 environment, using various types of hypervisors).

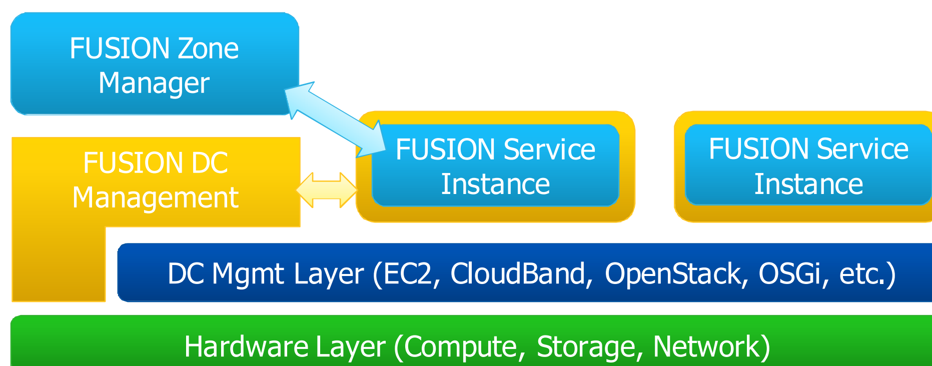

Figure 37: Deploying a FUSION execution zone on top of existing managed data centres.

Copyright © FUSION Consortium, January 2014

6.1.2 High-level design

The high-level functional design of the key functions of the execution zone and their primary interactions are shown in Figure 38. Note that most functions are considered to be part of the execution zone manager. For clarity, we did not include a security component and all its interconnections, but obviously, all authentication and policy decisions need to be handled by this component.

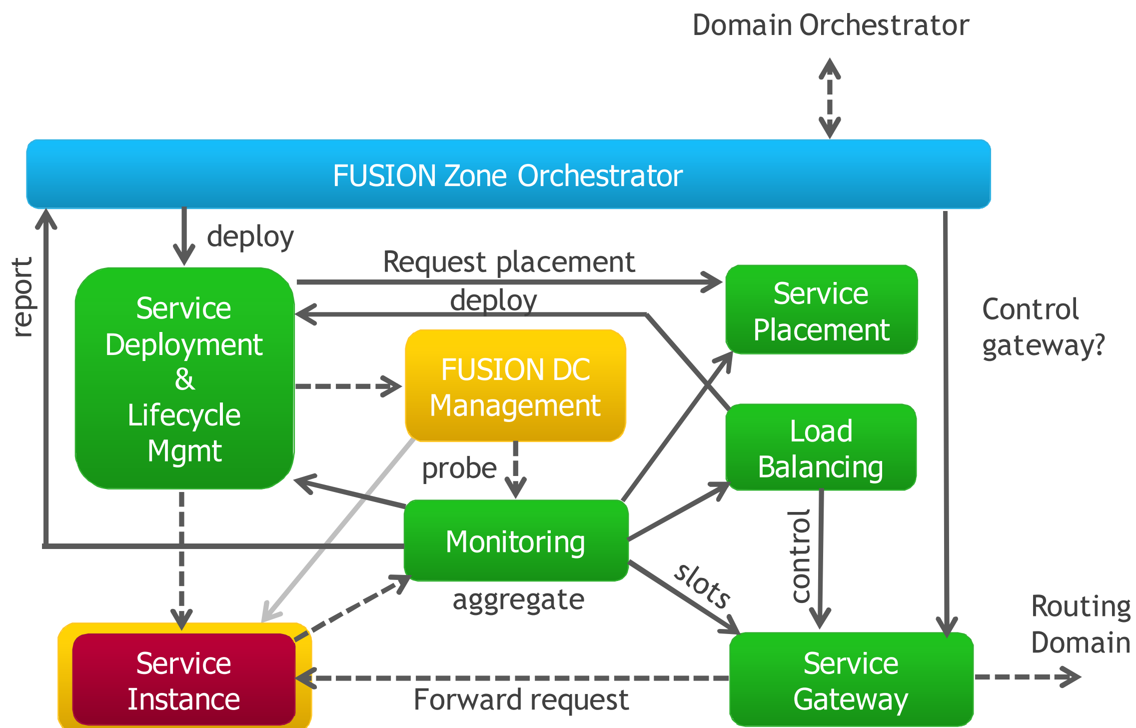

Figure 38: High-level design of a FUSION zone manager and its key functions.

- FUSION zone orchestrator

  The zone orchestrator handles the overall management and coordination of the execution zone. This component will implement the external interface towards the domain orchestration functions, and will also manage and control the other internal zone management functions (lifecycle management, service placement, etc.). This also includes ensuring proper operation of the execution zone as a whole and the individual services in particular and may involve several self-healing actions. Note that we may decide to split that function into a set of smaller modules as we move forward.

- Service lifecycle management

  This component is responsible for the life cycle management of all FUSION services and instances that are deployed within the execution zone. This includes managing the state of all FUSION services, communicating with the FUSION DC abstraction layer for doing the actual deployment and resource allocation, as well as taking into account the service monitoring information related to the overall health and operation of each service instance.

- Service scaling / load balancing

This component is responsible for ensuring proper balancing and scaling of FUSION service atoms within the zone. Depending on the requirements or constraints from the service as well as the complexity of the execution zone, an execution zone could automatically decide to scale in or out a number of service instances, or decide to migrate instances from one location to another.

- Service placement

  This component is responsible for finding the optimal set of hardware resources as well as physical location of a particular service instance within an execution zone. This component needs to take into account the service requirements on the one hand (e.g., resource requirements, being collocated with another service instance, etc.), and the infrastructure capabilities and load on the other hand. In case of large data centres, the placement across server blades and server racks may be important as well. Because a FUSION execution zone will typically be deployed on top of existing cloud management software, the accuracy and detail of what resources are available where and the overall placement accuracy may differ from implementation to implementation. For example, it may not be possible to easily guarantee deploying two service instances on the same physical machine, or to have a complete overview of the layout of the data centre – these capabilities will be imposed/limited by the actual interface to the underlying cloud infrastructure.

- Monitoring

  This component is essential for the overall operation and health of the service instances. This component will capture both monitoring information coming from all active service instances (e.g., the available session slots, the actual frame rate, etc.), as well as capture monitoring information from the physical infrastructure using probes. This information will be partially aggregated and forwarded to many other internal zone management functions. Some information will also implicitly or explicitly be forwarded towards the orchestration and the routing domains.

- Service gateway

  This component is part of the execution zone as well as at least one FUSION routing domain. External service requests will be terminated here and will be forwarded towards an appropriate service instance.

- DC abstraction layer

  As mentioned before, in the initial implementations, we envision an overlay PaaS approach, where the FUSION zone orchestration services are running on top of an existing cloud or infrastructure management platform. The DC abstraction layer is a crucial component that will shield most, if not all, DC specific functions from the other zone management functions. This includes the actual service embodiment and deployment, the monitoring information, etc. We do envision the capability for this DC abstraction layer to in some cases also have direct hooks or access to some parts of the hardware for higher efficiency.

6.2 Lifecycle management of an execution zone

As we currently assume an overlay approach for the FUSION execution zones, we assume that the physical infrastructure and the corresponding data centre management software is already operational. For deploying the execution zone functions, we are currently considering two main models.

In the first model, we assume that most of the zone manager functions are actually deployed and managed fully agnostic of the underlying environment. In this case, the DC abstraction layer should be deployed first, which then also has some basic functionality and capabilities for automatically

Copyright © FUSION Consortium, January 2014

deploying the other EZ services, which could be modelled as FUSION services themselves. The first EZ service to deploy is the EZ deployment and lifecycle management service, automatically wrapping them as needed in VMs or containers as supported by the data centre. Next, the other EZ services can then be deployed via the EZ deployment service, in which case the health and scaling of these services would then also be under the control of the execution zone itself using the same mechanisms as the FUSION application services. Although this model has the advantage of having the least dependencies with the underlying infrastructure, it does complicate the overall deployment and management of the EZ services.

In the second model, all EZ services are made compliant to the underlying DC management platform, and they are all deployed and managed using the interfaces and using the image formats supported by the DC management platform. The lifecycle management of the EZ services is also managed by the DC management platform. This model significantly simplifies the overall deployment of the EZ services on top of an existing DC infrastructure. However, this also implies that all EZ services need to prepared, tuned and deployed separately for all different types of DC infrastructures on which an EZ needs to be deployed.

After the EZ services are deployed, the execution zone needs to be connected to at least one orchestration domain as well as routing domain. This can be done in a number of ways:

- The home address of the orchestration services as well as service router(s) can be provided manually during deployment of the execution zone.
- The orchestration and routing domains could also be automatically discovered, for example via a FUSION broadcast mechanism. One approach here could be to first discover the home routing domain for that execution zone, which in its turn may then also provide the corresponding home orchestration domain.
- In case there is no orchestration domain available yet, an orchestration domain could also be deployed automatically inside that execution zone. This could then be the home orchestration domain for other execution zones as well. Alternatively, when another home orchestration domain is announced, the existing orchestration domain could decide to migrate its state to the new orchestration domain and shut itself down.

6.3 Key functions

This section provides details on the key functions of an execution zone for managing FUSION services that deployed on top of a particular data centre or compute environment.

6.3.1 Selecting a zone for service deployment

In FUSION, we envision evaluator services and execution zones to play an important and active role in helping to make the decision about the execution zone in which a particular new service should be deployed. An orchestration domain may request individual execution zones for making an assessment and offer for deploying a service in its execution zone. An execution zone in its turn may rely on external evaluator services for providing valuable input for making an appropriate offer. The entire process is depicted in Figure 39, and assumes the deployment of one (or more) service atoms within an execution zone.

Copyright © FUSION Consortium, January 2014

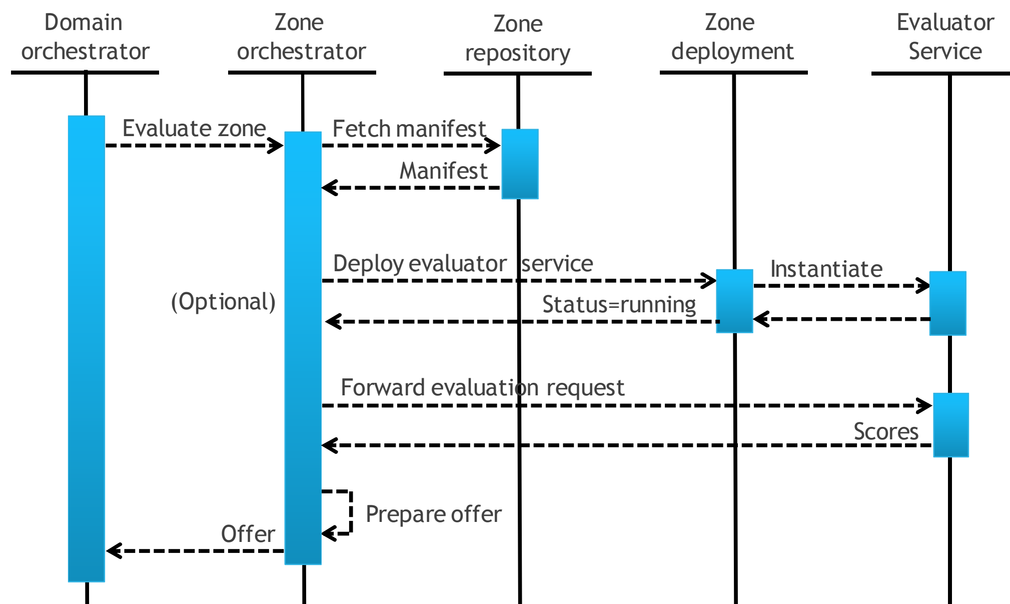

Figure 39: Sequence diagram for selecting a zone for service deployment.

1) The orchestration domain requests a zone to evaluate the deployment of new service instances with particular service-specific and FUSION-specific parameters.
2) The zone orchestrator subsequently first fetches all relevant information regarding that service type from an optional internal zone service repository, which caches all relevant information regarding previously deployed service types. This may include the service manifest, the software packages, historical monitoring data, etc. If the repository does not have actual information regarding the service, it needs to request the information from the orchestration domain (not shown on the diagram). Alternatively, the orchestration domain could also send the manifest along with the evaluation request.
3) In case the service requires a special evaluator service for making the evaluation, and in case this evaluator service is not deployed yet in the execution zone, the zone manager can decide to first deploy the evaluator service as a new service. Note that the zone manager could also decide not to deploy the evaluator service (e.g., due to time or policy constraints), and simply deny the zone evaluation request. When it is receiving many evaluation requests, it can decide to deploy that evaluator service to be able to evaluate future requests.
4) The evaluation request is subsequently forwarded to the appropriate evaluator service (which could be a generic evaluator service), which will do all necessary evaluations and which will return a score, which can range from a simple float value to a more complex set of key-value pairs.
5) Using the score, the zone manager will make an appropriate offer towards the orchestration domain. Note that due to internal policies, the zone manager may decide to decline the offer of change the conditions of the offer. For example, although a zone may be capable for deploying new instances of a service, it may decide not to do so in order to better serve future instantiation requests.

