Future Service Oriented Networks

www.fusion-project.eu

Deliverable D5.2

Final use case design, report on prototype deployment and initial evaluation results

Public report, Version 1.1, 19 December 2014

Authors

UCL	David Griffin, Miguel Rio, Khoa Phan
ALUB	Frederik Vandeputte, Luc Vermoesen
TPSA	Dariusz Bursztynowski
SPINOR	Michael Franke, Mahy Aly, Folker Schamel
IMINDS	Pieter Simoens, Lander Van Herzeele, Piet Smet
Reviewers	Pieter Simoens, Dariusz Bursztynowski, Frederik Vandeputte, Folker Schamel

Abstract

This deliverable describes the final application prototype use case design, reports on the current status of the prototype deployment and reports initial evaluation results. The four applications prototype are the media services Electronic Program Guide (EPG), thin client 3D game, media dashboard, together with the technical utility streamer service. The applications prototype designs, implementations and their use case scenarios are described, including the current status of their implementations, followed by an analysis of how the prototypes demonstrate and validate specific features of the FUSION prototype. The FUSION features demonstrated by the applications prototypes can be classified in deployment, service selection, scaling and load balancing. The status of the current prototype for FUSION-enabled services and host environment deployment and its initial deployment on the virtual wall at the iMinds facilities is reported. The status report is followed at a first evaluation of this FUSION prototype.

Keywords FUSION, use cases, requirements, demonstrator, testbeds, test scenarios

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
08/09/2014	Folker Schamel	Skeleton	0.1	Initial ToC
15/09/2014	Folker Schamel	Improved Skeleton	0.2	Revised ToC
10/10/2014	Michael Franke	Improved Skeleton	0.3	Consolidated input
05/11/2014	Michael Franke	ALUB, IMINDS, SPINOR major input	0.4	Consolidated
02/12/2014	Folker Schamel	Most contributions included	0.8	Review and rework overall document
02/12/2014	Folker Schamel	Most contributions included	0.9	Rework overall document and resolve formatting problems
05/12/2014	Folker Schamel	Most parts complete	0.10	Add various missing paragraphs
15/12/2014	Folker Schamel	Most parts complete	0.11	Integrate changes and general reworking
19/12/2014	Folker Schamel	Completed	1.0	Integrated various review changes
19/12/2014	Folker Schamel	Completed	1.1	Additional review and additional sections about service selection and placement

GLOSSARY OF ACRONYMS

4G	Fourth generation of mobile phone mobile communication technology standards
API	Application Program Interface
AR	Augmented reality
BGP	Border Gateway Protocol
BW	Bandwidth
C++	Object Oriented Programming Language
CPU	Central Processing Unit
Ctl	Control
DC	Data Center
DNS	Domain Name System
DRAM	Dynamic Random Access Memory
EC2	Amazon Elastic Compute Cloud
EPG	Electronic Program Guide
EZ	FUSION Execution Zone
FUSION	Future Service Oriented Networks
GB	Gigabytes
GPS	Global Positioning System
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDD	Hard Disk Drive
HTTP	HyperText Transport Protocol
IP	Internet Protocol
ISP	Internet Service Provider
MOSCOW	MUST, SHOULD, COULD, WON'T
MPLS	Multi Protocol Label Switching
NFS	Network File System
PaaS	Platform as a Service
PoC	Proof of Concept
QoS	Quality of Service
REST	Representational state transfer
RFB	Remote Frame Buffer
RGBA32	Red Green Blue Alpha 32 bit color space representation
RPC	Remote Procedure Call
Rsp	Response
RTP	Real-time Transport Protocol
RTSP	Real-time Streaming Protocol
RTT	Round-Trip Time
SaaS	Software as a Service
SMART	Specific, measurable, attainable, relevant and time-bound

SSD	Solid State Drive
SSH	Secure Shell
Tb/s	Terrabit per Second
TOSCA	OASIS Topology and Orchestration Specification for Cloud Applications
UI	User Interface
VLAN	Virtual Local Area Network
VM	Virtual Machine
VNC	Virtual Network Computing
VoD	Video on Demand
VPN	Virtual Private Network
WAN	Wide Area Network
Wifi	WLAN products based on the IEEE 802.11 standards
WLAN	Wireless local area network
XML	Extensible Markup Language
YUV4MPEG2	Uncompressed frames of YCbCr video formatted as YCbCr 4:2:0

EXECUTIVE SUMMARY

This deliverable describes the final application prototype use case design, reports on the current status of the prototype deployment and reports initial evaluation results.

The four applications prototypes are the media services Electronic Program Guide (EPG), thin client 3D game, media dashboard, together with the technical utility streamer service. This document describes the functionality, architecture and implementation of these applications prototypes and their current implementation status.

The features demonstrated by the applications prototypes can be classified in deployment of composite services and deployment taking into account metric performance and server capabilities, service selection based both on network and service metrics, deployment and scaling of resource demanding personalized services with tight QoS constraints, load balancing and service scaling for efficient server resource usage, faster deployment and dynamic inter-zone scaling based on changing demand patterns and network conditions.

The status of the current prototype for FUSION-enabled services and host environment deployment and its initial deployment on the virtual wall at the iMinds facilities is reported.

The status report is followed at a first evaluation of this FUSION prototype, which at this stage focusses on a functional analysis of service registration and automatic deployment, session slot service scaling, and load-aware service scaling and resolving.

TABLE OF CONTENTS

1. SCOPE OF THIS DELIVERABLE

2. APPLICATION PROTOTYPE DESIGN AND IMPLEMENTATION

Based on the analysis of use cases in D2.1 we have selected three application prototypes we are implementing to analyse, validate and demonstrate the various features of FUSION. These application prototypes are:

1. Advanced media services Electronic Program Guide (EPG), including a technical utility streamer service
2. Thin client 3D game
3. Media dashboard

These use cases resemble real world use cases and concentrate on FUSION specific functionality. Each of these use cases will use several functionalities that are provided by FUSION which either enable this use case or simplify the development. The game and dashboard prototypes are technically similar by being mainly different configuration of the same underlying prototype implementation based on the commercial Shark 3D software of Spinor [SHARK3D].

2.1 Summary of FUSION functionalities demonstrated in each application prototypes

The following table summarizes, which functionalities of the FUSION framework are demonstrated with each use case. Compared to the table in section 3.1, which focuses on the innovation of the different FUSION features, the following table focuses on mapping FUSION features to application prototypes.

	Advanced EPG	**Thin client 3D game**	**Dashboard**
Service Session slots	Yes	Yes, based on node-based factory composition	
Multi-service configuration	Yes	Yes, based on node-based composition	
Static service graphs	Simple graph		
Dynamic service graphs	Service Requests		Dashboard service uses other services for media input
Efficient provisioning / deployment	Docker containers		
Evaluator services	Simple evaluator	Checking for GPU	
Resource sharing	Shared videos (and encoder)	Support for sharing GPU resources between session	
Multi user sessions	No	Yes	No
Re-use of existing software in FUSION environment	Based on Vampire framework	Based on the commercial Shark 3D software	

The following sections describe the prototypes more in detail.

2.2 Advanced EPG

Video services are becoming increasingly personalised, especially with the massive introduction of "second screen" or HUD applications. These applications complement the primary (television or broadcast) streams with personalised information, either on secondary devices such as tablets and smart phones, or overlaid on top of the main device. This personalized GUI will in the coming years become increasingly important as advanced content navigation, target for infomercials and product placement, and even to add social gaming aspects to the classic TV experience.

These advanced interactive user interfaces could be very fancy 2D or 3D graphical environments. This will lead to massive amounts of potentially highly interactive video and graphics content that needs to be generated or processed, and delivered to the end-user on-the-fly, which cannot easily be done on devices with limited capabilities (e.g., a smart TV).

The role of FUSION for such application use case is to optimally take into account the various resource requirements and constraints (both compute and networking) during deployment of new instances as well as the optimal selection of an instance for a particular client.

2.2.1 2D EPG Service

For the first PoC implementation of this application use case, we implemented a basic 2D interactive EPG that enables browsing through a number of dynamic or interactive video sources or static pictures, and that can easily be extended towards integrating the output from other FUSION services as well.

This EPG service is developed in the Vampire framework [FV09], a media processing framework developed inside Bell Labs for quickly building media applications consisting of a number of reusable media components, each of which can be mapped onto a number of application threads.

A snapshot of the basic EPG service is depicted in Figure 1.

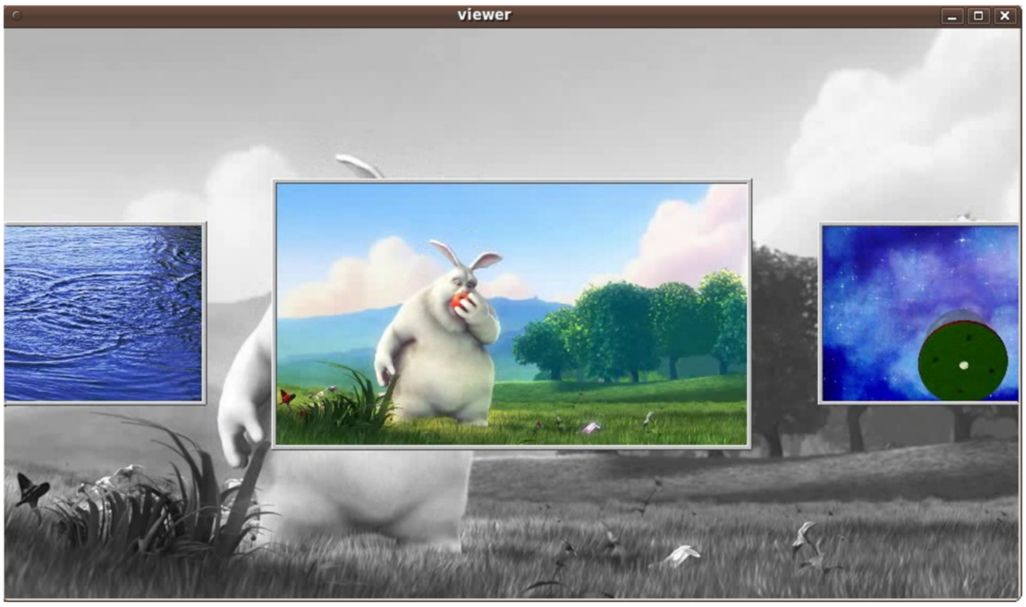


Figure 1– Screenshot of a 2D EPG service prototype

2.2.2 Functionality

For the initial setup, we limited the EPG service to only consist of a fixed number of static input streams, together with a series of static images. The end user can interact with the service either by pressing key strokes or mouse swipes to manually browse through the video sources.

Alternatively, the user can also toggle the service to automatically scroll through all video sources. We modified the implementation to support the basic raw video stream format as well as the RFB [RT10] feedback protocol for handling all service communication as described in Deliverable D5.1.

Using the underlying features of the Vampire framework, we reuse all input video sources across all sessions using an internal Vampire multicast mechanism to significantly reduce the resource requirements.

This service serves as an example of a personalized single-user streaming service, and can be used both as an atomic service (consisting of only the rendering component) or as a service component in a composite service (consisting of both a rendering and dedicated streamer component).

Regarding FUSION functionality, we implemented the following concepts in this first demonstrator service:

- We implemented the service session slots concept, allowing a configurable number of interactive sessions to be active at the same time;
- We implemented also the multiple service configuration concept, allowing multiple parameterizable service configurations to be added and removed at runtime, sharing the same set of session slots for optimal resource utilization. This allows to change the available parallel sessions, the output resolution, the endpoint port and the frame rate at runtime.
- We encapsulate the service components into a lightweight Docker container for efficient provisioning and deployment with minimal runtime overhead.

2.2.3 Architecture

This use case consists of one or two service components, namely the EPG rendering component and optionally a generic streamer component. Additionally, we also implemented an initial corresponding evaluator service associated with these service components.

The rendering component provides the core functionality of the EPG service and can be addressed individually by either clients or other services. Default, the EPG rendering service used the raw streaming format.

The streamer service component is a generic streamer component that currently takes in the raw streaming protocol and produces an H.264 encoded video stream. Secondly, it can also forward the incoming feedback stream to the connected service. Two key advantages of having a separate generic streaming service component are:

- This streamer component can be mapped onto specialized hardware that is optimized for efficient video encoding and streaming, such as GPUs or hardware encoders
- This enables to have more lightweight and specialized service components, not requiring the connecting service components to all have direct access to such hardware environments or having to implement such encoding capabilities internally. This can significantly improve deployment and runtime efficiency.

2.2.3.1 *EPG rendering service component*

The internal architecture of the EPG rendering service component is depicted in **Figure 2**. The key component in the internal application graph is the session and configuration factory component, that is able to automatically spawn a new session and service configuration specific graph on-the-fly that will handle the specific client session for that specific service configuration.

A user making a service request opens a connection to the TCP port of the corresponding service configuration (which is managed by FUSION orchestration and resolution layers). If there are still session available, the factory will create a parameterized session graph, make all connections with the various shared input videos, and passed the socket to the corresponding components. All this complexity is abstracted via the Vampire framework. Note that the shared input videos are only decoded once, and automatically multicasted via an internal shared memory Vampire protocol.

The available resource and service configuration session slots are all maintained at the session factory component. When a user finally disconnects, the session graph is completely removed, the session factor component is notified, which performs the necessary session accounting.

Using a Vampire-specific communication protocol, an external remote control management application can both read out or modify the application properties. Key properties that are relevant for FUSION include the available session slots, the service configurations and the service instantiation parameters. Each of these can be monitored or modified at runtime. This allows an external wrapper to monitor or even change the number of available session slots, add a new service configuration etc. In a FUSION-agnostic manner.

For the interactive streaming protocol, we leveraged the raw streaming format as well as the RFB format for the feedback channel. Unless one of the video sources originates from another FUSION service, we do not require the FUSION client API for automatically connecting to the optimal instance.

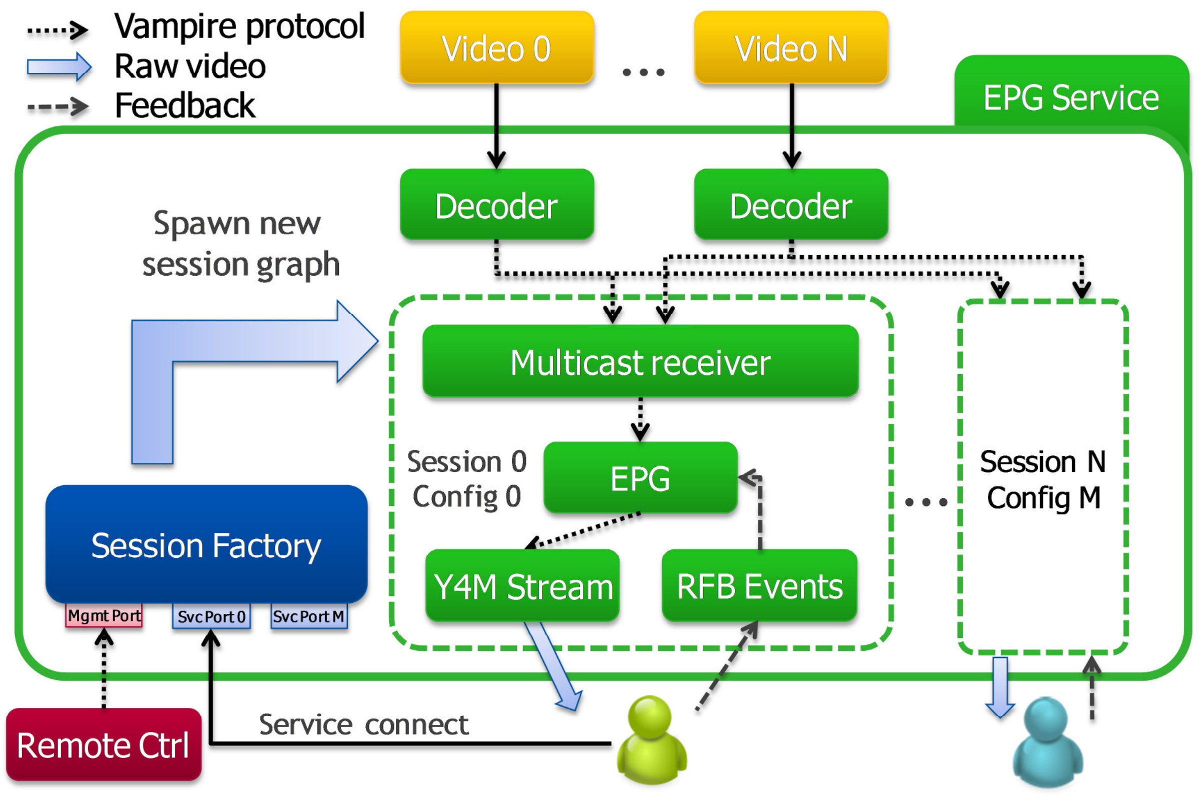


Figure 2 – Software architecture of the multi-session multi-configuration enabled EPG service

2.2.3.2 *Streamer service component*

The internal architecture of the streamer service component is depicted in Figure 3. The overall structure is quite similar to the EPG service component, and we used the same generic session factory Vampire component for handling all session and multi-configuration related aspects.

For every new streamer session, a new internal pipeline and context is created for handling the streaming session. Only at the time, the streamer session will open a connection to the upstream service. The service name can be either a FUSION service name or an explicit URL. We implemented a small FUSION-enabled API that will automatically detect whether the provided service name is a FUSION name or URL and respond accordingly. In case it is a FUSION service name, the small library will contact a FUSION service resolver for finding the endpoint of an optimal instance of that service. Once resolved, the library function will automatically try to open a socket connection to that instance.

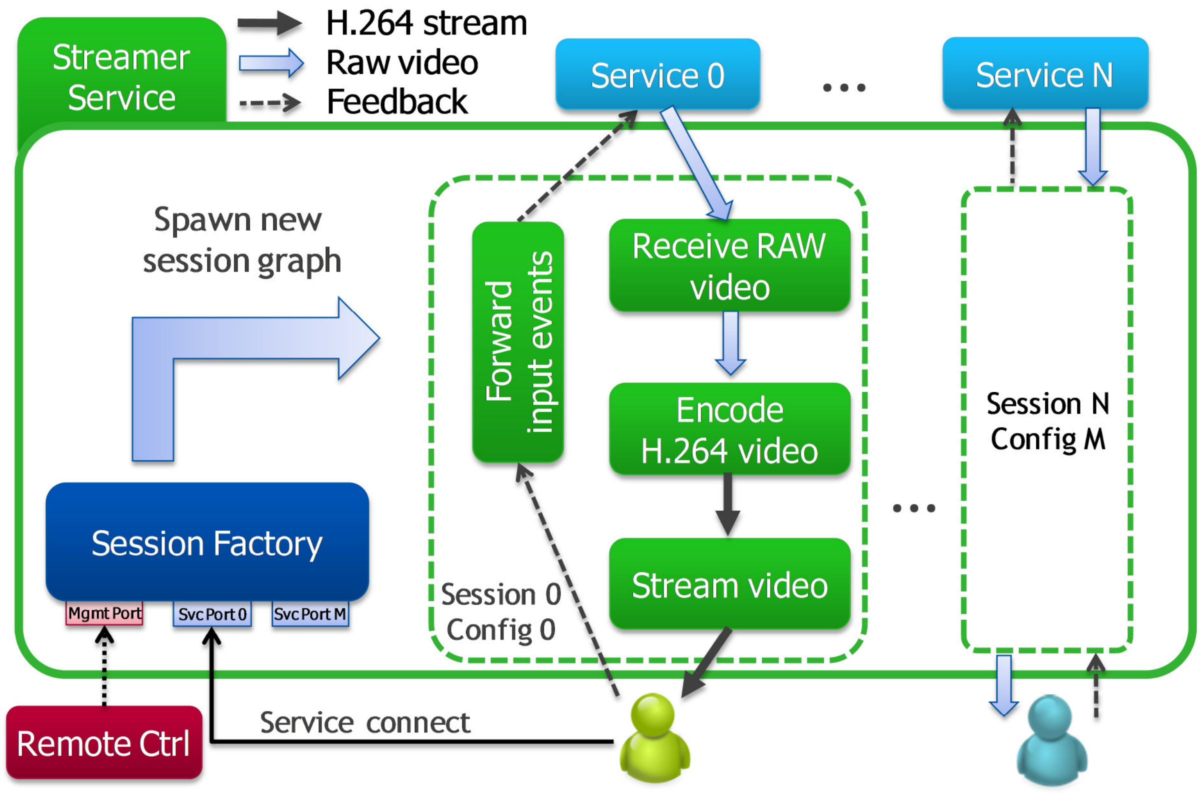


Figure 3 – Software architecture of the multi-session multi-configuration enabled streamer service

2.2.3.3 *Evaluator service*

As part of this EPG service, we also provided an initial evaluator service. Initially, this evaluator service is mainly to validate the FUSION workflow during service deployment, but in the last year of the project, we will extend and leverage this service for assessing what environment and execution zone is most suited for a running the EPG (and streaming) service.

2.2.4 Implementation

We have implemented the various software components using our Vampire framework [FV09], which enables quickly developing media processing applications on multi-core architectures. We implemented the service architecture, the functionality and the basic protocols described above in a number of software components. Although the service components do incorporate concepts such as service sessions, session slots and multiple service configurations, these were implemented in a FUSION agnostic manner.

All FUSION-specific communication was provided in an external simply Python wrapper that communicates with the zone manager and the ETCD key value store. This Python wrapper regularly inspects the Vampire application using a dedicated Vampire protocol regarding the available session slots, and pushes changes to the zone manager. Vice versa, it monitors the request for adding a new service configuration from the ETCD data store and subsequently triggers the Vampire application to add and configure a new service configuration on a new port using the same custom Vampire protocol. As such, the application itself can be designed independent of the FUSION protocol.

For streaming video and the feedback channel, we implemented the raw protocol described in Deliverable D5.1.

The application binary, wrapper script, static input videos, images and other artefacts subsequently were wrapped into a Docker container image using a simple Dockerfile, enabling easy and fast deployment of the service on any Docker-enabled machine. We developed such container for both the EPG rendering and streamer service components.

For reusability and fast provisioning, we made optimal usage of the image stacking concept in Docker, where different layers of containers can be layered on top of each other and shared in a hierarchical manner. As such the base layers (e.g., consisting of the basic libraries) can be shared by many Docker images, and only the application-specific binaries, libraries and artefacts need to be provided in a separate layer. When subsequently provisioning a new machine with a new Docker container, only the upper file system layers need to fetched remotely, and not the entire VM image.

As an example, the current full hierarchical structure of all container image layers as currently used for the FUSION prototype and demonstrator components is shown in **Figure 4**.

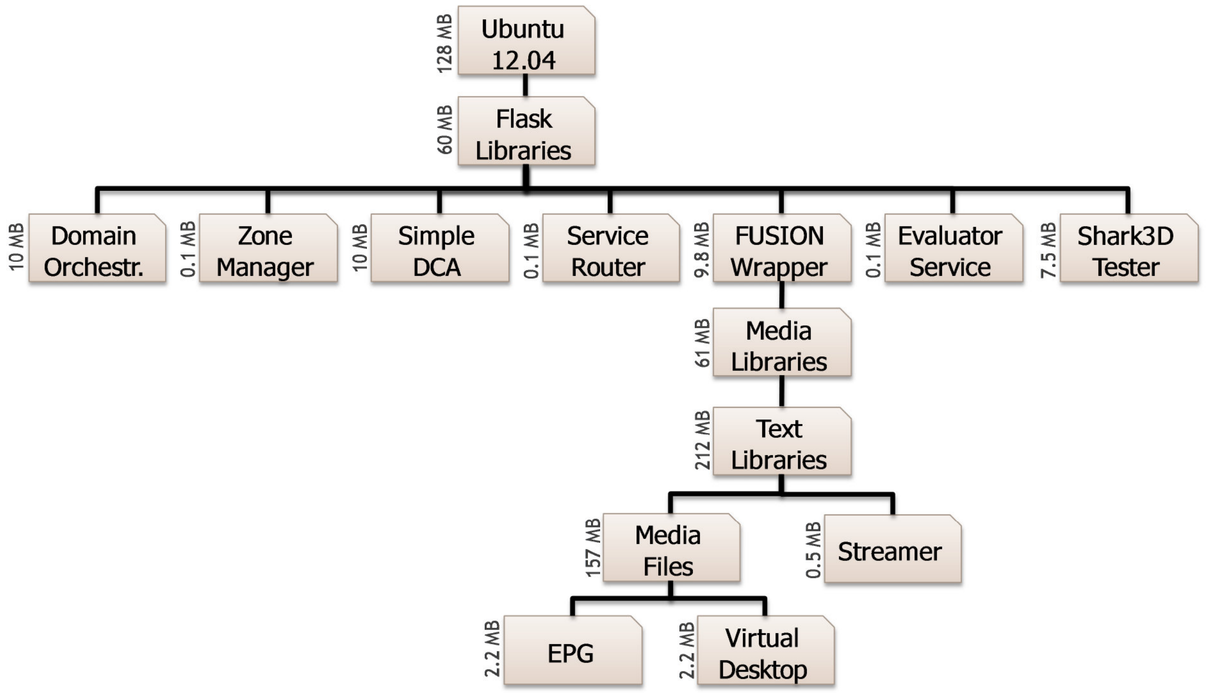


Figure 4 – Docker image layers with relative layer sizes

As can be seen, all FUSION orchestration prototype components (i.e., zone manager, DCA, domain orchestrator and resolver) are built on top of the same Ubuntu and Flask library layers. As a result, the size of the application-specific layer is extremely small. Similarly for the application services and evaluator service, by carefully reusing particular libraries as much as possible by putting them in common image layers, it is possible to keep the sizes of the application-specific layers to a bare minimum. For example, image the EPG service is currently already provisioned in some execution zone. In case the virtual desktop service needs to be provisioned and deployed, only the last layer (i.e., 2.1 MB) needs to be fetched remotely during provisioning, instead of several hundreds of megabytes or gigabytes of data, significantly reducing both the provisioning delay, storage requirements as well as network bandwidth requirements.

Note that for the classical VM-based deployment, we simply wrap the Docker container in a VM, which is something that the DCA layer can very easily can do automatically on any IaaS cloud infrastructure, by injecting the Docker container in an upfront prepared and platform optimized VM.

2.2.5 Integration

Figure 5 depicts how the EPG rendering service is currently integrated in the FUSION demonstrator; note that the integration of the streamer service component is identical. As mentioned in the previous section, the wrapper script is responsible for all FUSION-specific interactions as well as starting the EPG application. The entire service is wrapped in a Docker container, which the DCA layer knows how to efficiently deploy on the underlying architecture.

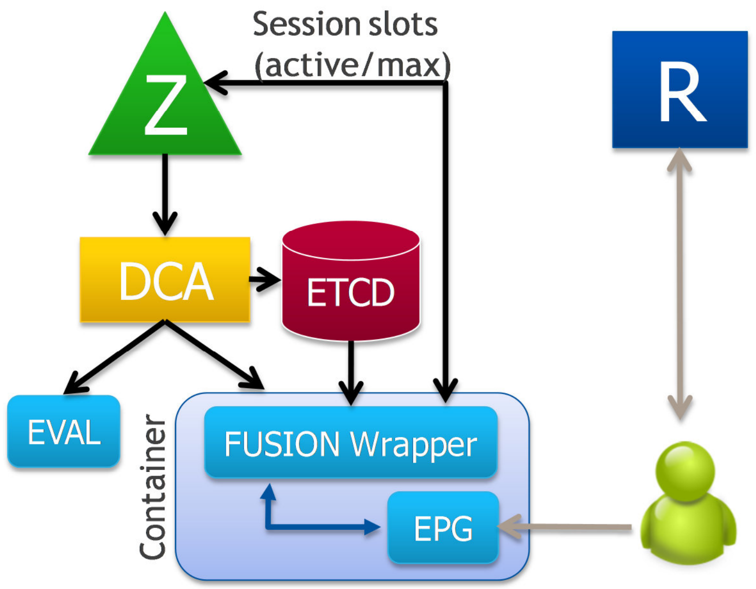


Figure 5 – Integration of the EPG service in FUSION

Apart from the Docker container, we also provided a corresponding evaluator service (which is also wrapped in a Docker container), as well as an initial simple manifest which is compatible with the current version of the prototype.

As mentioned earlier, the service currently supports session slots, service instantiation parameters as well as multiple dynamic service configurations.

2.2.6 Future implementations

In the final year of the project, we may integrate a second EPG service, such as for example a 3D EPG service as depicted in Figure 6, which has more demanding resource requirements (e.g., GPU availability, etc.), and for which the thin-client approach is even more crucial, especially if such EPG services also should be easily supported on TVs, without being constrained by the limited capabilities of the device with the lowest capabilities.