6.3.2 Service placement

Service instances that will be deployed inside a FUSION execution zone needs to be placed on the appropriate data centre infrastructure. In the simplest case, the zone manager could rely on the available service placement functionality that is provided by the existing data centre management

Copyright © FUSION Consortium, January 2014

layer. This may involve translating FUSION service specific requirements related to compute, memory, storage, network and QoS onto the available flavours and data centre locations.

Depending on the amount of flexibility and configurability supported by the underlying DC management layer, the FUSION zone service placement may have either only limited impact on the service placement, or may have a strong impact. Due to the specific requirements that FUSION services may have (for example related to specialized hardware or QoS guarantees), only having a limited impact on the service placement and resource control may not be enough however. In such case, extensions could be made to the DC management layer to enable more fine-grained control over the DC resources. This could for example be achieved by providing plug-ins and extensions to OpenStack for supporting these capabilities, which the FUSION zone management function subsequently can use. In the project, we will explore this option for specific scenarios and test cases (e.g., exposing heterogeneous hardware, probing and enabling efficient inter-service communication mechanisms, etc.)

6.3.3 Service deployment

Service deployment in an execution zone is a complex function that consists of many key steps with the final goal to have a new FUSION service instance up and running in a particular execution zone. This involves proper placement, fetching the software, preparing an execution point environment, and instantiating the FUSION service inside that environment. This section discusses these steps in more detail.

6.3.3.1 Service deployment scenario

In this section, we discuss a typical FUSION service deployment scenario. We assume that there is an explicit trigger from the orchestration domain to deploy a new instance of a particular service in that execution zone. Note that the trigger could also come from the execution zone itself (load balancing) or even from the routing domain. We assume that the first three steps of the domain-level service deployment already have been completed and that this execution zone has been selected to deploy one or more instances. The different steps are depicted in Figure 40.

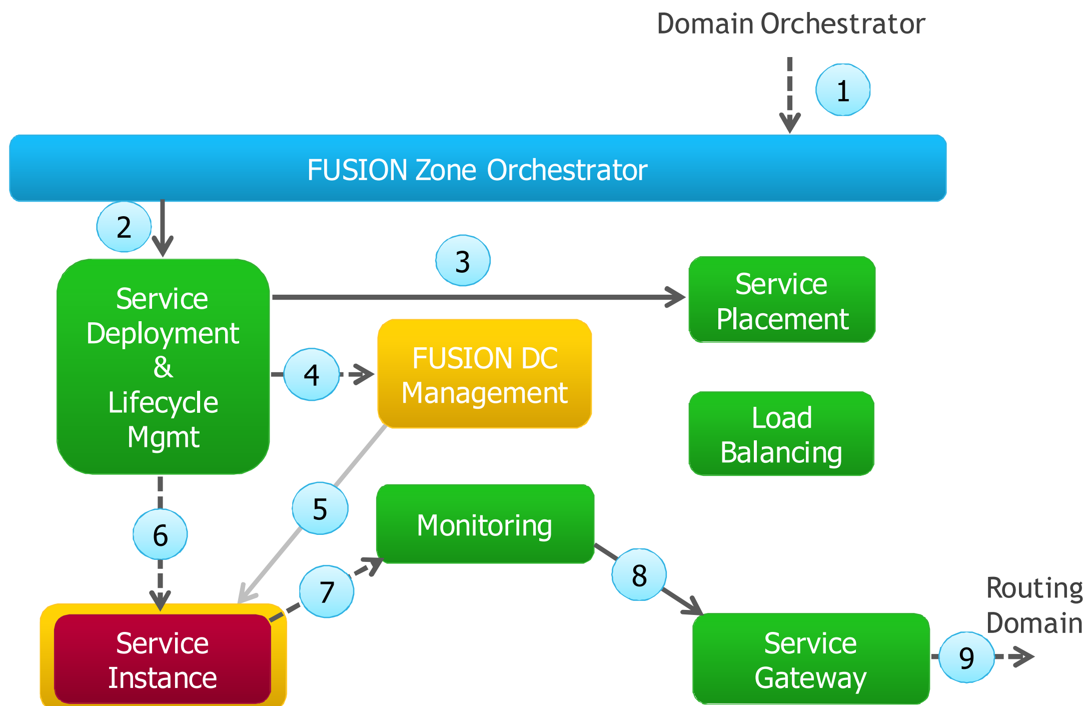

Figure 40: Sequence flow for deploying new FUSION instances in an execution zone.

Copyright © FUSION Consortium, January 2014

1) The domain orchestrator first triggers the selected execution zone to deploy new service instances using the deployment interface. The zone orchestrator will handle the request, check whether the request aligns with previously made engagements, and sends an initial response back, while further processing and managing the request. As mentioned earlier, we assume that the execution zone selection procedure has already been completed.
2) The zone orchestrator then triggers the lifecycle management service to start deploying new instances of that service type inside the execution zone. This is a complex step that involves many subsequent smaller steps that we will discuss in more detail in the next section. This includes fetching all necessary information concerning that service type from a repository or external location. This includes the software packages, images or snapshots and other relevant information that is required to deploy that service. In case the latest version is already available, this data may also be fetched from a local repository.
3) Before the service can actually be instantiated, the zone manager also needs to find the best location inside the data centre to place the corresponding service atoms involved in the service instantiation process, based on service requirements and resource availability. Note that the placement service may reuse cached placement decisions made earlier, for example during the evaluation phase or during previous deployment requests. Note also that this step can occur in parallel to the fetching of all necessary software packages or images.
4) The lifecycle management service then requests the DC abstraction layer to prepare the software package or image for deployment, for creating the necessary environment and allocating the necessary resources for hosting the new instance(s). The preparation step may involve creating and installing a new VM image based on the provided software packages, resolving all dependencies, etc. It may also involve registering and uploading the VM into the DC management platform.
5) Next, the DC abstraction layer starts creating the necessary environment for creating new instances. This may involve actually physically deploying and preparing the environment, or simply instructing an existing cloud management platform to create a new instance of a particular image using a particular flavor. This step also includes setting up the necessary intra-zone networking to make the service visible and available internally and/or externally.
6) Once the new environment is ready, the lifecycle manager triggers the FUSION instance (which is embedded in that environment) to start running, passing on the service instantiation parameters provided during the instantiation request. We envision a number of extra FUSION-related management services to also be present in that environment, which will assist in a number of tasks like lifecycle management, monitoring, self-healing, etc.
7) Once the FUSION application service is up and running, it should announce its presence and announce its available session slots (amongst other information) towards the monitoring service.
8) This monitoring service will keep track of this information and forward it towards all necessary management services, including the service gateway.
9) The service gateway finally injects this information (i.e., the extended session slot information) into the routing domain so that new service requests can be forwarded towards these newly created instances. Note that the orchestration domain or execution zone could already have inserted this information provisionally in the routing domain, before the instances are already available, to reduce the amount of time before the new instances are visible and routable from the routing domain.

Copyright © FUSION Consortium, January 2014

6.3.3.2 *Service lifecycle management*

In this section, we discuss lifecycle management of FUSION services within an execution zone. We focus on the lifecycle management of one particular service atom, and we start by zooming in on the deployment and instantiation phase. This is shown in Figure 41.

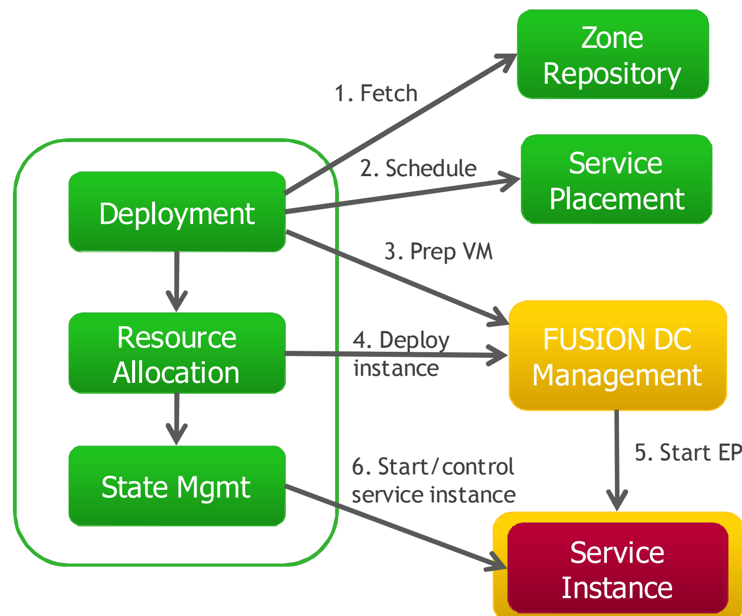

Figure 41: More detailed deployment sequence flow on top of an existing DC management layer.

1) In a first step, the latest version of the service metadata and all necessary software packages or images need to be fetched remotely. This application data can partially come from a FUSION repository inside an execution zone or from the domain orchestration, but may also partially come from external repositories in case (some of) the data is stored externally. In many cases, the execution zone will cache this data (if allowed by the service provider) to significantly reduce the deployment latency and download bandwidth when new requests to deploy that service are issued, especially as these software packages can be quite significant for non-trivial application services.

2) As soon as the service metadata is available to the deployment service, it can issue the service placement function to find the optimal location for deploying the new instance within the data centre. Note that this step may also reuse previous service placement results. The service placement function may interact with the DC abstraction layer, as the FUSION placement function may not have direct impact on the exact low-level placement onto the physical infrastructure.

3) In the third step, the environment or execution point in which the service instance will be deployed, needs to be prepared. As this is a DC specific function, it is up to the DC abstraction layer to perform this function. The preparation step could involve installing all necessary software packages and configurations inside a DC-specific VM. It could also involve deploying the same software already on the physical hardware, for example in case of light-weight containers. This will likely also involve installing FUSION-specific software utility packages that will assist in the overall operation of the FUSION service within the data centre. Again, this process of encapsulation could be cached: DC-specific VMs containing all necessary software packages and pre-configurations could be stored in a local (or even remote) repository so that they can easily be reused during later deployments.

Copyright © FUSION Consortium, January 2014

4) Once the service preparation and encapsulation is done, the resource allocator will be triggered to allocate a particular amount of resources and instantiate a new instance. This function will also be handled by the DC abstraction layer, as this is very DC-specific.

5) The DC abstraction layer will then bootstrap the environment containing the service instance. This may involve spawning a new VM instance using a particular flavour or instance type (e.g., a tiny, medium or large instance), creating and starting a new container, booting a system, etc.

Once the containing environment or execution point has been bootstrapped, the lifecycle management function should trigger the FUSION application service to start running. This can be done in a number of ways, and will be worked out in more detail during the project. One option is to trigger a FUSION helper service inside the environment that knows how to start that service with the necessary parameters. This helper service is an agent that will moderate and coordinate all lifecycle management functions between the zone manager and the application service. This includes starting and stopping an application, extracting monitoring information, triggering specific actions when particular events occur, etc.

A second key function is service termination. In this case, the active service instance will be terminated and removed from FUSION. The trigger for termination could be multifold, either by the service instance itself, explicitly by the service provider, or due to an automatic scaling down operation issued by either the load balancing function of the zone manager or the orchestration domain. This is shown in Figure 42, where we assume the trigger is coming from the orchestration domain.

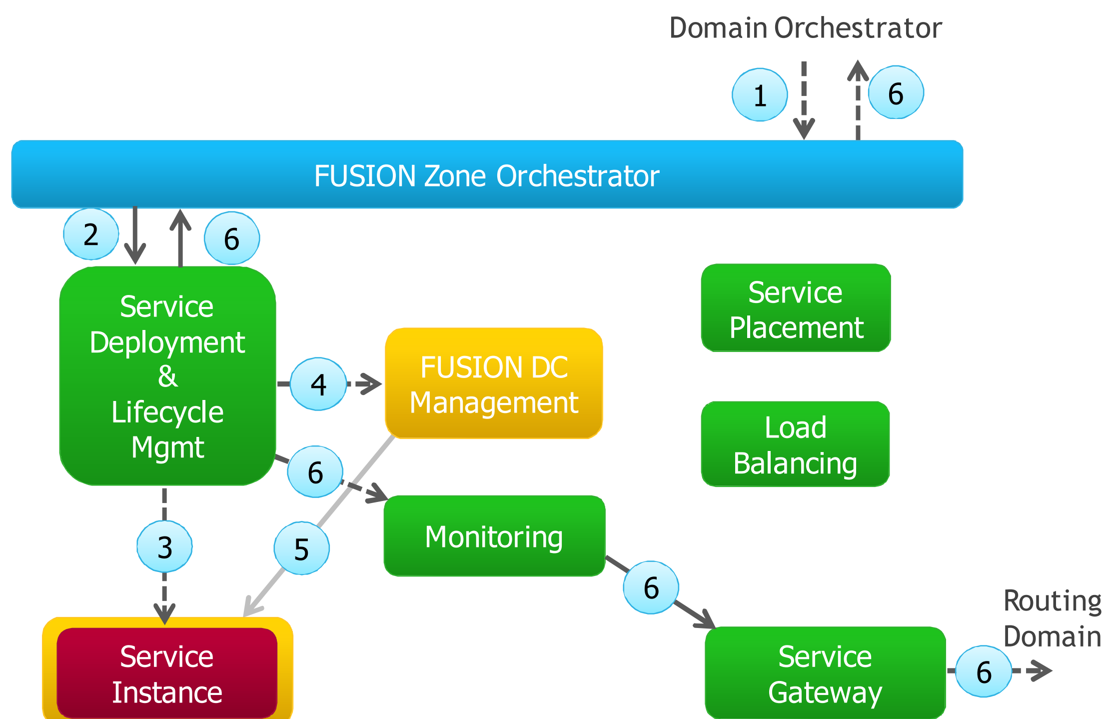

Figure 42: High-level sequence flow for service termination.