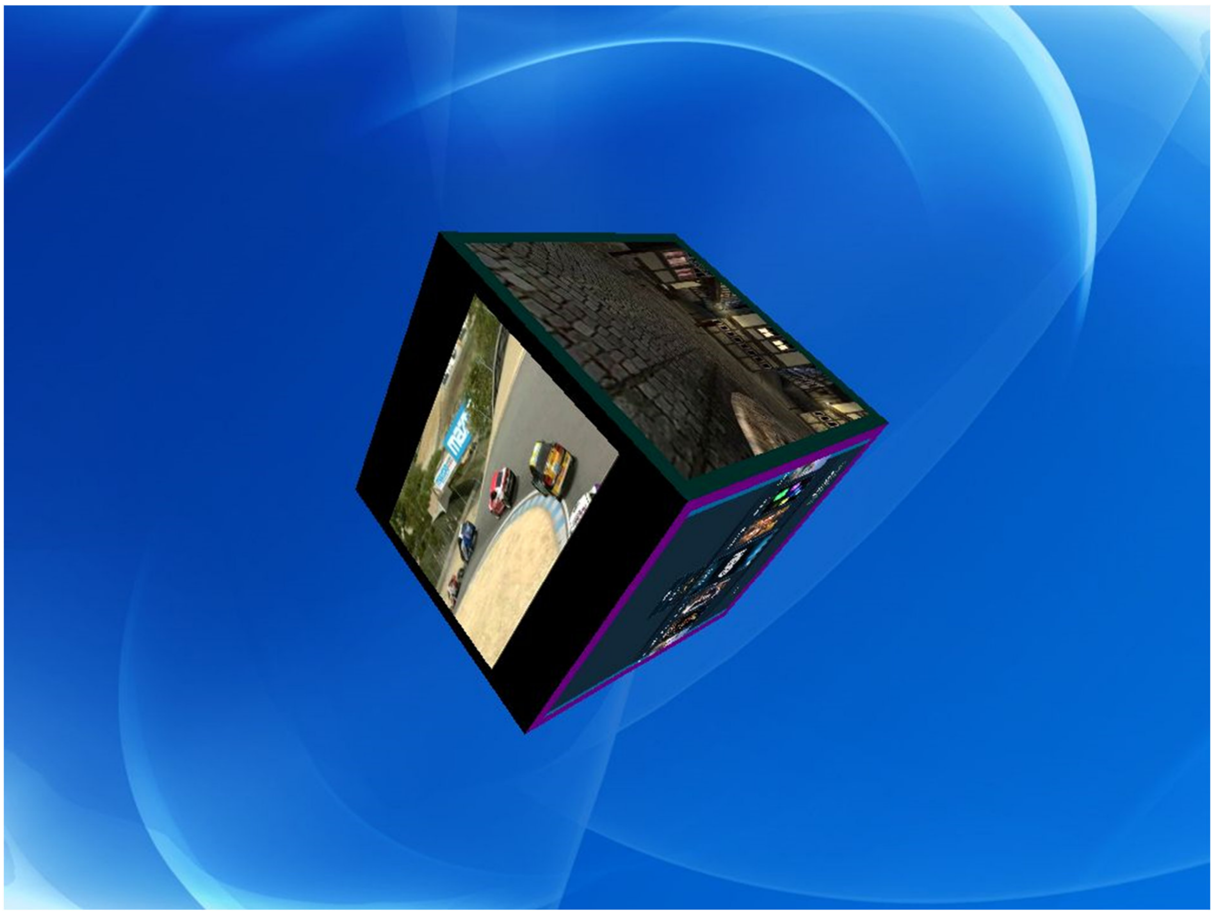


Figure 6– 3D cube EPG prototype

2.3 Thin client 3D game

A thin client game is an interactive software where the rendering of the 3D scenes is not done on the end user's device but on a separate server from where it will be delivered as a video stream. The end users only launches a viewer application that decompresses the video stream and presents it to the user. Additionally the viewer has functionality to capture input and pass it back to the rendering server. All in all this approach is comparable to remote desktop applications like the RDP or VNC protocol.

It is a more and more interesting for application developers, since applications that need high computational power can also be accessed on weak (e.g. mobile) devices.

The thin client 3D game will be implemented in two steps: in a first step the game will be implemented as a single user application where world simulation and 3D rendering are performed by one single service. This step is done as a preparation of the second step, where the application will be extended to multi-user sessions (i.e. multiplayer games). For this the world simulation and the rendering are split into two separate services. The reason is that this approach will be more common in real applications: it is easier to scale the applications to more users if the world simulation is done in a separate service because the calculation power needed for the 3D rendering can be offloaded to separate hardware. Since the world simulation has to be done centralized (except some more sophisticated approaches taken for example in massively multiplayer games), but rendering can be decentralized, this is the optimal architecture for scaling the number of users.

2.3.1 Functionality

The first step of the demonstrator can be interpreted as a single user service. Since the video will be streamed to the end user device this will be a streaming service. The second step of the demonstrator will implement a multiplayer game and can therefore be seen as a multi user streaming service.

To perform the rendering it is necessary to have access to the GPU. This is a non-standard requirement towards the execution zone which has to be checked whether it is met by using the evaluator services.

At the testing site probably only one GPU powered server will be available, which has to be selected by the evaluator service. This means that for a multiplayer game (step 2) with two users both have to be rendered on the same hardware. The world simulation will run on separate hardware decoupled from the rendering services. Since each user's rendering output can be seen as independent from the other's rendering output, the rendering could be realized as two independent rendering services or as one rendering services with two different session slots, which use resource sharing (e.g. the 3d model data).

Compared to the EPG service the network distance between streaming source and consumer is more relevant, because in contrast to the EPG service the game client is interactive. Buffering of video streams is therefore not possible.

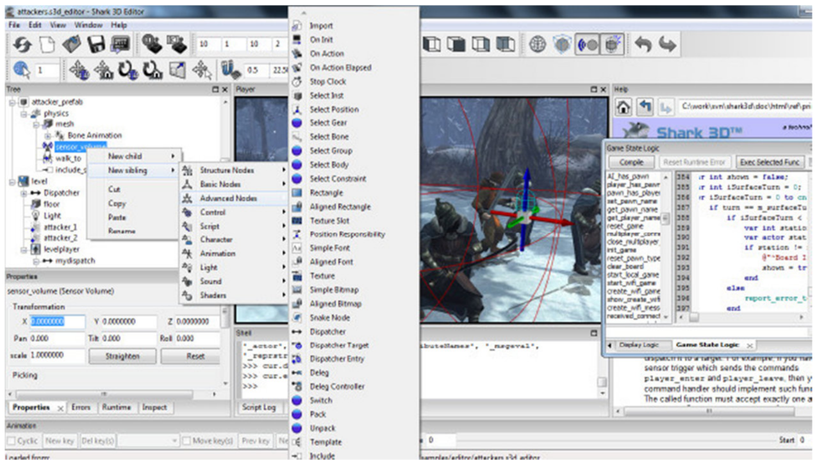
This demonstrator will largely depend on the utilization of the existing Shark 3D software [SHARK3D]. Therefore this demonstrator can also be seen as a practical test to measure how much changes and adjustments have to be applied to existing software to enable it as a FUSION service. Compared to the EPG service the code base that has to be prepared for FUSION is larger and may therefore provide a valid scenario to test, which efforts have to be made to enable existing software for FUSION.

To summarize this demonstrator will make use of the following functionalities provided by FUSION:

- Evaluator services
- Low latency streaming services
- Session slots with resource sharing
- Single and multi-user services
- Integration of existing professional software into FUSION
- Hardware access (GPU)

2.3.2 Architecture

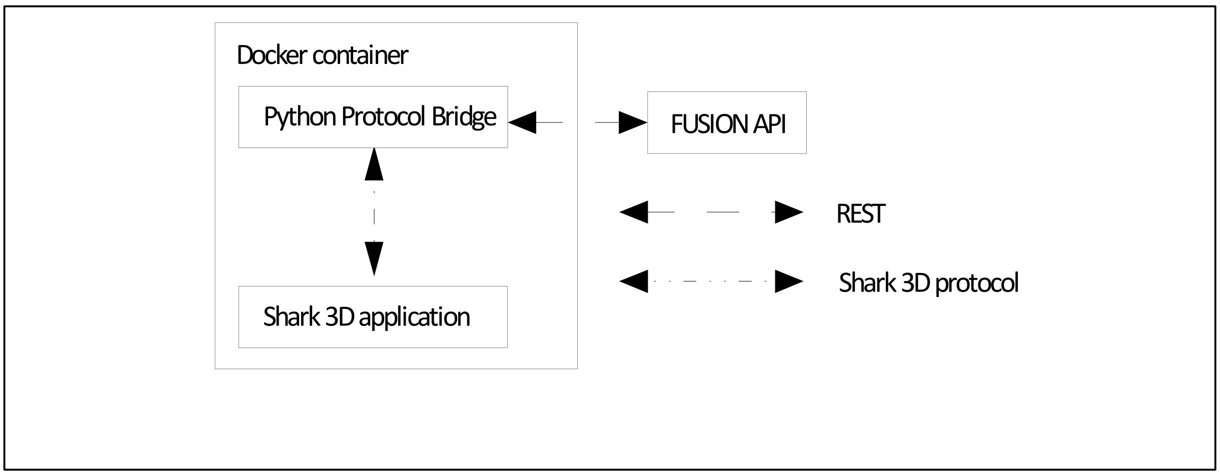
For this demonstrator the Shark 3D software [SHARK3D] will be packaged into docker containers and deployed the same way as for the EPG use case. From a FUSION perspective the Shark 3D containers will behave the same way as the EPG containers, so the existing infrastructure can be used.



The commercial Shark 3D Software [SHARK3D] used for creating the game prototype

Inside the containers, a connection between the Shark 3D software and the FUSION API has to be established. Two possible solutions are either to extend the Shark 3D software to directly call the REST API which means to add HTTP functionality to the software. This approach may be efficient regarding runtime behaviour. Another approach would be to package a separate bridging service inside the same container, which communicates to the Shark 3D software on the one hand and the FUSION API on the other hand. This service could be implemented in Python. The Shark 3D software already has an interface for network communication (Telnet based), so this approach would merely mean write a translator from custom Shark 3D protocol to FUSION REST calls.

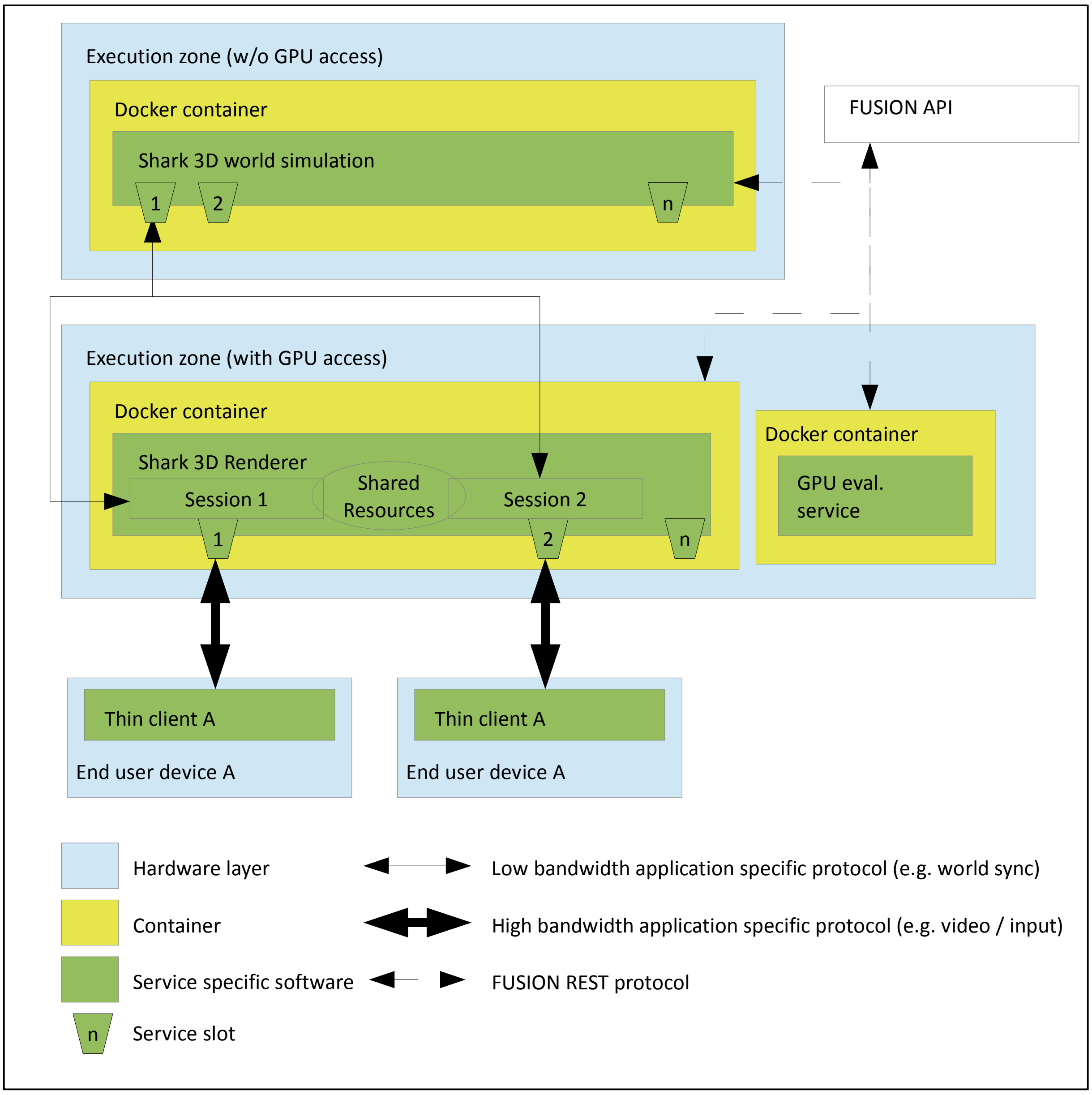
Since the Python libraries to communicate with FUSION are already available from the EPG use case and there are Python implementations available for the Shark 3D protocol as well, this can be implemented with very few efforts. The runtime efficiency will not be as good as for the integrated approach, but since calls to the FUSION API are for management purposes only (no streaming data delivered), the overhead will be small. Therefore this solution is currently preferred and will be implemented.



For the first step of the demonstrator the world simulation and the rendering will be performed in one monolithic service. This service will be connected to the thin client application on the user device.

In the second step the service will be split up into two services, a simulation and a rendering service. The rendering services will connect to a single simulation session which may run on the same execution zone or on another one. Since only one execution zone with GPU will be available, the rendering service will only be instantiated once with multiple session slots.

The connections between rendering service instance and world simulation instance will be using the already existing Shark 3D network protocol, the connections between rendering service instance and thin clients will be realized using a streaming protocol and backwards input channel.



2.3.3 Implementation

For packaging and deployment a docker / vampire setup as described in the EPG use case will be used. The FUSION API will be realized as REST service as described in the EPG use case.

The Shark 3D software is a compiled C++ software consisting of several modules that are linked into one Linux executable using static linking [SHARK3D]. The configuration of each module is done with separate configuration files, which make it possible to start and configure a module as needed. Therefore the one single executable can be used both for the world simulation and the rendering service. The different behaviour is achieved by different configurations. This also enables easier deployment if pre-distribution is chosen, because only differences in configuration have to be transmitted as deltas to the execution zones.

The GPU evaluation service however may be also be based on the current Shark 3D software, since this software can evaluate best which capabilities are needed for the rendering (shader model etc.). It has to be decided whether the GPU evaluation service is also a full version of the Shark 3D software with just another configuration or a separately built software derived from the existing one. Probably, because auf the necessity to keep the evaluators very lightweight, it may be necessary to strip all unneeded modules from the existing software and create a very thin separate software for the evaluation service. This approach is currently planned.

The protocol for connecting the thin clients to the render services will be based on the VNC / RFB protocols. These have the advantage, that applications implementing these protocols are already existing, meaning that for testing purposes an existing application can be used to implement the missing functionality at the renderer. One disadvantage is that the available compressions for standard RFB are optimized for remote desktop connections, meaning that they work best if large areas of the transmitted image remain static and only small areas change. For this use case this may however not be the case. It will therefore probably be necessary to extend the existing protocols by additional compression algorithms. For the FUSION port for example, Shark 3D was prepared to render compressed video streams as 3D textures on the one hand and to render output which can then be encoded as video stream on the other hand. This functionality is provided by the enhancement for the graphics back ends in combination with an integration of the FFMPEG library [FFM14]. The advantage of the library is that it provides support for different state of the art compression algorithms.

Another architectural change in the Shark 3D engine, which would also be necessary for any other software package using a similar architecture, is the abstraction of the input to also accept input that is transmitted via network, for example by using the input events provided by the RFB.

One challenge here is the relatively complex handling of input. Since keyboard details are different for example on different operating systems (e.g. compare the Apple keyboard with a Windows keyboard) there has to be some abstraction and mappings between keys. The RFB already offers some rules for that, but they are not always as clear as they are expected to be.

When using advanced compression algorithms the standard VNC clients will no longer be suitable as thin client application. Two options are possible and can be implemented without big effort to enable demonstration of the described functionality. Either the existing client developed by ALUB for the EPG use case is extended or a proxy application is installed which locally converts the compressed video stream into uncompressed video and forwards this to the VNC client. Input data from the client is just passed through.

The connection between the world simulation service and the renderers will be realized by using the existing Shark 3D network protocol, the communication will be transparent to FUSION.

2.3.4 Current status and next steps

The following table summarizes the current status of the prototype implementation plus planned next steps.

Application prototype feature	**Description**	**Accomplishments so far**	**Next steps**
Session slots	Session producer for handling and managing service session requests on runtime side and corresponding editor extensions	Successful production of multiple session instantiations	Prototype specific definitions of shared and session-specific data

World simulation deployment	Includes reactivation of Spinor's Linux port of the Shark 3D world simulation and integration into Docker	Successful compilation of most modules.	Compilation of remaining modules. Packaging of binaries and creation of resource packages.
Renderer deployment	Includes reactivation of Spinor's Linux port of the Shark 3D renderer and integration into docker	First tests with Ubuntu and Docker and GPU access	Renderer code porting, compilation, packaging of binaries and creation of resource packages, especially shader code generation
Input channel	Keyboard, mouse based on VNC protocol	Basic implementation of VNC protocol	Connecting input data to session instances
Output video channels		Successful rendering into uncompressed image sequence	-
Output video channel compression	Based on ffmpeg.	Successful rendering into compressed video data stream	Streaming compressed video data into network connections
Single user game application	Setup game application where game world simulation and 3D rendering are performed by one single service	-	Game prototype programming
Multi-user game sessions	Setup multi-player game application	-	Game prototype programming

2.4 Dashboard prototype

The dashboard use case is based on gaming console main menus, which are a merger of EPGs and 3D games. Real time data from different sources is merged as video streams or interactive video streams (see Thin client 3D game use case) into an interactive world, where the user can move around and consume the data. It therefore integrates the both use cases described before in one more advanced and challenging application.

To create a representative example different service types will be integrated to one single dashboard application. The combination of the services will take place on demand, that means that the service graph will not be deployed statically but will be created depending on the user input at runtime.

2.4.1 Functionality

Since the Dashboard prototype is a mixture or EPG and Thin client 3D game, it combines the functionality of both use cases.

The most important additional feature is the dynamic combination of different services. The dashboard service as main service presents to the users the output of other services, like video streaming and chat service.

As with the Thin 3D Client the rendering will be performed on GPU, so also here the evaluator services are necessary. Since the graphics backend for both the Thin Client 3D Game and the Dashboard service are the same, it is possible to re-use the existing evaluator services. This can be seen as a general good practice in FUSION environment since the outcomes of evaluator service invocations may be cached and re-using existing evaluator services means re-using existing evaluator results and therefore reduces the number of necessary evaluator runs.

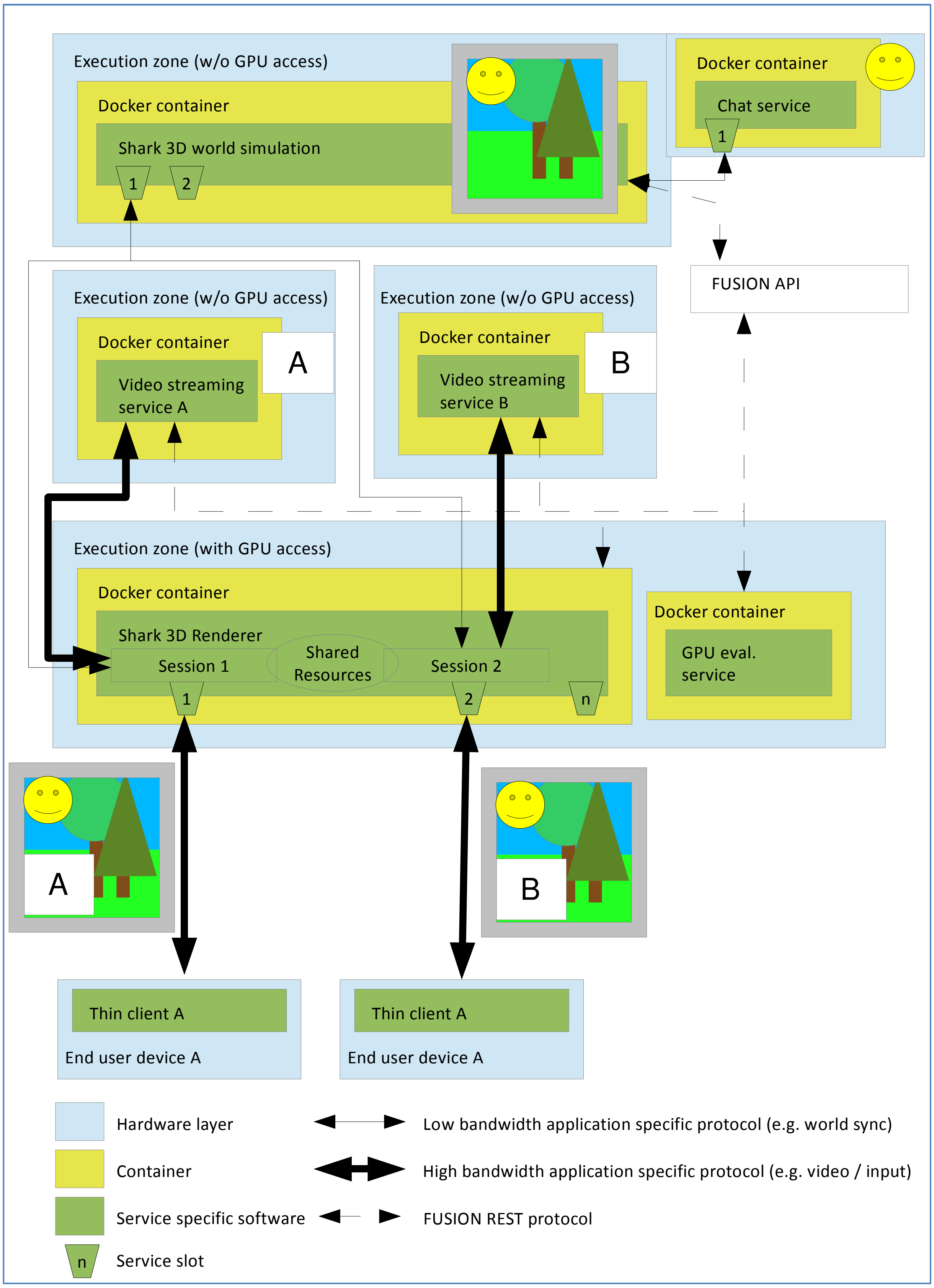
- Multi-user service
- Low latency streaming services
- Re-use of existing evaluator services
- Dynamic service graphs
- Session slots with resource sharing
- Hardware access (GPU)

2.4.2 Architecture

Main feature of this use case is the dynamic connection and disconnection of other services. This integration of other services can occur at different places, either at the world simulation service or at the rendering service. For video streams rendered into the graphic output it is for example not necessary to route them through the world simulation service since only output is affected. This connection can be made during runtime and can even be different for different users. This may for example be wanted to display different contents to different users which are in the same session (e.g. localized video streams or commercials).

Other services like the chat service rely on a centralized structure and may therefore be routed through the world simulation service.

The video streaming services need higher bandwidth connection than the world simulation server, so it may be necessary, to place them nearer to the rendering services than the world simulation. This may also be checked by a separate evaluation service.


Execution zone (w/o GPU access)
Docker container
Shark 3D world simulation
1
2
Docker container
Chat service
1
FUSION API
Execution zone (w/o GPU access)
Docker container
Video streaming service A
A
Execution zone (w/o GPU access)
Docker container
Video streaming service B
B
Execution zone (with GPU access)
Docker container
Shark 3D Renderer
Session 1
Shared Resources
Session 2
1
2
n
Docker container
GPU eval. service
A
B
Thin client A
End user device A
Thin client A
End user device A
Hardware layer
Container
Service specific software
Service slot
n
Low bandwidth application specific protocol (e.g. world sync)
High bandwidth application specific protocol (e.g. video / input)
FUSION REST protocol

2.4.3 Implementation

This use case extends the previous described use cases by a chat service and the combination of world simulation and streaming services. Since the chat service is only there for demonstration reasons, a simple implementation using Python will be deployed using the Docker environment described before. For simplicity, the communication between world simulation service and chat service will take place using a rest protocol. Inside the world simulation service, a Python bridge will translate between Shark 3D application and the chat service API.

The combination of the video streaming services and the world simulation service in the renderer services will be done the following way: The world simulation service provides the data to update the current world state in the renderers. This state includes a plane or other surface containing a video texture. The video texture is defined by a resource identifier, which will be a FUSION service identifier. This identifier is used at the rendering service to query the actual service instance providing the stream from FUSION. This query can be session specific, so that different users can get different streams.

The video stream is decoded using the FFMPEG library and the content is copied into the texture buffer each frame. This texture buffer is then used for the rendering of the scene. This way the same scene used by different users will receive different contents, depending on the session slots.

For real world applications this architecture can be used to reduce required network bandwidth because the video streaming services can be places nearby the renderers. The possibly far distance connection between renderer and world simulation only needs low bandwidth because of the highly optimized application specific geometry-based synchronization protocol.

2.4.4 Current status and next steps

The dashboard is based mainly on the same components as the game application. Therefore the following table only includes modules which are specifically for the dashboard.

Application prototype feature	Description	Accomplishments so far	Next steps
Output channel	Keyboard, mouse based on VNC protocol	-	Implementation and connection with session instances
Input video channel	Implementation of communication channels (video and input) between service components implemented by different FUSION partners	Streaming of uncompressed video stream into network communication channels	-
Input video channel decompression	Based on ffmpeg.	-	Implementation.
Connection with EPG prototype	Connection of the EPG prototype with the dashboard prototype	Statically configured connections established	Dynamically controlled connections
Single-user dashboard		-	Dashboard programming

Service connections	Dashboard connecting to media sources	-	Implementation of data connections and integration into prototype setup
Multi-user support		-	Implementation of user sessions
Chat		-	Implementation of chatting functionality

3. FUSION PROTOTYPE USE CASE SCENARIOS

While the previous chapter describes the application prototypes itself, this chapter describes how the integration of the FUSION component prototypes with the application prototypes are used to test, validate and demonstrate different use cases.

3.1 Summary of use case scenario characteristics

Compared to the table in section 2.1, which focusses on the relationship of FUSION features to the application prototype, the following table focusses on the characteristics of the FUSION features which are tested, validated and demonstrated with the FUSION prototype together with the application prototypes.

FUSION Feature	**Characteristic beyond state of the art**
Static service graph deployment	Deployment decision depends on the requirements of multiple related software components based on evaluator services
Dynamic service instance graph deployment using FUSION service selection	Implicit formation of service graphs without requiring complex coordination or specification
Better resource utilization through multi-configuration service components	Deployment of multi-configuration service component which can be part of multiple service (instance) graphs at instantiation time
Satisfying service specific requirements in heterogeneous environments	Evaluator services for taking into account static and dynamic service requirements as well as infrastructure capabilities at the service layer
Service selection based both on network and service metrics	Service resolution is able to select between service replicas running in different locations according to current server capacity and network performance characteristics, matching service-specific performance targets
Service deployment on third-party	More flexibility: Closer to the user, higher

hardware	QoS
Hardware abstraction for service providers	Light-weight containers for service instance isolation
Resource sharing of multi-media service instances	Multiple service instances of complex 3d applications sharing data structures
Fast deployment	Reduce startup-time from 10s or 100s of seconds to the order of seconds
Dynamic inter-zone scaling	Dynamic inter-zone scaling based on changing demand patterns and network conditions

3.1.1 Composite service deployment

For demonstrating how FUSION will handle the deployment and management of composite services as well as composite service instances we will focus on three main cases, on which we will shortly elaborate:

- Deploying a static service graph;
- Dynamic service instance graph deployment using FUSION service selection;
- Better resource utilization through multi-configuration service components.

The first case deals with deploying a static service graph within a FUSION execution zone. In this case, the graph is fully specified statically within the service manifest, and it is up to FUSION to most optimally deploy this in a zone. We will for now focus on deploying such graphs within a single execution zone. However, the same strategy can later be extended at the domain level to deploy such composite services across multiple execution zones.

Key aspects to demonstrate and evaluate concerns the role of the evaluator services in case of static service graphs, the configuration of the various service component instances at deployment time as well as the sharing of service component instances across multiple service graphs (possibly coming from different composite service types).

In the second case, the service graph is not statically described in a manifest, but dynamically and more implicitly formed and changed at run time. This allows for more flexible distributed composite service, without requiring complex coordination or specification. We will demonstrate how FUSION service selection can be used for very quickly and efficiently creating or changing the composition of service instances to form more complex dynamic (distributed) service instance graphs.

In the last case, we will demonstrate the advantages of multi-configuration service components. A multi-configuration service component is a service component that can be part of multiple and potentially independent service (instance) graphs at instantiation time.

We will demonstrate how the concept of session slots as well as flexible session-slot based service scaling can be leveraged for sharing service components across multiple composite services, thereby further increasing the resource sharing capabilities.

3.1.2 Service deployment taking into account metric performance and server capabilities (e.g. the edge)

Another feature to demonstrate is the effectiveness of the FUSION architecture for efficiently deploying demanding services with specific requirements in a distributed heterogeneous environment. We will demonstrate and evaluate the concept of evaluator services for taking into account static and dynamic service requirements as well as infrastructure capabilities at the service layer.

An example of a static service requirement includes the availability of a GPU that has the appropriate features. The evaluator service can very efficiently and in very high detail provide the necessary feedback via a simple score whether a particular execution environment is effective (and cost-efficient) or not.