1) The zone orchestration service receives a request from the domain orchestrator to either explicitly reduce the amount of available service instances or to reduce the amount of session slots for a particular service type, irrespective of the amount of instances it comprises.

2) The zone orchestrator forwards the request towards the lifecycle management function, which will decide to either just trigger one or more instances to reduce the amount of session slots, or

Copyright © FUSION Consortium, January 2014

to explicitly terminate idle instances that are currently not handling any sessions. The state manager should try to reduce the session slots in an intelligent manner and avoid a fragmentation of many unused session slots spread across many service instances. At the end, this may even involve a compaction phase, where sessions are migrated to other instances or simply terminated.

Note that in our design, we envision an execution zone to have more freedom and flexibility towards deploying a particular amount of instances with respect to host a particular amount of service sessions within the execution zone. The execution zone could decide to be more or less aggressive with respect to the amount of deployed instances or available sessions slots than the orchestration domain requests. An execution zone could for example only publish a subset of its available instances or session slots to one particular domain. Consequently, the amount of instances or session slots the zone orchestrator requests to reduce could differ from the amount requested by the orchestration domain.

3) Assuming an instance was selected for termination, the instance is notified, for example via the FUSION utility service that is also deployed in the same environment, to enable a graceful termination of the instance and potentially any active sessions. When the instance is not responding, the instance will be terminated abruptly after a timeout period. The monitoring service will be notified of the reduction in available session slots, which will also propagate into the routing domain (step 6).

4) The lifecycle manager may then decide to terminate or simply clean the enclosing environment. This may involve terminating the execution point and its environment and removing it from the system. FUSION could also decide to simply clean the environment to be able to quickly reuse it for deploying a new instance of potentially other service types. Similarly, the hardware resources are freed so that they can be used for other service instances.

5) In this step, the DC abstraction layer triggers the environment to be reset or to terminate.

6) Finally, the zone orchestrator is notified of the successful termination of the requested amount of sessions slots or service instances, which may also notify in its turn the orchestration domain; similar information is also propagated to the routing domain.

6.3.3.3 Optimizing service placement and deployment

Because of the intrinsically demanding nature of the services we envision to be deployed on a FUSION architecture, optimizing the placement, deployment and configuration of a particular FUSION application service onto the physical resources of a particular execution zone, it is important to take into account the available requirements and monitoring data from the services on the one hand, and to combine this with the nature and availability of the physical resources (compute, memory, network, storage, other, etc.), the characteristics of the underlying software platform (hypervisors, schedulers, deployment framework, etc.) as well as the location and characteristics of the already deployed FUSION services. The latter can be leveraged for optimizing inter-service communication, following anti-affinity policy rules, taking into account interference patterns in between services deployed on the same infrastructure, etc. The overall architecture is depicted in Figure 43.

Copyright © FUSION Consortium, January 2014

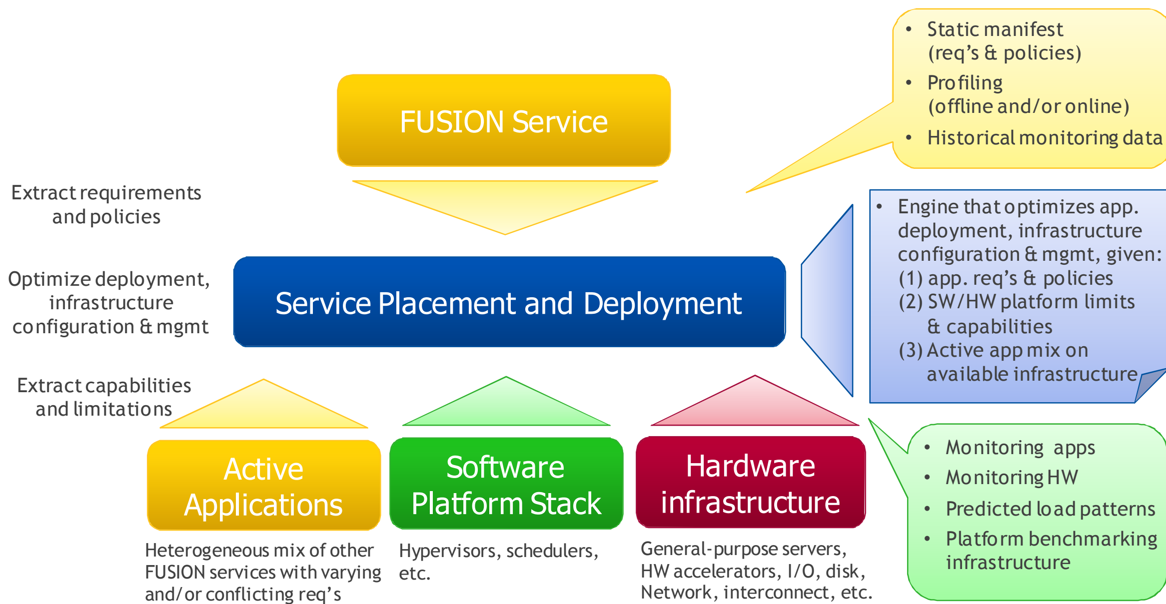

Figure 43: Optimizing service placement and deployment

Within the FUSION project, we will develop algorithms and heuristics for optimizing the placement and deployment of FUSION services by taking into account these aspects, both at a coarse-grain level across execution zones, as well as at a fine-grain level within a FUSION execution zone and potentially also within a FUSION execution point.

6.3.4 Monitoring

In this section, we will discuss both resource as well as service monitoring in the context of FUSION, where demanding interactive services are deployed in a distributed manner relying on various types of virtualization and hardware resources.

The goal of monitoring in FUSION is to capture the impact of FUSION services when being deployed on HW architectures (CPU load, memory bandwidth consumption, CPU cache behaviour, etc.), and vice-versa, to also capture the impact of a particular HW architecture onto these services. To enable this, we need to establish a set of metrics to be able to measure the overall operational health of both services as well as the infrastructure and platform. Furthermore, the monitoring information will also be key input with respect to the FUSION orchestration layer during service placement, load balancing, etc. Different possibilities are being evaluated at the moment of writing of this document.

In this section, we will discuss the various elements and pieces that will be studied in the FUSION project. Basically, we see three types of monitoring functions, namely probes, aggregators and reporting functions. We will now briefly discuss the role of each function:

6.3.4.1 Probing

The goal of probing is to profile and measure various types of runtime information concerning services or resources. These probes not only provide input towards health monitoring services, but also enable characterizing a particular type of service in terms of resource allocation and utilization, which is of key importance for the orchestration layer when deploying, planning and/or mapping a particular service instance onto a particular execution zone and physical host.

In FUSION, we will study what probes and resource metrics are necessary for particular types of services to be able to characterize and effectively monitor these services. This relates to the use case requirements we presented earlier.

6.3.4.2 Aggregation

The goal of the monitor aggregation services is to gather input from all relevant probes in a configurable or programmable manner to be able to generate a set of combined performance indicators. In other words, they collect the lower-level monitoring data and produce higher-level monitoring data based on for example a configuration script.

In the FUSION project, we will establish a set of relevant performance indicators for some of the use cases to evaluate the effectiveness of our approach. It is a topic of research how this configurability and programmability can occur in order to provide meaningful aggregation.

6.3.4.3 Reporting

The goal of reporting is to forward and distribute the aggregated monitoring information to other services in FUSION so that these services can exploit the captured and aggregated monitoring information. These target services can be either FUSION core services or other application services.

We currently consider three main reporting models: (i) a hierarchical propagation model, (ii) a distributed propagation model and (iii) a hybrid model whereby inter-zone and inter-domain reporting occurs in a distributed or hierarchical manner and a hierarchical model within a platform or execution zone.

Which information will be reported and its periodicity is one of the research aspects of FUSION. Another key question is whether a FUSION entity should also report on the actual resource usage of a service/platform/zone/domain, or whether it is more beneficial to report in terms of capacity and availability, for example towards orchestration. Does one report on specific entities or does one provide aggregated information? Other questions that we will focus on in more detail include how cross-zone services are monitored, aggregated and reported, and who is responsible for handle this information.

6.3.4.4 Service and Resource Monitoring Architecture

As an example, we display a possible service and resource monitoring stack for a single physical host in an execution zone in Figure 44 below.

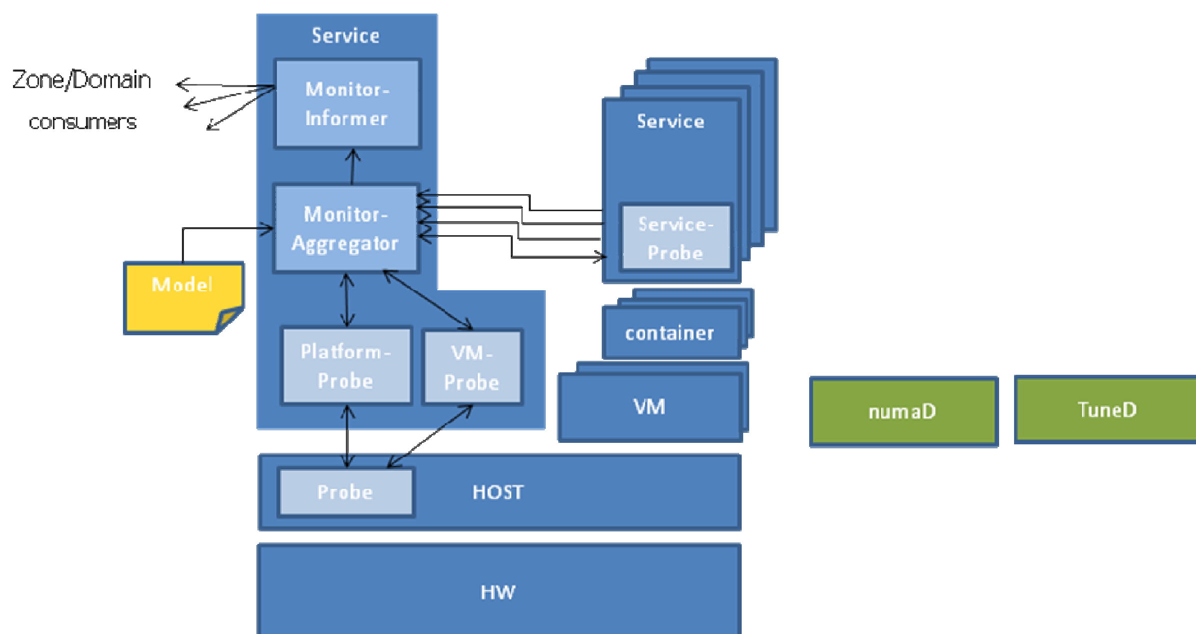

Figure 44: FUSION Service and Resource Monitoring Architecture

Copyright © FUSION Consortium, January 2014

We identified the following key components:

- ***Service probe***: measuring application/service specific metrics
- ***VM Probe***: measuring VM resource specific metrics
- ***Resource probe***: measuring HW platform specific metrics
- ***Resource aggregator***: collecting and aggregating all resource monitoring data
- ***Host aggregator***: collecting and aggregating the monitoring data from all deployed service instances and resource utilization
- ***Host reporter (or informer)***: deals with the distributed aspects of monitoring and the sharing of monitoring information (including privacy and policy aspects)

6.3.5 Data centre abstraction layer

We envision that a FUSION execution zone initially would be deployed on top of an existing data centre management system and depend on the underlying layer for managing the services and the hardware resources. Although this is not a strict requirement, it does simplify the deployment of FUSION execution zones in existing data centres, at the expense of reduced flexibility control as the zone manager may or may not have direct access or control to the underlying layers. It does however allow to very quickly enable parts of existing data centres for running FUSION services without immediately having to integrate all FUSION functionality into the existing management platform or replacing it by a native FUSION environment. It also means that the lowest layers of service and resource management do not have to be implemented from scratch, but that we can rely on existing proven management systems. In the long run however, we expect a closer integration of the key FUSION APIs into some of the existing cloud management APIs. For example, the OpenStack APIs could be extended with some of the key FUSION APIs and capabilities for enabling the required FUSION functionality. Data centres supporting these OpenStack APIs could then very easily be transformed into native FUSION execution zones.

In this section, we discuss some of the key functions of this DC abstraction layer. The key role of this service is to act as a mediator between the northbound zone orchestration services and the southbound data centre management platform. It needs to understand all requests coming from the northbound APIs and translate them into the appropriate southbound APIs, and vice versa. In the extreme case, the DC abstraction layer can also have direct access and control over the bare hardware infrastructure.

An overview of the key northbound functions of the FUSION DC abstraction layer are listed below. These will be worked out in more detail during the project:

- Encapsulation interface

  This interface provides a number of functions for creating DC-specific environments for deploying FUSION services. This can range from native environments for which physical resources need to be allocated, towards preparing VM images for inserting and deploying them in a data centre.

- Lifecycle management interface

  This is the overall interface for deploying and managing the lifecycle of the enclosing environment in which FUSION services are running. This includes starting and stopping these environments on the data centre infrastructure.

- Monitoring interface

Copyright © FUSION Consortium, January 2014

We envision a monitoring interface that enables querying the underlying data centre for resource and service monitoring information. We also foresee the possibility of a zone manager to be able to deploy specific monitoring probes into a data centre for measuring specific information.

- Administrative interface

  This interface contains several administrative functions, including accounting and billing, authentication and authorization, etc.

6.3.6 Service gateway

The service gateway is the intermediary between an execution zone and the FUSION service routing layer. Service requests that have been routed to the zone are forwarded internally to one of the running instances. This forwarding can be based on:

- A FUSION identifier: e.g., a stateful service may have its own identifier
- Load-balancing rules: the zone manager may set load balancing rules for the service gateway.

The functionality of the service gateway is detailed in Deliverable 4.1.

6.3.7 Light-weight virtualization and deployment

In FUSION, we envision light-weight deployment and virtualization models for quickly and efficiently deploying and running FUSION service instances within execution zones and execution points. As FUSION application services may need to be deployed on demand in a distributed execution zone, we need efficient ways for fetching the software packages and for provisioning and deploying new instances. Within the FUSION project, we plan to investigate the utilization of light-weight containers in combination with fine-grained resource control to facilitate these goals.

To measure the scalability and overhead of light-weight containers, we created an initial series of experiments, using the low-level capabilities of Linux for providing fine-grained isolation of the application environment as well as the resources, by leveraging the Linux namespaces, cgroups, chroot and by exploiting the shared mounting and binding functionalities in an efficient manner. We added very simple package management and state management capabilities as well as a management interface for quickly creating new containers or *nooks*, deploying a particular software package in the nook and starting/pausing/stopping the service running in the nook. A container or nook can be regarded as a FUSION execution point within which FUSION services can be deployed and managed.

In Figure 45, we show the overhead for the three main operations (and their respective reciprocal functions) needed to get a FUSION service up and running:

- NookAdd: creating a new execution point (here called a container or nook) for deploying a services.
- BundleIn: installing the software packages inside the environment; obviously, this depends on complexity of the service; in some cases, the software packages may be preinstalled and preconfigured along with the creation of the container;
- NookStart: starting the application service contained in the nook, as specified in a manifest.

In the graph, we show the amount of time each operation takes for adding/removing or starting/stopping the *N*th nook. As can be observed, deploying, starting and stopping the simple test services inside the nook only takes a fraction of a second, even when thousands of these services are active on the same physical system (note that these test service are not compute-intensive but mainly serve as dummy services for evaluating the system). The time for creating and destroying nooks (i.e., execution points or containers) seems to increase linearly with the amount of nooks that

Copyright © FUSION Consortium, January 2014

are already on the system. This is due to the overhead of the Linux mount tables that we use intensively for efficiently sharing software libraries and environments across containers, but which seems to scale linearly with the number of entries in the mount tables. However, even with a thousand active nooks, the overhead for creating another one still only takes roughly two seconds.

Note that due to the light-weight isolation, no memory needs to be reserved or partitioned explicitly as with full system virtualization methods. Also, as there is only one active operating system, a lot of duplication in kernel structures, as well as system software is avoided, saving large amount of memory in case of many light-weight services.

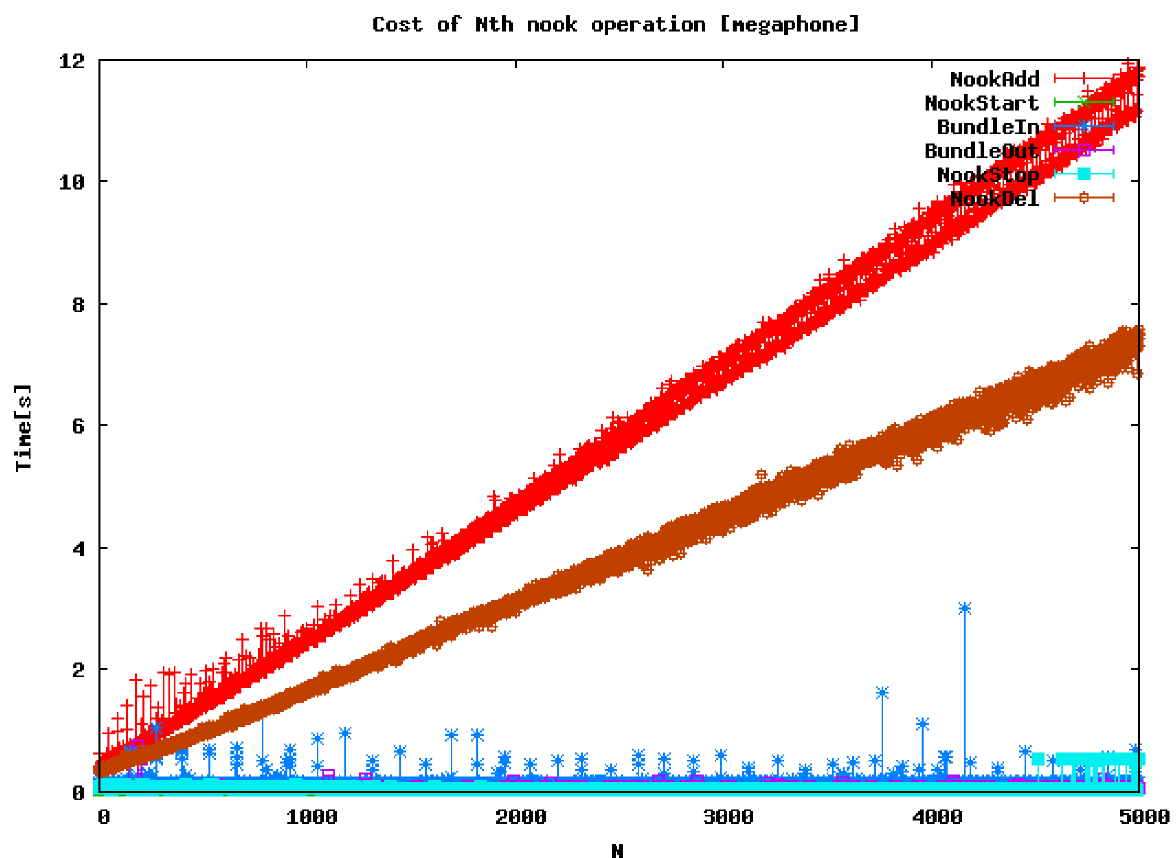

Figure 45: Overhead of light-weight container deployment

In Figure 46, we show an update version of our simple container-based management system, where we partially optimized the initial implementation. As can be observed, the overhead of creating a new execution point or nook is significantly reduced, allowing to create several thousands of light-weight environments, each taking at best a handful of seconds. Although we do not intend to create that many execution points in parallel on one physical machine, it does show the flexibility and speed of setting up or removing such an environment, which can significantly reduce the start-up latency in case of on-demand or on-standby service deployment scenarios.

Copyright © FUSION Consortium, January 2014

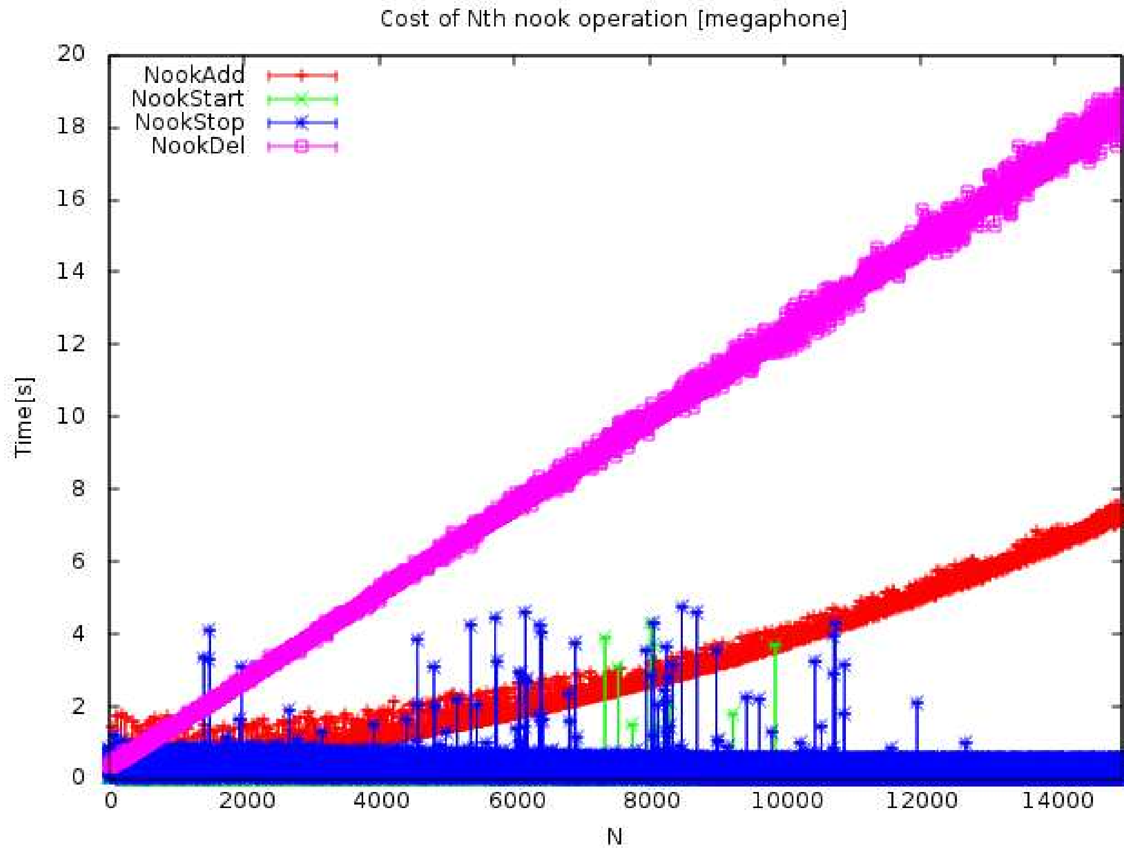

Figure 46: Update on overhead of light-weight container deployment

6.4 Management interfaces

The key management interfaces of a FUSION zone manager, who is responsible for managing FUSION services and execution resources, can also be divided into four key interfaces, as is depicted in Figure 47:

- An orchestration management interface
- A routing and networking management interface
- A service management interface
- A data centre management interface

Copyright © FUSION Consortium, January 2014

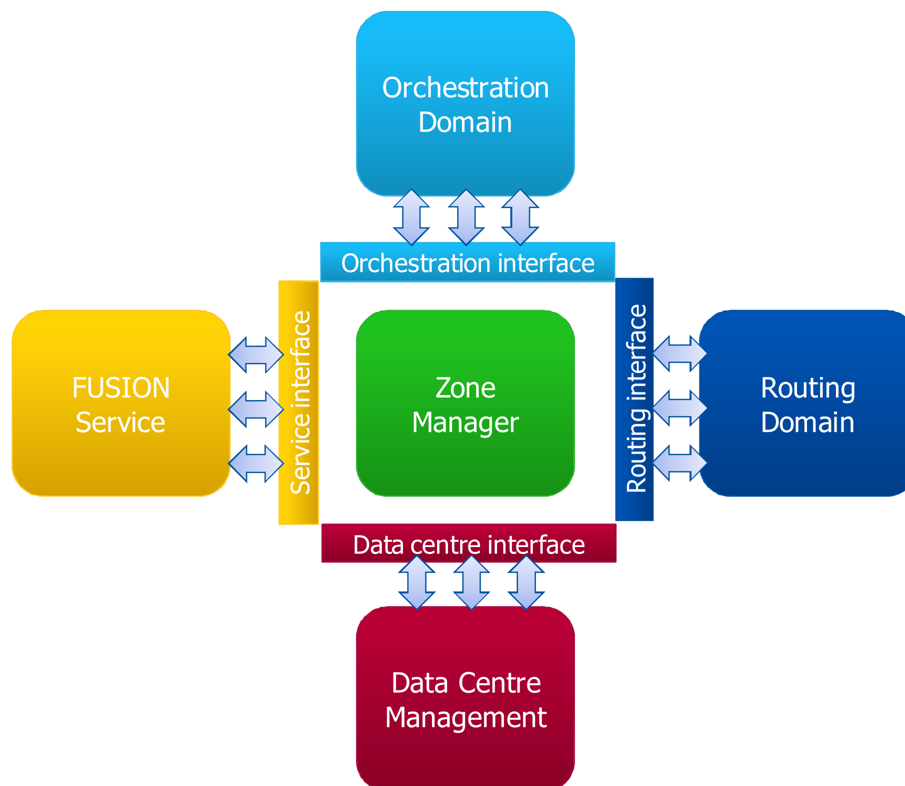

Figure 47: Key FUSION zone manager interfaces.