An example of a more dynamic service requirement includes the overall QoS that a particular service experiences when running in a particular execution environment. Execution zones that oversubscribe their available resources too aggressively, resulting in poor QoS for the deployed services, should be penalized so that future service deployments may consider other execution zones or deployment environments. We will illustrate how evaluator services could leverage historical data for making better decisions for deploying particular services on particular environments.

At the platform layer, we will demonstrate the effectiveness of a heterogeneous cloud platform for efficiently deploying the services on the available infrastructure, taking into account application characteristics as well as platform capabilities for providing significantly better QoS and efficiency when deploying demanding or sensitive applications on an unknown (cloud) execution environment.

3.1.3 Service selection based both on network and service metrics

FUSION service resolution is able to resolve queries for a service type to return the “best” running instance of that service for the endpoint requesting the service. “Best” means according to a combination of server and network metrics while respecting policies of the ISP deploying the service resolution plane. Network metrics include latency and throughput, service metrics are abstracted as session slots, network policies are related to the cost of forwarding traffic over inter-domain links. Optimal selection depends on service-type and a service-specific utility function will be defined per service to guide the service resolver decisions.

A network performance database in each service resolution domain will be populated by various means – local domain network topology and monitoring data; metrics provided by specific remote service resolution domains, e.g. using ALTO network and cost map information; global network measurement data provided by systems such as RIPE Atlas; peer-to-peer monitoring information, e.g. to specific target execution zones in remote domains; QoE feedback from users from prior service invocations. The network performance data will be used by service resolvers to identify which execution zones, hosting a service instance, meet the target performance metrics for that service type.

Available session slots per service type will be advertised by execution zones to local service resolvers, together with additional service information, such as the histogram of service duration times. Service resolvers will use current session slot availability to determine where capacity exists to serve a query. Resolvers will aim to balance load between execution zones and the histogram of service execution duration will be used for this purpose.

Finally, for inter-domain service resolution to execution zones in remote service resolution domains the inter-domain links will have different costs according to the business relationships with adjacent ASes depending on whether they are customer-provider or peering relationships. Service resolution will aim to minimize the costs of traffic on inter-domain links while meeting network performance targets and available server capacity constraints.

Service resolution based purely on network performance metrics will not take into account the dynamics of available server capacity, potentially resulting in blocked service requests by the selected execution zone. Resolution based purely on available server capacity and service duration statistics will ensure service requests can be processed by the selected execution zone but network performance metrics (latency, throughput, etc.) to that destination may not be suitable for the service type. Resolution based on both network and service metrics will allow load to be balanced between execution zones while meeting the target network performance for the service and minimizing the cost of inter-domain traffic.

3.1.4 Platform for deployment and scaling of resource demanding personalized services with tight QoS constraints on a highly distributed infrastructure

The demonstrator will also act as a personalized service with QoS requirements. This will be implemented as a game prototype service with a thin client approach. The server-side rendering and compression will require tight QoS to enable interactivity for the user.

This gaming prototype will be an interactive 3D world, where the user can walk around, and interact with in-game objects. There will be a screen or advertising pillar that streams video data from other services into the 3D world. This ensures very close similarity with the interactive dashboard, so that the advantages of FUSION for both variants can be demonstrated within a single application.

There are several technologies and advantages of the FUSION project, that can be demonstrated by enabling the demonstrator to support this variant:

First of all, it can be demonstrated that FUSION enables service providers to publish demanding services without the need of installing costly hardware by themselves, but the operation of the datacenters and management of the resources is handled by FUSION or the infrastructure providers respectively. While similar approaches are already possible with cloud computing, the instantiation of the service near the user, which enables higher QoS, is not yet possible with today's cloud computing: large cloud computing providers like Amazon have large datacenters at fixed positions, which normally causes a high network distance to the end user while smaller cloud providers, which may have installations nearer to the user, only support smaller regions.

The second FUSION feature that can be demonstrated is that a homogenous environment is provided, independent where the service is running. For the demonstrator this will be a Linux x86 environment. When the service is bundled for this environment it can be started on any hardware managed by FUSION. To isolate the services from each other, a lightweight container approach using the Docker software will be used. This ensures that the services can not interfere, while on the other hand reducing the required disk space overhead compared to virtual machines.

For Docker containers it is possible – similar to some VM implementations – to create basic containers that contain the main environment which are per-distributed. The actual containers later only require the distribution of the container deltas, which can be very small, depending on the service. With this technology the time overhead for moving a certain container to a target machine can be reduced to a minimum.

This is a strong argument for using the Linux operating system, since Docker requires it. Another plus for Linux is the easier management of licenses, especially with regards to automatic downloading and starting of the containers, which would raise licensing issues when using other – probably commercial – operating systems.

The demonstrator will therefore be packaged as a Docker container and uploaded to a server accessible by FUSION. After registering the service it can be deployed by FUSION to a specific hardware and started.

To find the optimal position to start a service the FUSION architecture uses the concept of demonstrator services which are run in advance to test the available hardware whether it is feasible for a given service. For the demonstrator with the server-side rendering there are two features that must be fulfilled by the underlying hardware: first of all, GPU access must be granted. Second the network distance to the streaming client must be as short as possible to reduce the time between input and feedback. Both of these restrictions can be tested by the evaluator services.

To report the feedback whether GPU is available and the network quality is sufficient for the streaming of the rendered content, an interface to the management layer is required. This is not only restricted to the evaluator services but also can be used by the application itself. These interfaces widely replace static manifest files since they can dynamically calculate whether requirements are met or not in contrast to fixed restrictions that have to be expressed in manifest files.

These management interfaces will communicate using a REST protocol, which makes it especially easy for service provider to implement or adjust to their needs. For the implementation of the demonstrator variants for example the logic handling the communication will be written in Python with the option to change to a more efficient language if this would be necessary.

The management of different service instances and user management can be demonstrated with the demonstrator in different ways: Either multiple players join the same session like in a classical multi player game or it may be necessary to have separate sessions (if they both want to use it as a single player game) but it may be still efficient to handle both within the same service instance. The demonstrator will show the different approaches necessary to provide both functionalities.

By installing virtual screens in the game worlds, which display the content from a different service, the demonstrator also proves the possibility of combining multiple services dynamically. This is for example required by the dashboard use case, which combines multiple streams into one interactive application.

3.1.5 Load balancing and service scaling based on session slots enabling more efficient server resource usage of media services

Besides the better placement of services another advantage of FUSION is the better usage of resources. The demonstrator will make use of the session slot mechanism to offer multiple session slots to FUSION. With this technique, multiple users can connect to the same game rendering service without knowing about each other, i.e. both sessions will be handled as single player games. The advantage will be, that since both players play in similar worlds, the rendering service only has to load the graphics data on the GPU and can use it for rendering of both scenes.

Depending on the size of the graphical assets, the client may also benefit from a shorter start-up time of the service because if the service is already loading and has vacant slots available, starting the service from the viewer's perspective is only connecting to the existing instance and loading and probably decompressing the data into GPU memory is not required.

Precondition for this to work is that the service logic software supports multiple session slots in an intelligent way. A simple method would always be to internally (inside the container) start a new process. But in this case the resource sharing advantage would be lost. It is therefore necessary, that the software is prepared to serve multiple incoming requests in parallel. Efforts have been taken to prepare the Shark 3D engine for supporting multiple session slots, so that the demonstrator can benefit from these advantages.

On the other hand the rendering service can only handle a certain number of clients connected because, depending on the complexity of the scene and the underlying hardware, the rendering takes some time. This hardware dependency is also a good example for the usage of evaluator services and dynamic performance measurements as opposed to static manifests, because it allows a more fine grained reaction on performance bottlenecks by adjusting the number of available session slots.

3.1.6 Faster deployment

One of the SMART objectives in FUSION is to reduce the start-up time for remotely executed service component instances to within the order of seconds compared to today's equivalent operation of instantiating a virtual machine in 10s to 100s of seconds.

This is important in the on-demand scenario, where services may have to be deployed on-demand in a particular execution zone when a new service request is made by a user. For example, for the long-tail of less popular services, it would be too expensive to deploy at least one instance of a particular service in (almost) every execution zone close to every user. In such cases, instead of using a more centralized service instance that has worse latency characteristics, it could be beneficial to deploy a new instance on-the-fly close to the requesting user. This of course requires deployment delays in the order of seconds instead of minutes.

Another motivation is to be able to cope with flash crowds, where suddenly huge amounts of requests are made for particular services within a region. In such case, being able to very quickly scale out or deploy new instances will significantly improve perceived QoE. As we target mostly long-running demanding services in FUSION where each service instance only can handle a discrete amount of sessions in parallel, FUSION must provide efficient deployment and scaling mechanisms to effectively deal with such dynamic behavior.

As such, we will demonstrate the benefits of using lightweight containers both for packaging as well as deploying FUSION services, and compare this with classical VM-based service deployment. We will demonstrate the two main benefits:

- Very fast deployment, in the order of seconds (at most);
- Faster (pre)provisioning due to shareable container image layers.

3.1.7 Dynamic scaling based on changing demand patterns and network conditions

Current cloud scaling approaches only take into account load in the local data center to up- or downscale new service instances. In contrast, the FUSION project targets a global optimization of service deployment across a multitude of distributed, possibly smaller, execution platforms that interfaces with a network-aware service request resolution.

The contribution of the FUSION project in this domain is that we demonstrate the benefits of interfacing between the service orchestration layer and the service request resolution layer. The FUSION orchestration layer is responsible for monitoring the current service load and taking the necessary deployment and scaling decisions, given network layer conditions and policies. Especially the placement of multimedia services can have a significant impact on the network, as these services are long-lived and involve high-bandwidth streams. Conversely, service placement of interactive services like gaming require low latency to the users. This can only be achieved by deploying services at multiple places in the network.

The interfacing between the orchestration and service resolution layer involves two scaling mechanisms:

- Up- or downscaling instances in a particular zone
- Deploying the service in a new execution zone, or unregistering the service from that zone

Whereas similar approaches have been demonstrated in the past in the context of CDNs we specifically target thousands of services that need to be distributed across tens of microdatacenters. Content can be easily replicated across servers, but services need to be provisioned, can typically handle only a limited number of users in parallel or are highly personalized. Hence, we need to take into account service holding time distributions, as well as more limited infrastructure availability in microdatacenters.

3.2 Service Registration and Automatic Deployment

This scenario has two purposes. First and most important, the goal of this scenario is, from a functional perspective, to validate whether the prototype as a whole as well as the key FUSION functional blocks, effectively allow for efficiently and optimally deploying new instances of existing or new services across multiple zones in a particular FUSION domain.

As service deployment involves almost all key functional blocks at all layers of FUSION, it is the ideal scenario to validate and evaluate the interfaces, interworking as well as implementations in between and of each of those blocks.

Apart from the overall end-to-end functional evaluation, we will also focus on evaluating the role and effectiveness of key enabling FUSION technologies and concepts, such as lightweight containers to reduce deployment and provisioning time as well as runtime overhead, the benefit of having evaluator services, etc.

This will result in a number of scenarios, focussing on evaluating different features in each scenario.

- In a first scenario, we will manually register a FUSION demo service such as an EPG service or Shark 3D service and manually trigger the domain orchestrator to deploy the service in an execution zone, triggering first its corresponding evaluator service (if any). This should result in the application service to be deployed and accessible from a test client. This scenario tests the deployment of atomic services in an arbitrary execution zone. As part of this scenario, we will also evaluate the deployment of VMs compared with lightweight containers.
- In a second scenario, we will register a FUSION demo service and let the domain orchestrator automatically deploy and scale an appropriate number of instances in a number of execution zones.

3.2.1 Involved components, their functionalities and implementations

Component	Functionality	Implementation
Service Component	EPG and Streamer service components, with corresponding evaluator service(s) Shark3D game service	FUSION-enabled services
Domain Orchestrator	Service registration, service deployment, evaluator-service based service placement, etc.	Main FUSION prototype, implemented in Python and wrapped in Docker container
Zone Manager	FUSION service lifecycle management and intra-zone orchestration: session slots, evaluator services, etc.	Main FUSION prototype
Service Resolver	Allow client to automatically find an active instance with available session slots. API interaction with ZM regarding available session slots.	Simple FUSION prototype
Data Centre Adaptor	Implement abstraction layer between FUSION ZM and underlying DC management.	At least one FUSION prototype, including a minimal Docker and KVM enabled implementation

Client	Simple interactive client	Simple Java client

3.2.2 Means of verification

First, to verify each of the components, we will provide a number of simple service manifests, register them into the prototype deployment and subsequently deploy a number of instances in one or more execution zones. Subsequently, we will connect to the application services in a number of settings and validate whether we can connect to these services as well as how smooth these services are running. This will be presented in a live PoC demonstrator.

This demonstrator will involve a web interface to register new services, deploy new instances, see the status of the various components, the number of session slots, etc.

Second, we will evaluate the total time it takes from deploying a new service and until the corresponding locator is announced and propagated throughout the service resolution layer. We will compare lightweight containers with classical VM based deployments.

3.3 Composite services

- Evaluate three scenarios:
  - Static service graph deployment (intra/inter-zone)
  - Dynamic service instance graph deployment
  - Multi-configuration service components (for better resource utilization)

The third scenario covers the multi-configuration feature of (composite) service components, enabling a better resource utilization as well as faster virtual scaling of new service instances across existing instances of common service components (see Deliverable D3.1 for a detailed explanation of multi-configuration service components). Specifically, we will leverage the EPG and streamer service components in different configurations and demonstrate as well as measure their impact on service deployment.

3.3.1 Involved components, their functionalities and implementations

Component	Functionality	Implementation
Service Component	EPG and Streamer service components Shark3D dashboard service	Multi-configuration enabled implementation
Domain Orchestrator	Service registration, service deployment, evaluator-service based service placement, etc. for composite services	Main FUSION prototype
Zone Manager	FUSION Service lifecycle management and intra-zone orchestration: session slots, evaluator services, etc.	Main FUSION prototype
Service Resolver	Provide (optimal) endpoint towards either composite service or for one component to automatically find the other components with which to connect to.	Basic FUSION prototype for evaluating basic functionality Advanced FUSION prototype for evaluating efficiency
Data Centre Adaptor	Allow adding/removing a new service configuration to a running instance.	Main FUSION prototype

Client	Simple interactive client	Simple Java client

3.3.2 Means of verification

We will validate the multi-configuration concept by implementing this capability in the main FUSION prototype implementation. Specifically, we will extend the Zone Manager and DCA for supporting this feature as well as extend the manifest specification to be able to signal which service components support and enable this feature for particular application services. The functional evaluation will be showcased by sending a request from a client for a composite service that comprises components developed by Alcatel-Lucent and Spinor.Inside a single zone, the zone manager will flexibly combine instances of both services with available session slots.

From a performance perspective, we will measure and compare the infrastructure capacity reduction that can be realized for a given demand pattern by adding new service configurations to existing instances, compared to an approach where for each composite service dedicated instances are deployed.