In the following sections, we zoom in on each interface and discuss the initial set of key functions per interface. In this section, we only describe the key properties and expected behaviour of the interface functions; it is up to the zone manager to decide how to implement this function. Later, we will elaborate on how this can be implemented by the functional design we described earlier.

6.4.1 Orchestration management interface

The orchestration management interface handles all direct communication in between an orchestration domain and a zone manager. Three sets of key functions that belong to this interface include the service pre-deployment and corresponding zone selection, overall service lifecycle management and exchanging service and resource monitoring information.

6.4.1.1 Zone selection

In the four-step placement approach, the execution zones and corresponding zone managers are actively involved in selecting an appropriate execution zone for deploying a number of instances.

Copyright © FUSION Consortium, January 2014

Zone Selection Interface		
FUNCTION NAME		
FUSIONZoneEvaluate (domain orchestration → zone manager)		
BEHAVIOR		
This function allows an orchestration domain to request a zone manager to evaluate the request for deploying a number of instances of a particular service type inside the zone managed by a zone manager.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	URI	Name of the service
ServiceParameters	Blob	Deployment parameters for the service
DeployParameters	Structured	FUSION-aware deployment parameters for the zone
Timeout	Time	Maximum amount of time for making the evaluation
RETURN VALUES		
Name	**Type**	**Description**
Score	KeyValueList	The multidimensional score

6.4.1.2 Service lifecycle management

These service lifecycle management functions are high-level management functions coming from the orchestration domain to trigger an execution zone to create, destroy or manage service instances.

Copyright © FUSION Consortium, January 2014

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONDeployService (domain orchestration → zone manager)		
BEHAVIOR		
An orchestration domain can use this function for explicitly deploying a number of instances in the execution zone.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	URI	Name of the service
ServiceParameters	Blob	Deployment parameters for the service
DeployParameters	Structured	FUSION-aware deployment parameters
Timeout	Time	Maximum amount of time for deploying the services
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
InstanceIDs	[FUSIONID]	List of the service instance IDs of the newly created instances

Service Lifecycle Management Interface		
FUNCTION NAME		
FUSIONTerminateServiceInstance (domain orchestration → zone manager)		
BEHAVIOR		
With this function, the orchestration domain can explicitly terminate particular instances. Note that alternative functions will also be possible: reducing the available sessions, etc.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
InstanceID	URI	Name of the service
Timeout	Time	Maximum amount of time for terminating the sessions
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

6.4.1.3 Monitoring

With these functions, an orchestration domain can explicitly query an execution zone for the status of particular services and the availability of the execution zone resources.

Copyright © FUSION Consortium, January 2014

Service Monitoring Interface		
FUNCTION NAME		
FUSIONGetServiceInformation (domain orchestration → zone manager)		
BEHAVIOR		
Retrieve aggregated monitoring information for a particular service type		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	FUSIONID	Name of the service
Category	Enum	Specific subset of information
RETURN VALUES		
Name	**Type**	**Description**
Information	Structured	The requested information

Resource Monitoring Interface		
FUNCTION NAME		
FUSIONGetZoneInformation (domain orchestration → zone manager)		
BEHAVIOR		
Retrieve aggregated available information about the execution zone		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Category	Enum	Specific subset of information
RETURN VALUES		
Name	**Type**	**Description**
Information	Structured	The requested information

6.4.2 Routing management interface

This interface between the Zone gateway and the FUSION Service Routers is discussed in Deliverable 4.1.

6.4.3 Service management interface

This section covers how a FUSION service can interact with a zone manager.

6.4.3.1 Service lifecycle management

These functions have already been described in Section 4.5.1.

Copyright © FUSION Consortium, January 2014

6.4.3.2 Service monitoring

With this function, a zone manager can explicitly request particular monitoring and health information for a service instance.

Service Monitoring Interface		
FUNCTION NAME		
FUSIONGetServiceInformation (zone manager → FUSION service)		
BEHAVIOR		
Request particular status information from a service instance		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Category	Enum	Specific subset of information
RETURN VALUES		
Name	**Type**	**Description**
Information	Structured	The requested information

6.4.4 Data centre management interface

This section covers the bidirectional interface between a zone manager and its underlying DC abstraction layer for managing services and resources.

6.4.4.1 Service DC environment

The data centre abstraction layer is responsible for managing an encapsulating environment in which FUSION services can be deployed and managed by a zone manager.

Copyright © FUSION Consortium, January 2014

Data Centre Management Interface		
FUNCTION NAME		
FUSIONDCPrepareServiceEnvironment (zone manager → DC agent)		
BEHAVIOR		
This function enables a zone manager to create and prepare an environment for a particular service to be deployed. This can range from physical to virtualized environments, consisting of a particular amount of resources that are allocated for that service.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
ServiceName	FUSIONID	Name of the service
EnvParameters	Structured	Environment parameters (type, resources, VM location, etc.)
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
EnvironmentID	FUSIONID	Reference to the newly created DC environment

Data Centre Management Interface		
FUNCTION NAME		
FUSIONDCResetServiceEnvironment (zone manager → DC agent)		
BEHAVIOR		
This function enables a zone manager to reset an existing service environment, allowing to quickly recycle the environment for another service deployment.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
EnvironmentID	FUSIONID	Reference to the environment
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

Copyright © FUSION Consortium, January 2014

Data Centre Management Interface		
FUNCTION NAME		
FUSIONDCTerminateServiceEnvironment (zone manager → DC agent)		
BEHAVIOR		
This function enables a zone manager to clean up and terminate the enclosing environment that was used for hosting a FUSION service. It is assumed that the FUSION service was already stopped beforehand.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
EnvironmentID	FUSIONID	Reference to the environment
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

6.4.4.2 Monitoring

Being able to monitor both the individual service environments as well as the overall DC resources is crucial for guaranteeing a proper operation of an execution zone to be able to host FUSION services with the appropriate QoS levels. This includes both enabling specific monitoring tools as well as retrieving the resulting monitoring information from the underlying system.

Data Centre Monitoring Interface		
FUNCTION NAME		
FUSIONDCGetInformation (zone manager → DC agent)		
BEHAVIOR		
Request specific monitoring information for the underlying data centre.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
Category	Enum	Specific subset of information
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
Information	Structured	The requested information

Copyright © FUSION Consortium, January 2014

Data Centre Monitoring Interface		
FUNCTION NAME		
FUSIONDCGetEnvironmentInformation (zone manager → DC agent)		
BEHAVIOR		
Request specific monitoring information for a particular service environment.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
EnvironmentID	FUSIONID	Reference to the environment
Category	Enum	Specific subset of information
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
Information	Structured	The requested information

Data Centre Monitoring Interface		
FUNCTION NAME		
FUSIONDCEnvironmentEnableProbe (zone manager → DC agent)		
BEHAVIOR		
Enable a particular monitoring probe for a specific FUSION service environment.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
EnvironmentID	FUSIONID	Reference to the environment
Probe	Enum	Select a particular type of probe
ProbeParameters	Structured	Configuration parameters for the probe
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code
ProbeID	FUSIONID	Reference to the newly created probe

Copyright © FUSION Consortium, January 2014

Data Centre Monitoring Interface		
FUNCTION NAME		
FUSIONDCEnvironmentDisableProbe (zone manager → DC agent)		
BEHAVIOR		
Disable a particular monitoring probe for a specific FUSION service environment.		
PROPERTIES & PARAMETERS		
Name	**Type**	**Description**
EnvironmentID	FUSIONID	Reference to the environment
ProbeID	FUSIONID	Reference to a particular type of probe
RETURN VALUES		
Name	**Type**	**Description**
Status	Int	Return code

6.4.4.3 Security interface

The security aspects of all these interfaces and components will be tackled in Deliverable D3.2.

Copyright © FUSION Consortium, January 2014

7. INITIAL SERVICE PROVISIONING ALGORITHMS

7.1 Usage patterns

The orchestration layer is responsible for the deployment of service instances across execution zones. Deploying an instance comes at a cost. It is therefore important to take into account the forecasted service usage patterns.

Energy consumption imposes a significant cost for data centres, but much of that energy is used to maintain excess service capacity during periods of predictably low load. On the other hand, there are also significant costs to dynamically adjust the number of active servers. These costs come in terms of the engineering challenges, as well as the latency, energy and wear-and-tear costs of the actual 'switching' operation. The value of dynamic resizing is highly dependent on statistics of the workload process. [WLCW12]

There are two factors of the workload that provide dynamic resizing potential savings:

1) Non-stationarities at slow time-scale, e.g. diurnal workload variations, peak-to-mean ratio

2) Stochastic variabilities at a fast time-scale, e.g. the burstiness of request arrivals

While one might expect that increased burstiness provides increased opportunities for dynamic resizing, it turns out the burstiness at the fast time-scale actually reduces the potential cost savings achievable via dynamic resizing. The reason is that dynamic resizing necessarily happens at the slow time-scale, and so the increased burstiness at the fast time-scale actually results in the SLA constraint requiring more servers at the slow time-scale due to the possibility of a large burst occurring. Second, it turns out the impact of the SLA can be quite different depending on whether the arrival process is heavy- or light-tailed. In particular, as the SLA becomes more strict, the cost savings possible via dynamic resizing under heavy-tailed arrivals decreases quickly; however, the cost savings possible via dynamic resizing under light-tailed workloads is unchanged.

7.1.1 Service popularity

The popularity of a service can be defined as the fraction of all user requests during a given time frame. Historically, the popularity distribution of items has often been modelled to follow a power law: when the items are ranked according to their popularity, the number of requests for the *n*-th item is proportional to $\frac{1}{n^\alpha}$. In a strict sense, only when α = 1, the power law can also be referred to as Zipf's law. In many cases, researchers apply however the term 'Zipf law' for coefficients α smaller than, but close to 1. If α < 1, the distribution has an infinite tail. The closer α is to 1 (while still being smaller than 1), the more heavy-tailed the distribution is. This means that there are many items with almost equal (low) popularity at the tail of the distribution.

Table 1 provides an overview of the values for the parameter α that have been discovered by other researchers, for various content and service types. Most work has focused on the popularity of content, such as viewing statistics of user-generated videos on YouTube, broadcast TV channels or torrents. Considerably less research is available with measurements of the popularity of web-based services. Besides web service popularity, we have added numbers on the distribution of the number of mobile app downloads to Table 1.

Table 1 Distribution of various content and service types.

Service/Content type	Distribution [parameters]	Long tail observed?	Reference
User-generated content • YouTube	Zipf [0.8]	yes	[CKRA07]

• DailyMotion	Zipf [0.56] Gamma [k=0.51, lambda=6700] Weibull [k=0.44, theta = 23300) Zipf [0.88]	yes yes yes	[GALM07] [CDL07] [CDL07] [CKOS11]
Video On Demand	Zipf [0.65 – 1] Zipf [0.5] for top 100; then Zipf [1.2]	yes yes	[YZZZ06] [CKOS11]
Live Event Broadcast • TV channel • user generated	2-mode Zipf [coefficients not reported] lognormal [μ = 1.03; σ = 1.045]	yes yes	[SCC13] [SAG11]
Torrents • Demonoid • PirateBay	Zipf [0.82] Zipf [0.75]	no not reported	[FRSS12] [FRSS12]
Web services	Zipf for top 318	yes	[JTPS13]
Mobile apps downloads • iPad • iPhone	Zipf [0.903] Zipf [0.944]	not reported not reported	[GT12] [GT12]

As shown in Table 1, many researchers have indeed discovered a power law distribution with a coefficient depending on the actual content or service type being requested, but in general between 0.6 and 0.9.

Nevertheless, in many cases the power law overestimates the popularity of lower ranked items. This effect is referred to as the 'long tail': there are many items with very low popularity. The effect is better observed in large collections with items that are very specialized or only of local interest. For example in YouTube, many videos will only be of interest to friends of the uploader, will be in a particular language or treat a very specialized subject that is only of interest to a niche group.

The long tail effect has been observed by many researchers: after an initial top-X of the most popular videos, the tail decreases tremendously. A more accurate modeling is therefore to use the concatenation of two Zipf curves (2-mode Zipf [CKOS11, SCC13]), a log-normal distribution [SAG11] or Weibull and Gamma distributions [CDL07].

The main conclusion to draw is that the long or heavy-tail of service popularity could be a strong challenger for the scalability of FUSION. A rapid, on-demand service instantiation mechanism will be needed.

7.1.2 Geographic locality

Service popularity may vary with the geographical location of the users accessing these services. The underlying reasons can be cultural, linguistic, or sharing with friends and family which in a large majority live close to each other on a global scale.

In [SDJS10], the sharing of data between geographically distributed clients was observed in traces of the Live Messenger and Live Mesh applications. Live Messenger is an instant messaging application, and Live Mesh provides a number of communication and collaboration features, such as file sharing and synchronization. The traces covered all users and devices worldwide that accessed the service. To estimate the client location, the IP-address was mapped on geographical coordinates by the commercial Quova IP Geo-Location database. The authors have calculated for each observed item

Copyright © FUSION Consortium, January 2014

the centroid on the globe and analyzed the distance of each user to this centroid. The results are presented in Figure 48.