3.4 Service Placement Optimisation

Several execution zones will be set up in the testbed that vary in terms of:

- Capacity: one large execution zone, representing a central cloud provider; several small execution zones with limited capacity, representing local processing capabilities in access networks.
- Network location and performance: the underlying network topology and performance (latency, bandwidth) will be configured in the Virtual Wall environment to deliver different network performance characteristics to the set of end-user locations.
- Costs: the fixed and incremental costs to instantiate service instances in an execution zone.

Two services: Advanced EPG and Thin client 3D game will be defined in terms of the usage patterns (number of users, their location and service invocation times), the target performance (maximum latency, minimum throughput), and the total deployment budget.

The service placement logic will identify the execution zones to run the services and the number of instances required in each location to meet predicted demand within the total budget.

3.4.1 Involved components, their functionalities and implementations

Component	Functionality	Implementation
Application Component	Advanced EPG Thin client 3D game	Application components implemented in Docker containers.
Orchestrator	Service Placement Algorithm	Plug-in to main FUSION orchestrator prototype implemented in Python.
Zone Manager	FUSION service lifecycle management functions to allow automated instance deployment in selected execution zone.	Main FUSION prototype.
Service Resolver	Minimal implementation to select an available service	Simple FUSION prototype.

	instance.	
Client	Invoke service and report on overall service performance.	Dummy clients as per main FUSION prototype.

3.4.2 Means of verification

First of all a single service will be deployed, aiming to show that the performance targets for that service are met, within the execution zone capacities and the deployment budget.

The second service will be deployed and the placement logic will attempt to optimise the location of the service instances, given that some execution zone capacity has been consumed by the first service. This may result in a sub-optimal deployment.

Both services will be deployed at once showing that trade-offs can be made between maximal performance and minimal cost to allow both services to meet performance targets while keeping within the constraints of cost and execution zone capacity.

By relaxing or tightening total budget for instantiation per service or minimum service performance targets we will show that service placement decisions can differ.

3.5 Session Slot Service Scaling

In this scenario, we will demonstrate the use of load information reported by services to trigger scaling decisions. Session slots and service holding time are proposed by the FUSION consortium as an abstraction of application-specific load metrics and heterogeneity of the underlying infrastructure, both intra-zone and inter-zone.

The demonstrator scenario will be as follows:

- Two services from different providers have been registered and pre-deployed on two different execution zones (one replica per zone). One can consider this as the outcome of demonstrator “Service Registration and Automatic Deployment”
    - Services: EPG (Alcatel-Lucent) and Shark 3D (Spinor)
    - Replicas running in at least two different execution zones. After all replicas have booted, they each report a different number of session slots and/or expected service request time (determined through evaluator services, or a direct mapping from the amount of resources allocated to the execution container)
- As user load is generated, the number of available session slots visible to the service resolver fluctuates. In this step of the demonstrator, we exclude scaling (both intra-zone and by the orchestrator). If the number of session slots is depleted in one zone, all service requests are directed to the replica in the other zone.
- We now enable scaling in a single zone. As more requests arrive, the number of instances is scaled up when the number of available session slots drops below a predefined threshold. When the number of available slots goes above a predefined threshold, some instances are been marked as “decommissioned”. These instances will not receive new requests, but they are only be shut down when the last pending request has been served.

3.5.1 Involved components, their functionalities and implementations

Component	Functionality	Implementation
Application Component	EPG service	Closed-source, provided by ALU in a Docker container and using

	Shark3D Rendering service	the FUSION API. Closed-source, provided by Spinor in a Docker container and using the FUSION API.
Zone Manager	Intra-zone scaling decision	Integrated with HEAT in OpenStack
Zone Gateway	Aggregates monitoring information over all service replicas and injects the information in the service resolution plane and to the orchestrator.	VM/container running in the zone Zone monitoring infrastructure
Orchestrator	Implements deployment and scaling decisions.	One per domain.
Client		Dummy client, provided by ALU

3.5.2 Means of verification

A GUI will show the following information:

- Number of session slots per service replica. This will be used to demonstrate intra-zone scaling, as well as the outcome of the evaluator service.
- Aggregated number of session slots/instances per zone, as visible to the service resolver. This information might lag the actual information in the zone; for scalability reasons.

3.6 Evaluator services

An evaluator service is a service which is implemented by software vendors but then used by FUSION itself to score possible instantiation locations of a service. A evaluator service is closely related to a specific service, and is usually implemented by the same software developer as that service. For example, FUSION may use an evaluator service for a game service to check if a particular execution zone supports the GPU features required by that game service.

Note that the main motivation for evaluator services is not to replace static manifests, but to avoid core FUSION components to understand all possible requirements of services. For example, while it would be possible to describe GPU requirements (e.g. minimal shader model version) in a manifest, implementing this within FUSION would make service vendors depending on implementing support for their requirement into FUSION, for example if new hardware is available or features must be tested in a different, service-specific way. Technically a evaluator service may work very well with configuration parameters or manifests, as long as the code interpreting them are not hardwired in the core of FUSION.

3.6.1 Involved components, their functionalities and implementations

As part of the demonstrator, we will implement a evaluator service for the Shark 3D based 3d service component, which requires a GPU. As described in section 2.3.3, a likely option is to implement the evaluator service for a Shark 3D based service component also based on Shark 3D.

Component	Functionality	Implementation
Application Component	EPG evaluator service	Closed-source, provided by ALU in a Docker container and using

	Shark3D rendering evaluator service	the FUSION API. Closed-source, provided by Spinor in a Docker container and using the FUSION API.
Zone Manager	Intra-zone deployment decision	Integrated with HEAT in OpenStack
Zone Gateway	Aggregates evaluator results	VM/container running in the zone Zone monitoring infrastructure
Orchestrator	Implements deployment decisions.	One per domain.
Client		Dummy client, provided by ALU

3.6.2 Means of verification

The two means of verification are:

- FUSION must deploy a Shark 3D based service only on servers which have the requirements checked by the evaluator services for Shark 3D, which mainly means checking for the precence of a GPU. Otherwise FUSION might try to install a Shark 3D based service on a server which does not support running that service successfully, resulting in an run-time error, which must not happen.
- The evaluator service provides FUSION information about preferred deployment options, for example that one server is better suited to run a particular service (e.g., a Shark 3D based service) than another because of better GPU features, resource characteristics or overall better runtime behavior. While the service could run on both servers, FUSION needs the help of the evaluator service for making a better decision. Whether this scenario can be verified by the demonstrator is not clear yet since it depends on the possibility of having multiple different GPU-powered servers available in one of the testbed, which currently is not the case.

3.7 Heterogeneous cloud environment

We will demonstrate the impact of heterogeneous environments and demonstrate how various FUSION layers take this into account in a few dedicated test scenarios and setups. Specifically, we will demonstrate the use and effectiveness of service metrics such as session slots as well as evaluator services for efficiently deploying services with particular requirements on hardware infrastructures with particular capabilities.

- We will demonstrate how particular services can exploit particular hardware capabilities to maximize the number of available session slots per type of environment;
- We will demonstrate how evaluator services can be intelligently deployed in particular hardware and software environments for efficiently selecting the optimal environment for a particular service, without having to deploy instances of all evaluator services in all possible environments;
- We will demonstrate the impact of various platform optimizations such as lightweight virtualization, NUMA-pinning and real-time guarantees for providing better QoS towards the FUSION applications;

- We will demonstrate how the underlying cloud platform can leverage these FUSION metrics for optimizing how particular services can be optimally deployed on the underlying infrastructure (improving the overall density and stability of deployed services), and how this (in its turn) is reflected in a higher efficiency score due to better overall efficiency and predictability, provided by the underlying optimizing platform.

3.7.1 Involved components, their functionalities and implementations

Component	Functionality	Implementation
Service Component	EPG and streamer service components Various evaluator services	Prototype service components
DCA	Platform-aware physical DCA prototype, optimizing the deployment of services on particular physical environments Non-optimized DCA implementation, which will be used as a baseline	Prototype implementation of a platform-aware DCA, provided by ALU
Zone Manager	Efficient (evaluator) service placement based on evaluator service feedback Agnostic zone manager, assuming a homogeneous cloud data centre	Main FUSION prototype
Orchestrator	Evaluator-service based placement Agnostic orchestrator, assuming a homogeneous distributed cloud	Main FUSION prototype
Client	Simple interactive client	Simple client

3.7.2 Means of verification

The overall goal of this scenario is to evaluate how effective the various FUSION concepts and layers can handle service, platform and infrastructure heterogeneity. This will be done by measuring different types of metrics of both the applications and platform. Specifically, we will create a GUI, depicting live metrics such as:

- Application performance metrics, such as number of supported session slots;
- Application QoS metrics, such as average and worst-case application latency;
- Infrastructure performance/efficiency metrics, such as power efficiency;
- Platform QoS metrics, such as scheduling latencies;

We will verify this scenario by creating two main FUSION setups, namely a non-optimized platform-agnostic FUSION domain and zone, where all execution zones as considered to be similar to each other with respect to resource capabilities, and a heterogeneous-aware FUSION domain and zone, where evaluator services, session slots and lower level optimizations are combined to drastically improve overall efficiency and QoS.

We will deploy an EPG service each of the setups, connect an application client to each of the services, and evaluate the overall QoE of the application in terms of responsiveness, quality and overall smoothness.

4. FIRST PROTOTYPE DEPLOYMENT

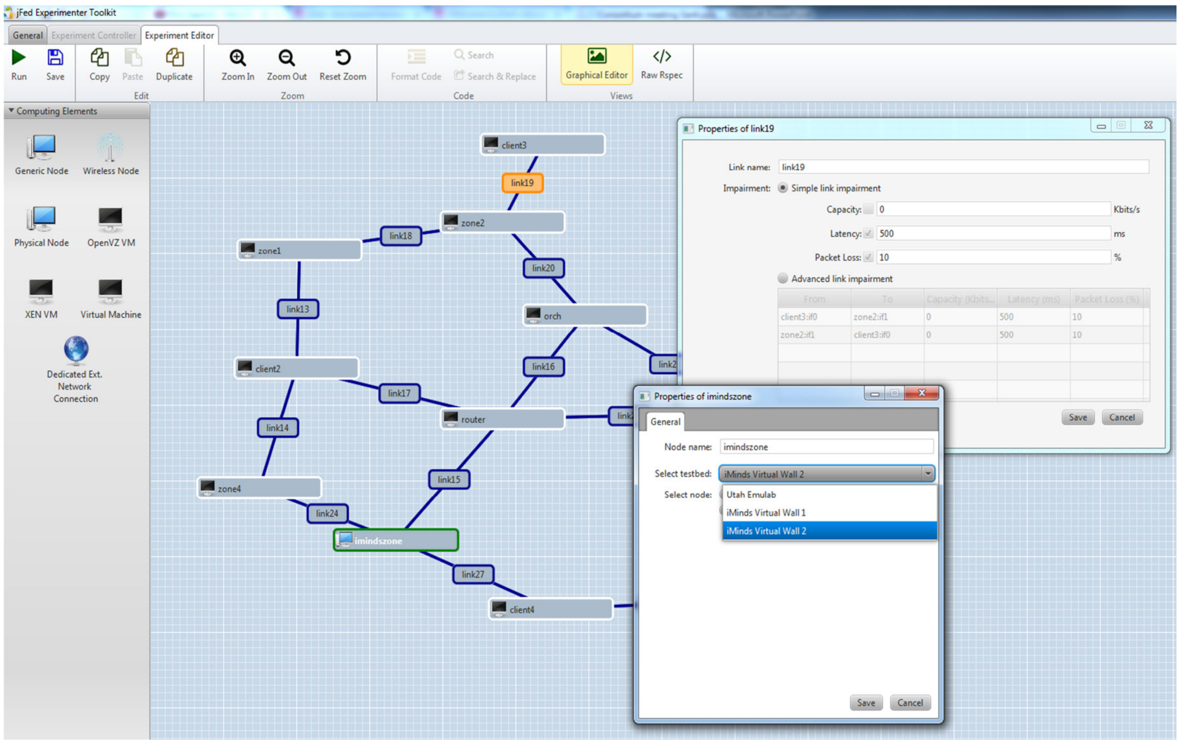
4.1 Virtual Wall Deployment Environment

To evaluate the FUSION components and architectural design, we are building an integrated prototype using the jFed [2] suite for testbed federation. JFed connects testbeds located on different geographical locations such as the iMinds Virtual Wall, PlanetLab Europe and Utah Emulab.

The deployment process goes as follows: first a user creates the network topology using either a configuration file or a web-based GUI. Next, the experimenter can operate each node through remote login such as ssh.

In the current stage, the FUSION integrated prototype is being build using jFED on a single testbed, namely the Virtual Wall [1] testbed. The iLab.t Virtual Wall facility is a large scale generic test environment for advanced network, distributed software and service emulation and evaluation, and supports scalability research.

Using the Virtual Wall we are able to quickly emulate any network topology of choice interconnecting a number of execution zones. Experiment deployment is scripted using a so-called *Rspec* file. First, the user defines the network topology in this *Rspec*, using either a GUI or his own script.



Properties of link19

Link name: link19

Impairment: (●) Simple link impairment

Capacity: ☐ 0 Kbits/s

Latency: ✓ 500 ms

Packet Loss: ✓ 10 %

( ) Advanced link impairment

From	To	Capacity (Kbits...	Latency (ms)	Packet Loss (%)
client3:if0	zone2:if1	0	500	10
zone2:if1	client3:if0	0	500	10

Properties of imindszone

General

Node name: imindszone

Select testbed: iMinds Virtual Wall 2

Select node:

Utah Emulab

iMinds Virtual Wall 1

iMinds Virtual Wall 2

Then, the user specifies in the experiment script the software to be installed on each node: zone gateway, orchestrator, service resolver, etc. Once the nodes, edges and their respective characteristics (node operating system, edge bandwidth, edge latency, …) are set, the experimenter starts the deployment process which is completed in a matter of minutes. Scripts are available to provision nodes as zone gateway, orchestrator, service resolver or client.

To facilitate the deployment process and minimize the required manual intervention, we developed a configuration generator similar to the one described in 4.2 (REFERENCE). First, we generate a network topology using a network generator. Next, we assign FUSION functionality to each node; execution zones start up with the required functionality to process requests, each service resolver builds its own resolution and forwarding table and waits for incoming requests while client nodes contain request generators. Last, we generate a configuration file containing the above mentioned information and use this to deploy our setup on the jFed testbed.

Currently, we have a running prototype for a single domain, comprising:

- 1 orchestrator
- 1 service resolver
- 5 execution zones (4 emulations, 1 zone running OpenStack)

Working use case scenarios are:

- Registration and deployment of services
- User queries, following a Poisson distribution for the interarrival time between requests
- Instance availability update from execution zones to the client

4.1.1 Next steps

The Virtual Wall uses virtualization to emulate a large network but it is only one testbed on one location. As a result, FUSION can not use location-aware algorithms when running on the Virtual Wall alone. To overcome these limitations we use the jFed **Invalid source specified.** testbed, a suite for testbed federation. Similar to the Virtual Wall, jFed creates topologies using a configuration file containing the network topology and optional functionality to be deployed on nodes. Nodes can now be deployed over multiple testbeds located in different geographical regions, each running a different configuration.