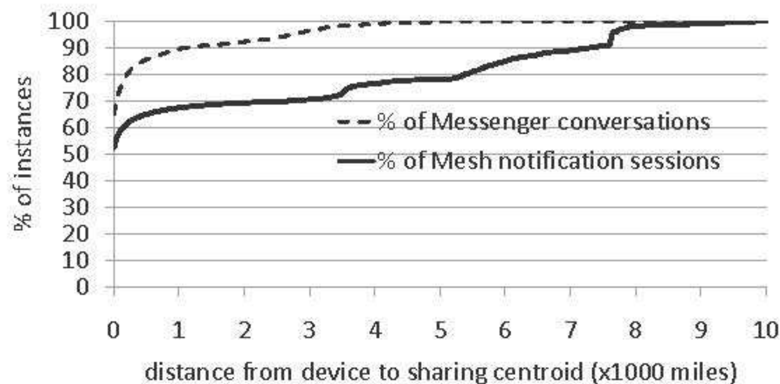

Figure 48: Sharing of data between geographically distributed clients [SDJS10]

As can be derived from this graph, more than 50 % of the items have a distance of 0, meaning that the items are not shared, or are shared between users whose IP addresses map to the same geographic location. Owing to the nature of the analyzed service, it is perhaps not surprising that large amounts of sharing happens between very distant clients, such as corporate branches in different nations. The authors suggest placing data closest to those who use it most heavily, rather than just placing it close to some particular client that accesses the data.

In [KCLC13], the authors analyze 2 billion geo-tagged tweets. Dividing the globe into squares of 10 km by 10 km, the similarity of hashtag sharing between two regions drops quickly with the distance. Specifically, around 50% of hashtags derive at least 50% of their postings from a single location.

In [HKKT12], the authors highlight that the views of niche videos on YouTube (with less than 1000 views) are geographically highly concentrated: 80% of all views of these niche videos come from less than 15 countries. This concentration effect remains relatively strong for all videos up to 100M views and gradually fades out as the popularity of videos increases, to reach an almost uniform global distribution over all countries for extremely popular videos (more than 10^7 views).

The main conclusion to draw is that a geographic diversity in content consumption has been confirmed by many authors. Further research is required, but one might expect the same might be true for services, e.g. based on cultural or linguistic preferences, social media sharing, etc.

7.1.3 Variations over time

YouTube video popularity evolves over time, exhibiting a peak of interest which then fades away [BSW12]. Videos often experience a peak in their number of views when they are featured on popular websites or when they become viral and spread on online social networks. An important consideration is that the spatial properties of the views also change in time. On average, a video tends to become popular in a single region ('focus region'), and receives only a few views from other regions. Immediately after the peak, the fraction of views from the focus region goes down, and interest shifts to other locations. Finally, view traffic shifts back to the focus location. Social networks have are an important driving factor behind this pattern.

On shorter terms, the expected diurnal patterns have been observed for VOD in [YZZZ06]. Within one single day, the number of users drops gradually during the early morning (12 AM – 7 AM) and the afternoon (2PM – 5PM), while it climbs up to a peak when users are in noon break or after work (6PM-9PM). On a weekly basis, the authors observe the highest peaks in the weekend.

Copyright © FUSION Consortium, January 2014

The popularity distribution versus age was also studied by [CKRA07]. Excluding the very new videos, user's preference seems to be relatively insensitive to video's age. While user's interests is video-age insensitive on a gross scale, the videos that are requested the most on any given day seem to be recent ones. A revival-of-the-dead effect, where old videos are suddenly brought up to the top of the chart, was not strongly observed by the authors.

The popularity of blog articles steadily drops [JKHS12] over time. However, external references by other bloggers may incur a sudden surge of the number of views, even if the reference article is a few days old.

Strong diurnal effects in the number of arrivals per unit of time have been observed in Hotmail, a large e-mail service running tens of thousands of servers. The authors in [WLCW12] used traces from 8 such servers over a 48-hour period. The peak-to-mean ratio is 1.64.

In [SCC13], they observed points of big rises and falls with respect to the number of requests for live streamed sports events. There are also some unexpected bursts that may occur during the course of the game due to some unexpected game activities that develop. An example is given of a sudden increase in the number of Taiwanese viewers for an American sports event because one Taiwanese player had been in the center of public attention in his domestic country.

Conclusions:

- diurnal patterns should be taken into account to predict demand
- in general, popularity decreases over time (except for the most popular services)
- the effect of external references (social media recommendation, reviews, blogs) should not be underestimated

7.1.4 Session length (Service holding time)

Each user entering the system will hold the allocated resources for some time. The expected holding time per service session determines the time window that should be taken into account when allocating resources for a new request, and also reflects the pace at which new network paths may need to be configured (if resource allocation is done on a per-session basis).

Multimedia services, which are the main focus of FUSION, are typically characterized by a longer holding time: users watch TV for a longer time, or are continuously connected to the cloud for the analysis of their video stream. In FUSION, we specifically target long-lived multimedia sessions. The session duration for MMORPG was monitored in [SDM09]. The mean session duration value is about 55 minutes. Interestingly, about 24% of the sessions lasted 2 minutes or less, while the longest session lasted almost 10 hours. A high percentage of short sessions can be explained by the phenomenon of alts: alternative or secondary characters that are used as an extension of the main character's capacity to circumvent the game limitations.

The authors in [BRC09] evaluated the usage pattern of four online social network sites. All four OSN sites exhibit a consistent heavy-tailed pattern in their session duration, however the media session durations vary across OSNs, from 3 seconds up to 13.4 minutes.

Also the session length for videos in a VOD system is quite short: 52.6 % of all requests have less than 10 minutes of session length, and 70 % of sessions are terminated in the first 20 minutes [YZZZ06]. The dominance of short videos is confirmed in [VPG05], showing that 20-40% of a movie session is terminated in less than 10 minutes. In [CSK11], the holding time distribution indicates that the mean holding time is 22.50 minutes.

The authors in [SAG11] suggest a lognormal distribution for the duration of viewing session to live streaming of user-generated content. The authors evaluate both short (< 20 min) and long viewing

Copyright © FUSION Consortium, January 2014

sessions and observe in both cases that the distribution of the viewing time is skewed towards very short durations: around 70% of all sessions have durations under 200 seconds in both classes.

Since watching video requires the user's full attention, this may impact the session length distribution. Listening to music usually does not require constant attention from users. The average duration of a listening session in Spotify is varying with the time of day. The length of sessions started on a desktop peaks in the morning (400 min) and decreases almost monotonously until late night (100 min). The authors attribute this effect to the users that launch Spotify to have background music at work. On the other hand, mobile sessions are much shorter than desktop sessions. Although the number of mobile sessions is much larger than the number of desktop sessions, the media mobile session length exhibits small variation over time (10-20 min). This suggests that the usage pattern is dramatically different between desktop and mobile users.

Conclusions:

- session time can be highly variable, and application-specific
- session time tends to be short
- mobile may even result in shorter sessions (e.g. while commuting, quick checking, …)

7.1.5 User arrival rate

The strong daily patterns lead to significant variations in the arrival rate of users. Typically, the session arrival rate will be lower at night, and peak during the day. Moreover, for example in Spotify, the authors observe that the arrival rate peaks for mobile sessions peaks one hour before the desktop sessions, owing to the commuting of its users [ZKI13]. The authors propose to model the arrival rate as a non-homogenous Poisson process, which is a Poisson process with its rate parameter λ changing over time. The expected number of events between time *a* and time *b* is:

$\lambda_{a,b} = \int_a^b \lambda(t)\, dt$. The authors model constant values of λ in 10-min intervals. The percentage of exponential interarrivals and independent sessions arrivals is above 95 % for both the desktop and mobile datasets of all countries observed.

The session arrival time during live streaming of user-generated content can be fitted by a uniform curve, despite a slight tendency for users to arrive closer to the beginning of a transmission [SAG11].

In [CSK11], the authors show a good consistency between their measurements and a Poisson distribution, for both VOD and VOIP calls. The average arrival rate for VOD is 14.09 requests per second and 26.27 connections per second for VOIP calls.

In [CKRA07], the authors' measurement traces only indicate the number of requests for a video per day. Assuming that requests are exponentially distributed over the day, they conclude that roughly 95% of the videos are requested once every 10 minutes or longer.

The main conclusion is that user demand can be modeled as a non-homogeneous Poisson process, with the parameter lambda following diurnal patterns and perhaps with its mean decreasing over time.

7.2 Inter-zone service placement

7.2.1 Introduction

This section presents a first modelling and simulation of the problem of server placement between execution zones, focusing on the issue of resource allocation within execution zones when there is competition for resources from multiple applications each being managed by an orchestrator. A detailed algorithm specification together with a set of evaluation results from simulations will be presented in deliverable D3.2.

Copyright © FUSION Consortium, January 2014

Execution zones can be deployed in various places throughout the Internet: in network-edge points of presence close to users; collocated with routers within an ISP's network; in local data-centres owned and operated by ISPs; and in traditional data-centres and service farms operated by cloud and service providers.

Unfortunately, some of the smaller execution zones will be unable to provide the essentially boundless elasticity that purpose-built data centres can provide. This means that, in many instances, there can be more requests for resources within any given execution zone than there are resources available. In these cases, resource allocation mechanisms that deal with service contention will be required. When securing resources from execution zones, ochestrators acting on behalf of applications will need to balance the benefit that these resources bring to the QoE of their users with the costs that they need to pay to the computation resource providers. We propose resource allocation algorithms to address efficiency, contention and oversubscription, and evaluate them through simulation.

7.2.2 Architecture

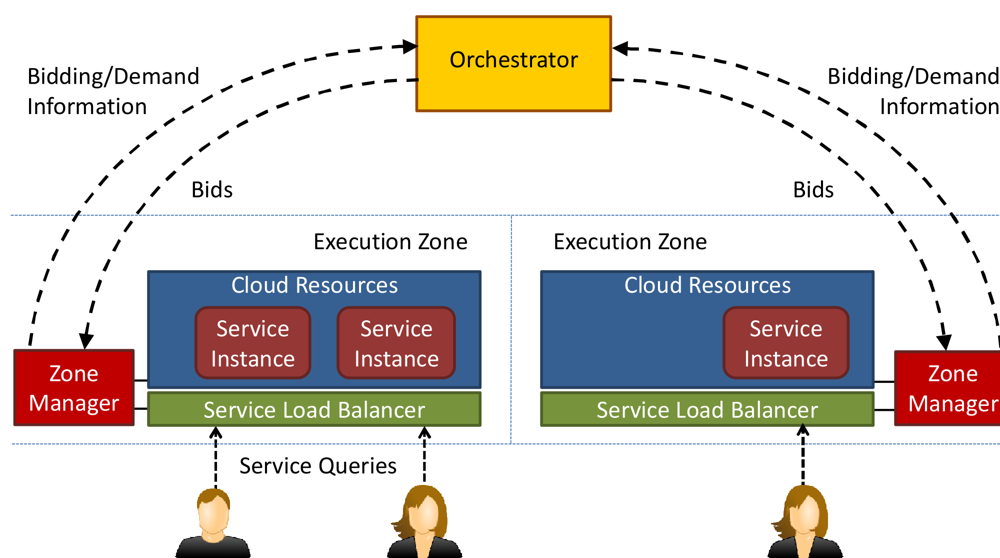

Figure 49: Service provisioning architecture for atomic services in the decentralized cloud

We assume the existence of a large number of execution zones (EZ), that is, cloud endpoints where computational capabilities are offered to applications. Resources in each EZ are managed by a *zone manager*. Applications can deploy *services* (i.e. self-contained application logic components) on these execution zones under the direction of a FUSION orchestrator that implements the appropriate cost-benefit tradeoffs for that application. The orchestrators acting on behalf of their applications then compete with one another for resources on the execution zones.

Rather than relying on a centralized resource allocator to grant orchestrators access to EZ resources, we allow each execution zone to individually perform resource allocation decisions. There is a rich literature on how to solve this problem [KRAU02] [LAND09] [WANG05]; in this section we propose the use of a market system to allow execution zones to allocate resources to those applications that value them the most (see [FU03] [GUPT02] [SHNE05] [WALD92]). Due to their price self-determination property, we use auctions to allocate resources to applications (see [CHUN05] [FELD05] [KARV06] [LEVI08] [STILL10] [ZAMA11] for other examples of auctions used for resource allocation in grid, peer-to-peer and cloud computing). In essence, our architecture allows each orchestrator to bid for resources on each execution zone according to its individual policies, and implementing its own tradeoff between user satisfaction and infrastructure cost.

We will focus our attention on the dynamic use of execution zone resources by orchestrators. To simplify the resource allocation tasks of execution zones, we will assume that each EZ will offer a discrete number of identical *service slots* which can be univocally assigned to a single application. In addition to its scalability benefits, this simplifies the resource modelling in execution zones and allows a more straightforward calculation of the tradeoffs involved.