4.2 Orange Datacenter Deployment Environment

4.2.1 Current setting

One possible deployment scenario for FUSION that is interesting from the point of view of network operator assumes the use of mini-data centres integrated with the network infrastructure of the operator in selected points-of-presence. Orange is considering this kind of integration as one of the strategic options for migrating its infrastructure under the heading of Next-Generation PoP (NGPoP). The main underlying idea for NGPoP in case of Orange is to organise it relatively close to clients around collocated access/aggregation infrastructure for both wired and wireless access technologies. Within FUSION experimentation work, Orange is planning to mimic a simple NGPoP setting using its laboratory infrastructure as depicted schematically in Figure 7.

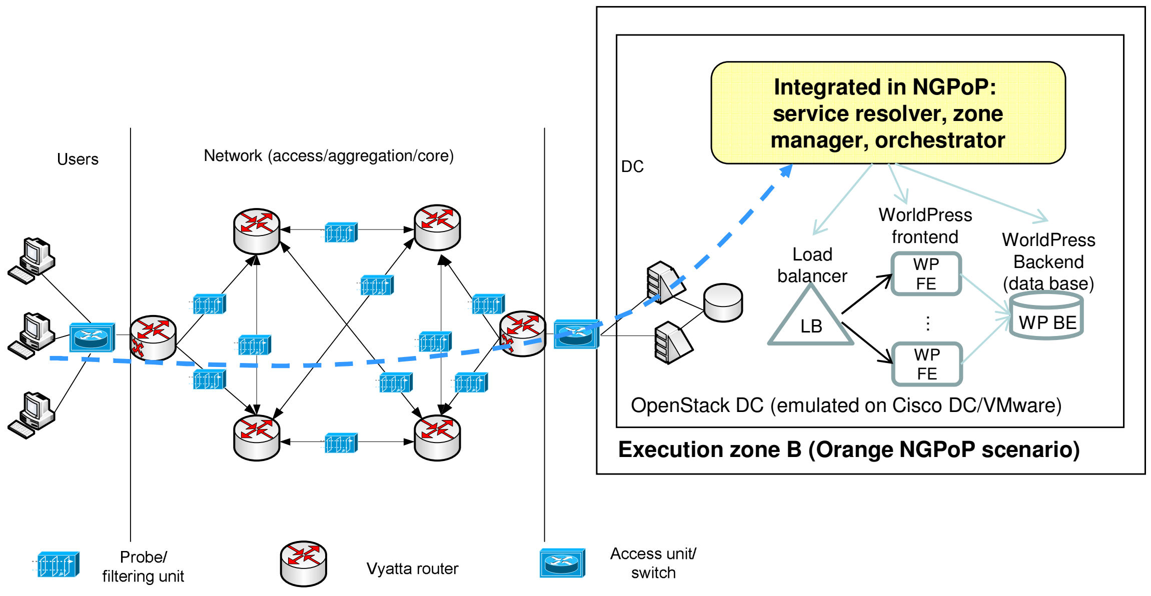

Integrated in NGPoP: service resolver, zone manager, orchestrator
Users
Network (access/aggregation/core)
DC
WorldPress frontend
WorldPress Backend (data base)
Load balancer
LB
WP FE
WP FE
WP BE
OpenStack DC (emulated on Cisco DC/VMware)
Execution zone B (Orange NGPoP scenario)
Probe/ filtering unit
Vyatta router
Access unit/ switch

Figure 7 NGPoP-based FUSION deployment.

The OpenStack data center is currently emulated using Cisco platform running VMware. The role of servers managed by OpenStack Nova perform virtual machines enabled by VMware that run run KVM. These machines are used to host target VMs that perform as applications managed by FUSION. Although this “nested virtualization” setting may pose performance restrictions for certain types of experiments, nevertheless it can well support functional tests including many experiments with service resolution and orchestration. For more advanced test in the future, migration to bare metal OpenStack is envisioned.

Current experiments covered integrated service routing and orchestration assuming no specific zone manager capabilities to be available. Therefore direct access to OpenStack IaaS was used based on open REST API provided by OpenStack (http://developer.openstack.org/apiref). Current implementation of our integrated service router/orchestrator) is based on Java using SDK Apache jclouds. The latter provides access to the functions of OpenStack API.

4.2.2 Next steps

Next steps include integration of network awareness into the resolution component which will require collection of respective data from network infrastructure. The latter is based on Vyatta routers and we envision SNMP-based access to selected measurements and CLI-based or equivalent access to forwarding information. Whether or not relevant information will be presented to resolution functions in ALTO format is a matter of future decisions, although we are aware of potential difficulties in implementing ALTO server and possibly limited impact such implementation would have externally. Another step planned is the integration of our infrastructure with Virtual Wall to enable common experiments with services being developed by other partners.

4.3 First prototypes of FUSION-enabled composite services

As a first prototype, a combination of the ALUB Java output client, based on the RFB protocol in combination with the Spinor Shark 3D engine were launched on different PCs and connected to each other. The main purpose was to demonstrate the interoperability between the two services building a composite service and the possibility to create new session slots inside the Shark 3D engine. By connecting the client to the Shark 3D software, a new viewer context was instantiated and the rendered output for this context was captured and transferred to the client using the RFB. In this sample, the context mapped to the same world as other exiting contexts, but it is just a matter of configuration, which parts should be instantiated on connection and therefore a whole new scene for each connection could be instantiated, too, which could then be enhanced to fully functional session slots.

4.3.1 Next steps

Next steps are using video compression and input channel.

4.4 Host environment deployment

For implementing and testing the various FUSION functionalities, APIs as well as initial demonstrator services, we developed and implemented initial prototypes of a FUSION domain orchestrator, a zone manager, and a host based data centre adaptor layer leveraging Docker and KVM as virtualization and isolation mechanisms.

Below a summary of the various actions we did:

- We first implemented a skeleton implementation of these core FUSION layers, focussing on implementing and validating the APIs detailed in Deliverable D3.2.
- Next we expanded the skeleton implementation into a working prototype to validate the overall FUSION functionality for registering, deploying and managing FUSION application services on top of a Docker based host environment.
- We embedded the various Python-based prototype implementations in Docker containers (as well as VMs) and deployed them on a Dockerized implementation of the host-based DCA prototype implementation. In other words, we used the DCA layer, which itself was wrapped in a container, to manage and deploy a FUSION zone manager, domain orchestrator and service resolver on top. This is illustrated in Figure 8.
- We also integrated several demonstrator service component prototypes, such as an EPG, streamer and virtual desktop component. For this, we embedded them in a Docker container (as well as VM), developed the necessary wrapper scripts for exchanging session slot and service configuration data with a FUSION zone manager, and created JSON-based manifests for providing initial service manifest descriptions. The latter were used for registering and deploying these services on the prototype.
- All these components subsequently were integrated and deployed on the virtual wall infrastructure provided by iMinds. This includes integration with their service scaling prototype based on an OpenStack HEAT environment.

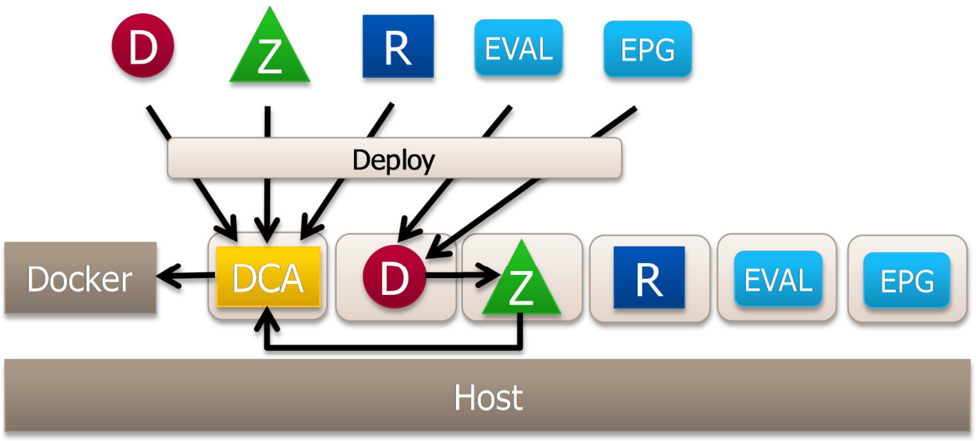


Figure 8– Deployment of prototype components on top of a Docker-based DCA environment

Using this initial prototype deployment on a Docker host environment, we tested a number of key FUSION scenarios, some of which will be described and evaluated in more detail in Section 5.1:

- FUSION core service deployment (i.e., domain, zone, service resolver) on the simple DCA implementation;
- Application service registration in a FUSION domain;
- Application service deployment, involving all components: domain orchestrator, zone manager, evaluator service, DCA, Docker daemon, service resolver, etc.;
- Integration of an evaluator service during service deployment;
- Automatically connect and interact with services deployed on the prototype using the service name and service resolver;
- Validating the functionality of the session slots as multiple clients connect and disconnect;
- Validating the aggregation capability of available session slots coming from multiple active instances of the same service type;
- Validating the multi-service configuration capabilities of the various application services for sharing service component instances across multiple services (composite or not);

5. FIRST PROTOTYPE EVALUATION

5.1 Service Registration and Automatic Deployment

In WP3, we provided already an initial design and functional implementation of the key FUSION orchestration layers, implementing both the REST APIs as well as their overall function. In this section, we describe how we validated and evaluated the current FUSION prototype and their end-to-end interactions.

Specifically, our initial evaluation of the FUSION prototype involves the registration and subsequent deployment of one or more FUSION services, as this incorporates almost all key FUSION components, as is depicted in Figure 9: the orchestration domain, zone manager, DC adaptor (as well as the DC itself), the evaluator services, session slots and monitoring, zone gateway, service resolver, and obviously the application services themselves. After a service is deployed by FUSION, user client can to connect to an available instance of the deployed service, only by providing the corresponding service name.

We first elaborate on the functional aspects, followed by an initial performance evaluation for a number of scenarios.

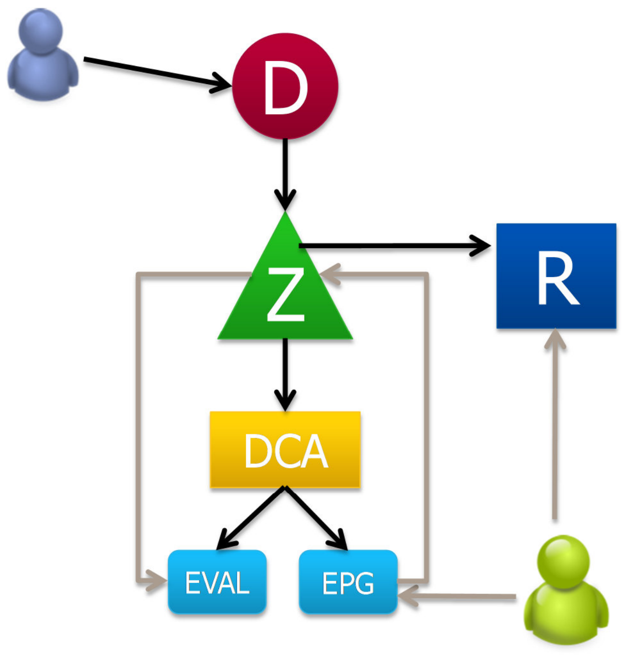


Figure 9 – Service registration and deployment scenario

5.1.1 Functional evaluation

We first implemented the service registration function, allowing a service provider to register a new service in a FUSION domain by providing a service manifest when registering that service to a specific FUSION domain orchestrator. This manifest is stored in the domain after which the service can be deployed.

To validate our initial end-to-end implementation of the service deployment function, we currently manually trigger the FUSION domain orchestrator to deploy a number of session slots of a registered service, which will trigger the sequence of steps described below in our working prototype:

1. We first trigger the domain orchestrator to start deploying a new instance in the domain by triggering the corresponding REST method (e.g., PUT /1.0/services/epg1.bell-labs.be/state for the EPG service);

2. The domain orchestrator implementation then first makes a preselection of feasible execution zones; in the current implementation of the preselection module, all registered zones are considered feasible; in a later prototype, we will integrate with service requirements, load and network monitoring data;

3. For each of the selected zones, the domain prototype will first check whether that service was already previously registered in that zone; if not, the domain orchestrator first registers the service into the zone;

4. Next, for all selected zones, the domain orchestrator implementation checks whether there is still a valid offer available for deploying that service in that zone. In such case, the cached offer will be reused and the zone manager will not be contacted. Otherwise, the orchestrator requests the zone manager to prepare a new offer for that specific service deployment request, passing along the necessary deployment and instantiation parameters.

5. The zone manager component in our prototype will now prepare a new offer. For services that have specified an evaluator service, the zone manager first looks up the endpoint(s) for various instances of that evaluator service, or simply leverages a generic evaluator service. In

case no evaluator instance is running yet, the current implementation returns an empty offer to the domain orchestrator. Later implementations could automatically deploy the evaluator service in the zone, assuming there is enough time available. Indeed, in the current implementation, the domain orchestrator can specify a deadline within which all offers need to be provided by the zone manager. Depending on whether the deployment was triggered by an on-demand or pre deployment scenario, there may be more or less time available for evaluation and deployment. Note that using light-weight containers facilitate fast deployments, including of evaluator services (see Section 5.1.2).

6. The zone manager prototype then triggers each evaluator service instance to make a proper service-specific evaluation and return a corresponding score.

7. All scores are collected and the best is chosen by the zone manager. For that score and evaluation request, a new offer is created and added to the zone manager data base for future reference. The corresponding offer is then returned to the domain orchestrator.

8. Based on all offers that were received (in time), the domain orchestrator makes a final selection and chooses the best zone(s). In the current implementation, only one zone is selected; however, in future implementations, we envision that multiple zones may be selected, each for deploying a subset of new session slots, especially for large-scale deployments, as some zones may only be able to deploy a portion of all slots at some price point or efficiency score.

9. The domain orchestrator now triggers each of the execution zones (in parallel) for deploying a number of session slots according to a specific offer.

10. The zone manager in each zone now will start deploying one or more instances. For this, it will trigger the DCA to deploy new instances on the underlying DC infrastructure.

11. In our current DCA implementation, this means either a Docker instance or KVM instances will be deployed on the DCA host, based on the manifest data, DCA capabilities as well as deployment parameters. The corresponding instantiation parameters are stored in the ETCD key/value store and passed to the application service instance.

12. While the container or VM is booting, the DCA returns a status code to the zone manager, which in its turn returns an appropriate status code to the orchestrator, all the way back to us (as we manually triggered service deployment).

13. In the mean time, when the application service is up and running and the new session slots become available, the service reports the newly available slots to the zone manager, which will subsequently inject them into the service resolution plane. Depending on the virtualization technology used, this may take seconds to minutes.