With these definitions our proposed architecture can be explained as follows (see Figure 49). Application users generate *service queries*, which are directed to the appropriate execution zones by an appropriate mechanism (e.g. differentiated DNS resolution). The zone managers in those zones load balance these requests between the available service slots that the relevant application has deployed in the EZ. The application achieves this through its orchestrator, that sends bids to execution zones specifying the number of service slots desired and the bid amount that they are willing to pay per service slots. The zone manager then allocates its available service slots by running a multi-item Vickrey auction [VICK61]; the top bidding orchestrator bids are hence selected for deployment.

7.2.3 Problem definition

We assume that the orchestrator can determine, by external means, the total user demand associated with a given execution zone. The question becomes: given the bidding profile at every EZ, for how many instances of the service should the orchestrator bid, and at what price?

Each orchestrator n_i will bid for a number of service instances in each EZ, according to the estimated demand consumption and a predefined notion of service quality. The problem for each manager n_i is to determine the number of service instances m_i that it will bid for within execution zone i, as well as the bid price v_i that it must offer in order to ensure that it can satisfy its customer demand with probability π.

Assume that each execution zone n_i can offer up to M_i service instances to managers (see Table 2), and that each orchestrator b_j sends m_{ij} bids to n_i, each at a value v_{ij}. The execution zone receives these bids and ranks them in increasing order of value, and the top M_i win a resource unit. The payment that will be applied to each winning orchestrator b_j is the sum of the top X^{ij} losing bids,where X^{ij} is the number of bids that b_j won.

To allow a manager to estimate the expected outcome of placing a given number of bids at a given price with an execution zone, it needs to estimate the number of competing bids it will encounter. Thus, rather than the total number N_i of bids on n_i, we are interested in the number of bids in n_i with which the bids of b_j must compete – which involves subtracting from N_i those bids sent by b_j itself.

N_i	Total number of bids that execution zone n_i receives
N_i^j	Total number of bids that execution zone n_i receives that compete with those placed by b_j
M_i	Total number of resource units that execution zone n_i offers
m_{ij}	Total number of bids that orchestrator b_j sends to execution zone n_i
v_{ij}	Monetary value for each bid that orchestrator b_j sends to execution zone n_i
X^{ij}	Number of bids that b_j wins on n_i
$f^{V^i}(v_{ij})$	Probability density function of bid values received by auctioneer n_i
$F^{V^i}(v_{ij})$	Cumulative distribution function of bid values received by auctioneer n_i

Copyright © FUSION Consortium, January 2014

$f^{V^i}_{(k)}(v_{ij})$	Probability density function of the k-th order statistic of the bid values received by auctioneer n_i
$F^{V^i}_{(k)}(v_{ij})$	Cumulative distribution function of the k-th order statistic of the bid values received by auctioneer n_i

Table 2: Auction-based resource allocation model notation

A detailed problem formulation, mathematical modelling and algorithm description will be presented in deliverable D3.2.

7.2.4 Evaluation

We evaluate our proposed architecture through simulation. We built a custom simulator where execution zones and orchestrators are implemented as parallel threads. The simulation proceeds by epochs, with one epoch consisting of a *bidding phase* in which orchestrators decide, in parallel, how many bids to submit to each execution zone and at what valuation. These values are accumulated at each execution zone, and when all required values are available, the system moves on to the *allocation phase*. At this point, execution zones independently perform resource allocation on the available bids, and calculate the QoS experienced by the users of each service in their designated population. Statistics are then gathered, and made available to orchestrators so that they can adapt their bidding behaviour. We now further explain our simulation setup and results.

7.2.4.1 Simulator setup

Each application and zone manager in our simulator is implemented as an independent thread and triggered by means of a thread pool [GARG02]. Simulation progresses by discrete time steps denoted as *epochs*. Each epoch consists of two steps, each one implemented as a *map/reduce* operation; simulation state is synchronized after each *reduce* step. We now broadly explain the two main steps of each simulation epoch by focusing in turn on the execution zones and the orchestrators (see Figure 50). The first step, denoted the *execution step*, implements functionality to allow execution zones to receive bids from all orchestrators, choose the winning set of bids, and calculate the QoS experienced by application users. Each EZ can then report to each orchestrator the total number of bids N_i that it received, the total number of service slots M_i that it offers, and the distribution $F^{V^i}(v_{ij})$ of its received bid values.

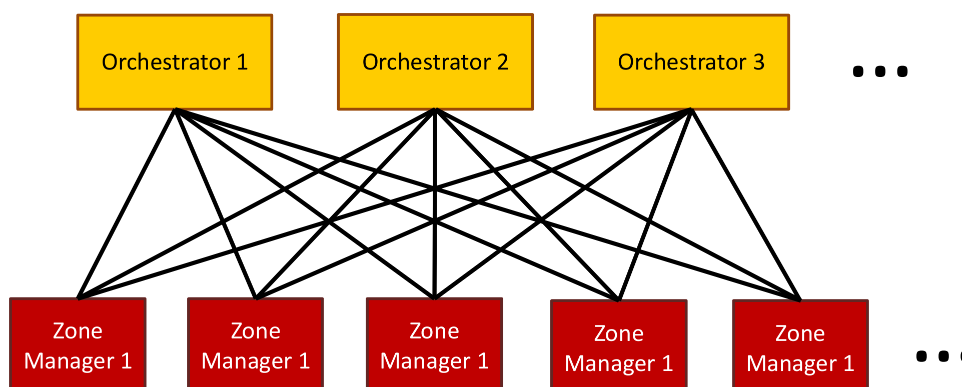

Figure 50: Simulator architecture

In order to capture the different incentives behind service deployment decisions, we consider three different kinds of orchestrators, which we denote as *QoS-sensitive*, *Price-sensitive* and *Background*. Whereas the first two aim to arrive at an optimal QoS/cost tradeoff, *Background* orchestrators

Copyright © FUSION Consortium, January 2014

generate bids and bid values as drawn from static probability distributions. Hence, *Background* helps us model additional execution zone load that is non-adaptive. Regarding *QoS-sensitive* and *Price-sensitive* orchestrators, the only difference is the chosen value of ζ. For *QoS-sensitive* orchestrators we use $\zeta = 60000$; for *Price-sensitive* orchestrators we use $\zeta = 10000$. These values are arbitrary and only chosen to differentiate between these two categories of orchestrators.

We parameterize our simulations in terms of the variables presented in Table 3. In order to reliably test the statistical properties of our simulation, we organize our simulations into *runs*, with each run consisting of a set of N_Replicas simulation replicas. For each one of these replicas we use the same values for al parameters and report the averages. For each simulation run, we choose a combination of individual values of N_EZ, N_App_QoS, N_App_Price, N_App_BG and N_Srv_Slots; the rest of the parameters are kept unchanged for all simulation runs. The values used for these variables are shown in the third column of Table 3. In order to represent the probability densities $F^{V^i}(v_{ij})$, we used histograms that split the entire range for v_{ij} into N_Hist_Bins equally-sized bins. Further simulation details and experimental results will be presented in deliverable D3.2.

N_EZ	Total number of execution zones	5
N_App_QoS	Total number of *QoS-sensitive* apps ($\zeta = 60000$)	10, 50
N_App_Price	Total number of *Price-sensitive* apps ($\zeta = 10000$)	10, 50
N_App_BG	Total number of *Background* apps	50, 100
N_Srv_Slots	Total number of service slots M_i in an execution zone	90, 180
Mu_Max Mu_Min	Maximum and minimum values for the rate at which a given execution zone can process user requests	2.5 1.5
Lambda _Max Lambda _Min	Maximum and minimum values for the rate at which user requests for a given application are generated at a given execution zone	1.5 0.5
N_Iterations	Maximum number of epochs for a given simulation replica	100
N_App_Threads	Number of orchestrator threads in the bid planning pool	450
N_EZ_Threads	Number of orchestrator threads in the execution pool	450
N_Replicas	Number of simulation replicas per simulation run	20
N_Hist_Bins	Number of histogram bins to represent $F^{V^i}(v_{ij})$	255

Table 3: Simulation parameters

7.2.5 Related work

There is a large body of work on auction-based resource allocation. An example of this is CompuP2P [GUPT02], a system that implements an open market for peer resources quantised into different markets. Each market is managed by a particular peer through a dynamic hash table (DHT). Pricing is arrived at by using a *Vickrey auction* [VICK61] as an information-revelation mechanism, and the system is designed to be robust in the presence of self-interested decisions of resource providers and users.

Another auction-based system is Spawn [WALD92], a distributed CPU resource allocation problem is solved by using an open market. Money in this virtual economy becomes an abstract form of priority, so that better funded processes can obtain correspondingly better access to the computing infrastructure than others. The system uses sealed bid, second price auctions, and each of the

Copyright © FUSION Consortium, January 2014

economic actors taking part in the economy for CPU cycles maintains a *resource manager* that manages auctions and assigns CPU time slices accordingly.

A shortcoming of typical combinatorial auctions is that they can be very computationally intensive, and thus might involve large delays [CRAM06]. In [FELD05], each participant allocates its finite budget to bid on a given resource set, and receives a proportion of each resource commensurate with the proportion of its bid with the bids of other participants; this same technique was later on proposed by [LEVI08] as a replacement for the unchoking policy of BitTorrent. Another auction-based resource management implementation is Mirage [CHUN05], where combinatorial auctions using a centralised virtual currency environment are used for sensor network testbed resource allocation.

A first set of server placement techniques has been developed in the context of data centres. Their key rationale is to minimise data centre energy consumption by maximising the utilisation of the running servers through efficient placement of the work load (in the form of virtual machines). Various decision parameters have been studied, such as the workload of the virtual machines or their memory consumption [JANG11] or the type of resources requested [STILL10]. However, these algorithms are not directly applicable to our problem definition, as in the data centre case, the effect of the underlying physical network topology is negligible: inside data centres, dedicated high-bandwidth and low-latency links can be used. Another contribution in this direction is [KARV06], where the authors propose a centralised service placement algorithm requiring information on the entire network topology. The algorithm maximises the satisfied demand and minimises the number of placement changes. In [ADAM07], the authors propose a decentralised variant of this algorithm. However, neither of these two algorithms takes into account latency or any other characteristics of the underlying physical network. A dynamic and latency-aware service placement algorithm for assigning resources to services is presented in [FAMA09]. The servers are connected via a peer-to-peer overlay technology. Each server is responsible for both taking part in the management tasks and running a subset of the available services. All the technologies discussed so far require a mapping of individual workloads (services) to a set of servers. Within the field of distributed systems, the replica placement problem has also been studied extensively, i.e. where to cache or replicate popular data objects [ZAMA11]. A model taking QoS into consideration is presented in [CHEN09]. Parameters like storage cost, update cost and access cost of data replication are used for the replica placement. Being NP-complete, two heuristic algorithms are presented for the model. For tree networks, two new policies for QoS-aware placement are presented in [BENO08].

Copyright © FUSION Consortium, January 2014

8. CONCLUSIONS

This document describes the initial requirements, design and interface protocols for realizing distributed service orchestration and management of demanding interactive composite service graphs across a heterogeneous set of execution zones. The scope of this deliverable thus describes in detail the initial work of the FUSION consortium regarding the orchestration, execution and service layers of the FUSION architecture.

In the first year of the project, we have been focussing on identifying the key high-level use case requirements and how these translate into lower-level requirements that impact the FUSION architecture and design decisions with respect to service description, orchestration as well as service execution management within a FUSION orchestration domain. In this process, we also focused on defining the overall boundaries and constraints regarding the flexibility and dynamicity of FUSION with respect to the targeted use cases.

Regarding the service layer, we started evaluating and describing in detail the various service graphs and how this may impact service instantiation and service selection, as well as defining initial requirements for describing these services to enable automated deployment and management by FUSION. We developed the concept of service sessions and session slots for efficiently and scalably managing service sessions across service instances. We also started investigating late binding techniques as a way for improving inter-service communication and identified an initial set of key service management interfaces for managing the lifecycle and heterogeneous resource utilization of FUSION application services.

At the FUSION domain orchestration layer, we created an initial high-level design, identified the key orchestration functions, and started working out sequence diagrams and interaction patterns for these functions. We also proposed a four-step placement strategy and corresponding evaluator services for managing service placement in a distributed and flexible manner, enabling to take into account very service-specific details on the one hand, and allowing the possibility of resource abstraction at the execution zone level on the other hand. We also started working on several other functions like service deployment and inter-domain orchestration, and provided an initial set of key interfaces towards the various components and actors that will interact with a FUSION domain orchestrator.

At the FUSION execution layer, we focussed on what the key functions of a zone manager are and how a FUSION execution zone can be deployed on top of execution data centre management platforms. We initially envision an overlay approach that allows most of the FUSION zone manager functionality to be deployed on different types of data centre management platforms using a data centre abstraction layer for translating FUSION service state and resource management and monitoring functions into whatever language the underlying infrastructure supports. Within the scope of the project, we also envision using light-weight virtualization and deployment techniques as well as support for heterogeneous hardware for efficiently deploying and running services on top of sets of heterogeneous FUSION execution zones. We also provided an initial set of key management interfaces for the zone manager for managing FUSION services on physical execution resources. Lastly, we provided initial algorithms regarding service provisioning. More specifically, we started identifying key usage patterns and how it may impact service scaling and deployment. We also described an auctioning algorithm for efficient resource utilization of execution zones with very limited resources.

In the second year of the project, we will refine the initial high-level designs and interfaces for the orchestration and execution layer and work out an initial design on top of an existing data centre management platform like OpenStack. We will continue working on the placement and deployment strategies of various composite service graphs and validate them using the selected use case scenarios and initial demonstrator setup as described in Deliverable D5.1.

Copyright © FUSION Consortium, January 2014

9. REFERENCES

[ADAM07] C. Adam, R. Stadler, Chunqiang Tang, M. Steinder, and M. Spreitzer. A Service Middleware that Scales in System Size and Applications. In Proc. of IFIP/IEEE IM (Intl. Symp. on Int. Net. Man.), pages 70–79, 2007.

[AM13] Abaakouk, M., Autoscaling with heat and Ceilometer, http://techs.enovance.com/5991/autoscaling-with-heat-and-ceilometer, August 2013.

[APP13a] Amazon AppStream, http://aws.amazon.com/appstream/, 2013.

[APP13b] Amazon AppStream, http://aws.amazon.com/appstream/, 2013.

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

[BENO08] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Replica Placement and Access Policies in Tree Networks. IEEE Trans. Parallel and Distrib. Syst. (TPDS), 19(12):1614–1627, 2008.

[BERT92] Dimitri Bertsekas and Robert Gallager. Data Networks. P. Hall, 2nd edition, 1992.

[BERT82] D.P. Bertsekas. Dynamic behavior of shortest path routing algorithms for communication networks. IEEE Trans. on Auto. Control, 27(1):60–74, 1982.

[BRC09] F. Benevenuto, T. Rodrigues, M. Cha, and V. Almeida, Characterizing user behavior in online social networks", Proc. of the 9th ACM SIGCOMM conference on Internet measurement conference, 2009

[BSW12] A. Brodersen, S. Scellato, M. Wattenhofer, YouTube around the world: Geographic popularity of videos, Proc. of the 21st international conference on world wide, web, 2012.

[CDL07] X. Cheng, C. Dale, and J. Liu, Understanding the characteristics of Internet short video streaming: YouTube as a case study, arXiv:0707.3670, 2007

[CEIL13] Ceilometer, http://docs.openstack.org/developer/ceilometer/architecture.html, 2013.

[CHEN09] Chieh-Wen Cheng, Jan-Jan Wu, and Pangfeng Liu. QoS-aware, access-efficient, and storage-efficient replica placement in grid environments. J. Supercomput., 49(1):42–63, July 2009.

[CHUN05] Brent N. Chun, Philip Buonadonna, Alvin AuYoung, Chaki Ng, David C. Parkes, Jeffrey Shneidman, Alex C. Snoeren, and Amin Vahdat. Mirage: A Microeconomic Resource Allocation System for SensorNet Testbeds. In Proc. of EmNetsII, 2005.

[CKOS11] Y. Carlinet, B. Kauffmann, P. Olivier, and A. Simonian, Trace-based analysis for caching multimedia services, Orange Labs technical report, 2011.

[CKRA07] M. Cha, H. Kwak, P. Rodriguez, Y. Ahn and S. Moon, I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system, Proc. Of the 7th ACM SIGCOMM conference on Internet measurement, 2007.

[CRAM06] Peter Cramton, Yoav Shoham, and Richard Steinberg. Combinatorial Auctions. MIT Press, 2006.

[CSK11] Y. Choi, J. A. Silvester, and H. Kim, Analyzing and modeling of multimedia workload characteristics in a multi-service IP network. IEEE Internet Computing vol. 15(2), 2011.

[DAVID03] Herbert A. David and H. N. Nagaraja. Order Statistics. Wiley Series in Probability and Statistics. Wiley-Interscience, 2003.

[DOCK13] Docker, https://www.docker.io/, 2013.

Copyright © FUSION Consortium, January 2014

[FAMA09] J. Famaey, W. De Cock, T. Wauters, F. De Turck, B. Dhoedt, and P. Demeester. A latency-aware algorithm for dynamic service placement in large-scale overlays. In Proc. of IFIP/IEEE IM (Intl. Symp. on Int. Net. Man.), pages 414–421, 2009.

[FELD05] Michal Feldman, Kevin Lai, and Li Zhang. A Price-Anticipating Resource Allocation Mechanism for Distributed Shared Clusters. In Proc. ACM EC, June 2005.

[FRRS12] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, Impact of traffic mix on caching performance in a content-centric network, arXiv:1202.0108, 2012

[FU03] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: an architecture for secure resource peering. In Proc. of ACM Symp. on Op. Sys. Princ. (SOSP), 2003.

[GALM07] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, Youtube traffic characterization: a view from the edge, ACM SIGCOMM Conference on Internet Measurement, 2007

[GARG02] Rajat P. Garg and Ilya Sharapov. Techniques for Optimizing Applications - High Performance Computing. Prentice-Hall, 2002.

[GOPA11] Gopalakrishnan N. And Jayarekha P., Pre-allocation strategies of computational resources in cloud computing using adaptive resonance theory-2, Internatonal journal on Cloud Computing: Services and Architectures (IJCCSA), 2(1), August 2011.

[GT12] R. Garg, and R. Telang, Inferring App Demand from Publicly Available Data. MIS Quarterly, Forthcoming, 2012

[GUPT02] Rohit Gupta, Varun Sekhri, and Arun K. Somani. CompuP2P: An Architecture for Internet Computing Using Peer-to-Peer Networks. IEEE Trans. Parallel and Distrib. Syst. (TPDS), 17(11), 2006.

[HKKT12] K. Huguenin, A.M. Kermarrec, K. Kloudas, F. Taiani, Content and geographical locality in user-generated content sharing systems, Proc. of 22nd SIGMM International workshop on network and operating systems support for digital audio and video, 2012.

[IRMO11a] IRMOS Project Deliverable, D3.1.4 Updated Final version of IRMOS Overall Architecture, ICCS/NTUA and other partners, 2011.

[IRMO11b] IRMOS Project Deliverable, D7.2.2 Final version of Path Manager Architecture, ICCS/NTUA and other partners, 2010.

[IRMO11c] IRMOS Project Deliverable, D7.1.1 ISONI addressing schemes, USTUTT and other partners, 2008.

[JANG11] Jae-Wan Jang, Myeongjae Jeon, Hyo-Sil Kim, Heeseung Jo, Jin-Soo Kim, and Seungryoul Maeng. Energy Reduction in Consolidated Servers through Memory-Aware Virtual Machine Scheduling. IEEE Trans. on Computers, 60(4):552–564, 2011.

[JKHS12] M. Jeon, Y. Kim, J. Hwang, J. Lee, E. Seo, Workload characterization and performance implications of large-scale blog servers, ACM Trans. Web(6), 2012.

[JTPS13] Q. Jang, C.-H. Tan, C. W. Phang, J. Sutanto, and K.-K. Wei, Understanding Chinese online users and their visits to websites: Application of Zipf’s law, International Journal of Information Management vol. 33, 2013

[KCLC13] K. Kamath, J. Caverlee, K. Lee, Z. Cheng, Spatio-temporal dynamics of online memes: a study of geo-tagged tweets, Proc. of the 22nd international conference on World Wide Web, 2013.

[KAKA04] Sham M. Kakade, Michael Kearns, and Luis E. Ortiz. Graphical Economics. Learning Theory (Springer LNCS), 3120:17–32, 2004.

Copyright © FUSION Consortium, January 2014

[KARV06] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, and A. Tantawi. Dynamic placement for clustered web applications. In Proc. of ACM WWW, pages 595–604, USA, 2006.

[KKVZ06] Kolyshkin, K., Virtualization in Linux, http://download.openvz.org/doc/openvz-intro.pdf, 2006.

[KRAU02] Klaus Krauter, Rajkumar Buyya, and Muthucumaru Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. Softw. Pract. Exper., 32(2):135–164, February 2002.

[LAND09] Raul Landa, Richard Clegg, Eleni Mykoniati, David Griffin, and Miguel Rio. A Sybilproof Indirect Reciprocity Mechanism for Peer-to-Peer Networks. In Proc. of INFOCOM, 2009.

[LEKA12] Seung-Ik Lee, Shin-Gak Kang, NGSON: features, state of the art, and realization, IEEE Comm. Mag., Jan 2012.

[LEVI08] Dave Levin, Katrina LaCurts, Neil Spring, and Bobby Bhattacharjee. BitTorrent is an auction: analyzing and improving BitTorrent's incentives. In Proc. of SIGCOMM, pages 243–254, 2008.

[LWAT11] M. Lin, A. Wierman, L. H. Andrew, E. Thereska, Dynamic right-sizing for power-proportional data centers, Proceedings IEEE INFOCOM, 20111

[LXC13] Linux containers, http://linuxcontainers.org/, 2013.

[NGSO11] NGSON (Next Generation Service Oriented Network) – IEEE 1903, "Standard for the Functional Architecture of Next Generation Service Overlay Networks", 2011.

[Open13] OpenStack Architecture Overview, http://docs.openstack.org/trunk/training-guides/content/module001-ch004-openstack-architecture.html, 2013.

[RLZ13] Ren, Y., Liu, L., Zhang, Q., Wu, Q., Wu, J., Kong, J., and Dai, H, "Residency-Aware Virtual Machine Communication Optimization: Design Choices and Techniques." Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing. IEEE Computer Society, 2013.

[SAG11] T. Silva, J.M. Almeida, and D. Guedes, Live streaming of user generated videos: Workload characterization and content delivery architectures, Comput. Netw. 55(18), 2011.

[SCC13] Y.S. Sun, Y.-F. Chen, and M. C. Chen, A workload analysis of live event broadcast service in cloud, Procedia Computer Science 19, 2013

[SDJS10] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, H. Bhogan, Volley: automated data placement for geo-distributed cloud services, Proc. Of the 7th UNIX conference on Networked systems design and implementation, 2010.

[SDM09] M. Suznjevic, O. Dobrijevic, M. Matijasevic, MMORPG player actions: network performance, session patterns and latency requirements analysis, Multimedia Tools and Applications, vol 45(1-3), 2009

[SHIP13] Shipper, https://npmjs.org/package/shipper, 2013.

[SHNE05] Jeffrey Shneidman, Chaki Ng, David C. Parkes, Alvin AuYoung, Alex C. Snoeren, Amin Vahdat, and Brent Chun. Why Markets Could (But Don't Currently) Solve Resource Allocation Problems in Systems. In Proc. of ACM HotOS, USA, June 2005. USENIX - (TCOS).

Copyright © FUSION Consortium, January 2014

[STILL10] Mark Stillwell, David Schanzenbach, Frédéric Vivien, and Henri Casanova. Resource allocation algorithms for virtualized service hosting platforms. J. Parallel Distrib. Comput., 70(9):962–974, September 2010.

[TOSCA13] OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) specification, version 1.0, http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf

[VICK61] William Vickrey. Counterspeculation, Auctions, and Competitive Sealed Tenders. The Journal of Finance, 16(1):8–37, 1961.

[VPG05] M. Vilas, X. G. Paneda, R. Garcia, D. Melendi, and V. G. Garica, User behavior analysis of a video-on-demand service with a wide variety of subjects and lengths, Proc. EUROMICRO, 2005

[WALD92] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. IEEE Trans. Software Eng., 18(2):103–117, 1992.

[WANG05] W. Wang and B. Li. Market-driven bandwidth allocation in selfish overlay networks. In Proc. of IEEE INFOCOM, pages 2578–2589, 2005.

[WANG92] Zheng Wang and Jon Crowcroft. Analysis of Shortest-path Routing Algorithms in a Dynamic Network Environment. SIGCOMM Comput. Commun. Rev., 22(2):63–71, April 1992.

[WLCW12] K. Wang, M. Lin, F. Ciucu, A. Wierman, and C. Lin, Characterizing the impact of the workload on the value of dynamic resizing in data centers, ACM SIGMETRICS Performance Evaluation Review vol. 40(1), 2012.

[YZZZ06] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng, Understanding user behaviour in large-scale video-on-demand systems, SIGOPS Oper. Syst. Rev. 40, 2006.

[ZAMA11] S. Zaman and D. Grosu. A Distributed Algorithm for the Replica Placement Problem. IEEE Trans. Parallel and Distrib. Syst. (TPDS), 22(9):1455–1468, 2011.

[ZKI13] B. Zhang, G. Kreitz, M. Isaksson, J. Ubillos, G. Urdaneta, J. A. Pouwelse, and D. Epema, Understanding user behavior in Spotify, IEEE Infocom, 2013

[ZLR13] Zhang, Q., Liu, L., Ren, Y., Lee, K., Tang, Y., Zhao, X., & Zhou, Y. Residency Aware Inter-VM Communication in Virtualized Cloud: Performance Measurement and Analysis. In 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), (pp. 204-211). IEEE.

Copyright © FUSION Consortium, January 2014