14. When the service resolver have been updated with these new slots (which could come from an entirely new service type), the user or client (or another FUSION service) can make a service request to the FUSION service resolution plane, which may return the endpoint of one of the newly created instances, after which the client can directly connect to the service of interest.

We already implemented the entire process as described above as such and thus allows already an initial full end-to-end deployment of the various already available FUSION services (e.g., EPG, streamer, evaluator, etc.), using most of the key FUSION functional blocks, which are all already communicating and working together already.

Note that if at any point in time some step fails, a corresponding error message is returned to the previous component, which subsequently either takes correcting measures or returns with an error message. Note also that we implemented a standard user/role based authentication mechanism in all prototype components so that only registered and authenticated users or software components can trigger particular REST API functions, or only can see a limited view of the system state (e.g., a service provider can only see its own services as well as all other publicly visible services from other providers).

In Table 1, a summary is provided of the status of various tests we have performed already on the prototype. As can be noticed, we mainly focused so far on the creation and runtime management. The cleanup as well as more automated functionality will be for the third year of this project.

Table 1 – Functional status of working prototype

Status	Test	Comments
PASS	Register users	Each FUSION component can successfully register new users and assign roles.
PASS	Register multiple zones	We can register one or more zones to a FUSION domain orchestrator.
PASS	Register services	We can dynamically register services using simple JSON manifests to a FUSION domain
FAIL	Unregistering objects	Full cleanup of state, including termination etc. has not been fully implemented yet. For example, to unregister a zone, all running instances of all services of that zone first need to be removed
PASS	Deployment of atomic services	We can already deploy atomic services manually (as described above)
PASS	Evaluator-based placement	Services can already specify their dependency on external evaluator services for making a zone placement decision. In the current implementation, it is assumed that such evaluator services already have been predeployed.
FAIL	Deployment of composite services	We are currently working on implementing deployment of composite services in a zone.
FAIL	Automatic scaling	Automatic scaling and deployment
PASS	Running Docker services	We support running services wrapped as Docker containers. In fact, the FUSION orchestration services themselves are also Docker containers.
PASS	Running VM services	We also support simple KVM-based services for running services.
PASS	Instantiation parameters	We support providing and passing service deployment and instantiation parameters, both to the evaluator services as well as providing them to the instances, which can use those for customizing their instantiation.

PASS	Session slots	Services can forward their active session slot information to the zone manager, which will inject it to the registered service resolver.
PASS	Service requests	We can successfully connect to particular services using only the service name, and interact with the active session (e.g. EPG).
FAIL	Session slot zone scaling	We did not integrate the session-slot based scaling in this working prototype yet.
PASS	Adding new service configurations	We can already add new service configurations to existing service component instances for hosting multiple services and their corresponding slots.
FAIL	Removing service configurations	We did not fully implement the removal of service configurations in service instances yet.

5.1.2 Performance evaluation

In this section, we discuss initial performance results of deploying a FUSION application service (e.g., evaluator or EPG service) in our demonstrator setup. Note that the absolute timings should be regarded as a lower bound, as in a more complex full-blown (distributed) FUSION implementation, these timings obviously could vary significantly.

First, in Figure 10, we show a break-down of the total time it takes to trigger the deployment of a new service in our prototype. Note that for these timings, the implementation follows the steps described earlier in Section 5.1.1, but do NOT include the time for the instance to be fully instantiated and available for incoming requests (see further). Also, for these tests, all prototype components (i.e., domain, zone, DCA, etc.) were deployed locally on the same host as Docker containers. Next year, we will also evaluate and compare with distributed deployments on the vWall.

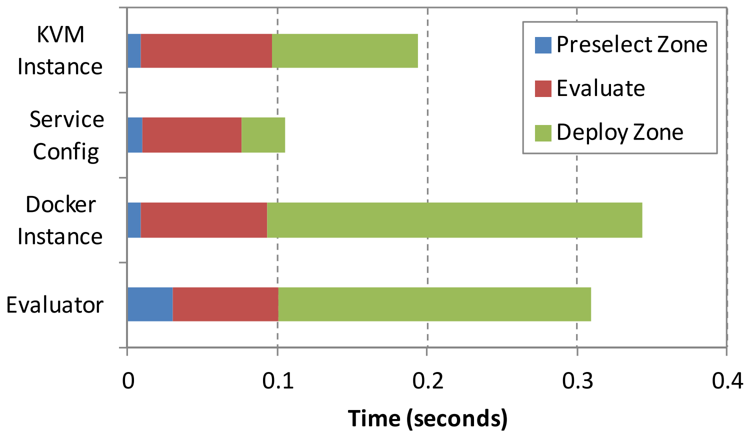


Figure 10 – Break-down of timing of deploying a new service in a domain

We demonstrate the deployment time for four service deployment scenarios:

- Deploying an evaluator service;
- Deploying a FUSION application service (e.g., EPG or streamer service) as a new Docker instance;
- Deploying a FUSION application service (e.g., alternative streamer service configuration) by adding a new service configuration to an existing Docker EPG service instance;

- Deploying a FUSION application service (e.g., EPG service) as a new QEMU/KVM instance;

From this graph, a few observations can be made:

- A first observation is that in the current implementation, deploying a new instance is very fast, taking less than half a second, even though there are already quite a few components and steps involved for implementing this deployment. Note that in this evaluation, we assume the VM/container images of the services are already readily available on the target host machine. If this is not the case, then the overall “Deploy Zone” time will increase significantly. Also, the current service evaluation step currently involves triggering a simple evaluator service that currently immediately returns a score without doing on-the-fly evaluations (which is not preferred anyway). In real deployments, we expect this evaluation to easily take up to a few seconds, depending on the amount of time the domain orchestrator allowed for making evaluations.
- Second, adding a new service configuration to an existing instance significantly reduces the total time for deploying a new instance. As in this case one can also assume that there will be no provisioning delay, this shows that being able to optimally reuse existing components and instances can significantly speed up domain-based deployment.
- Third, the KVM-based instantiation is faster than Docker-based instantiation. Note that de deployment time here does not include the time for the new instance to be fully operational, but only the time for the environment to be created. In case of Docker, it takes a fraction of a second for the container environment to be created. However, this delay could easily be mitigated in case the container creation would also be done asynchronously as with the VM creation.

As mentioned earlier, the timings above do not include the delay between a new service being instantiated and the service instance being fully operational, reporting available session slots to the zone manager. This delay is depicted in Figure 11, where we show the total instantiation time in case of three scenarios:

- Instantiating a new FUSION service in a Docker container;
- Instantiating a new FUSION service by adding a new service configuration to an existing FUSION instance running in a Docker container;
- Instantiation a new FUSION service in a KVM VM.

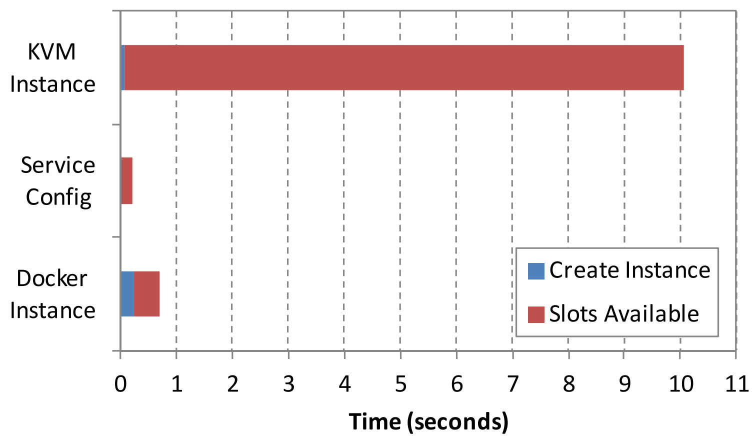


Figure 11 – Break-down of time needed for instantiating a new instance in a zone/DCA

The blue bar shows the time for creating the overall environment on the host. As mentioned before, we assume here also that the service container/VM images are already available on the target host machine. The red bar shows the time it takes for the application in the VM or container to start and report available session slots back to the Zone Manager. In case of KVM, this also includes the time to boot the VM. Obviously, the minimal time needed for an application to be fully operational can be very application-specific. However, for FUSION services, a design goal should be to keep this start-up delay as short as possible, especially for services that need to be able to scale out very rapidly.

Overall, a few conclusions can be drawn from these results:

- Firstly, booting a VM can quickly take a number of seconds (i.e., 10 seconds on this environment), and although this can be reduced by means of VM snapshots etc., there will always be a penalty on real cloud environments;
- Secondly, creating a new container and starting an application in a container instance can easily take less than one second, which can be especially advanteous in case of FUSION services;
- Thirdly, Simply adding a new service configuration to an existing instance reduces this delay even further, having new session slots available in only a fraction of a second. Note also that in this scenario, there will be no provisioning delay, and the start-up delay typically will only involve some internal configuration that needs to be done, rather than allocating all application resources and data structures from scratch (i.e., a hot boot scenario).

In summary, in this section, we already provided some initial performance results of our FUSION prototype for deploying FUSION-enabled application services wrapped in both Docker and KVM. In the final year of the project, we will expand these results by including more complex scenarios and evaluate these on more heterogeneous and distributed environments.

5.2 Service resolution

As has been mentioned in section 4.2, current evaluation experiments covered integrated service routing and orchestration. Actually, no specific zone manager capabilities have been implemented for this purpose so direct access to OpenStack IaaS capabilities based on open REST API was used.

The orchestrated service adopted is a simple WorldPress. The overall structure of this application and its relation to the simple orchestrator/service resolver is depicted schematically in Figure 7. The choice of WorldPress was motivated by our desire to start fast prototyping with selected functionalities like composite services and selected orchestration patterns. Of course, in the near future this application is planned to be substituted by services being deployed by other partners of the FUSION consortium. As can be seen, the application is built using the chaining pattern as of deliverable D4.2 (see section 2 therein) to allow for multiple components and load balancing. Related to this, but not shown in the figure is the adoption of shall scripting using CloudIinit tool in support of this chaining pattern.

We note that the decision for integrated resolution/orchestration to a great extent relates to the concept of NGPoP where we envision that FUSION components located in a given NGPoP will typically serve its own customers and optionally external customers. The validity of this assumption is of course for further studies, however, we believe the results achievable even under this specific scenario can still be valuable.

Next main steps that are planned are two-fold. First, there are plans to integrate our NGPoP demo with VirtualWall and also integrate into our NGPoP FUSION demo selected services currently being developed by other FUSION partners. EPG is a candidate application that is being considered for this role. The other dimension relates to testing the potential of selected concepts that are researched by FUSION (related mainly to orchestration and service resolution) to allow customers to build network of virtual WebRTC media-relays.

5.3 Load-aware service scaling and resolution

Both the orchestrator and the service resolver operate on the notion of session slots, an abstract load metric for long-lived sessions that plays a central role in FUSION. Figure 12 illustrates the different components and interactions in this integrated prototype. Service requests are resolved to the best available instance. In the current prototype, the request is resolved to the locator of the instance that has the lowest latency to the requesting user. Each instance reports its current slot count to the zone manager, who injects an aggregate report in the service resolution plane. Scaling mechanisms are demonstrated, both intra-zone and inter-zone.

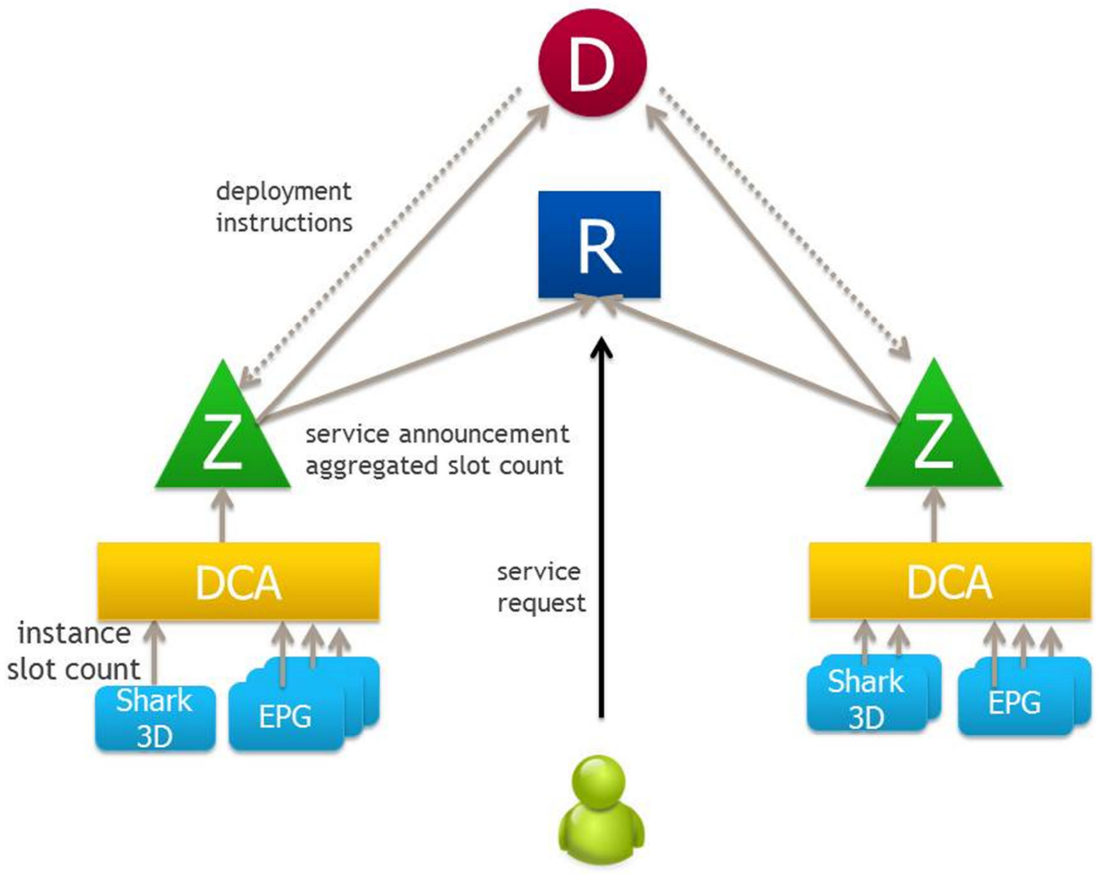


Figure 12 – Session-slot based scaling and resolution

5.3.1 Functional Evaluation

To demonstrate the interaction of slot-based scaling, we take a set-up where two services (EPG and Shark3D) are being registered with the domain orchestrator. These services have been extended by implementing the FUSION API needed to report session slots. Furthermore, there are two zones where services can be deployed. Initially, a single instance of the EPG service is deployed in zone A, and a single instance of the Shark3D service is deployed in zone B.

We assume a single network location from which service requests originate (e.g. a subnet). This originating point is closer to zone A in terms of network latency. If service instances are available in both zones, this essentially means that the service resolver will always resolve to the service endpoint in zone A and only direct users to zone B if the session slots are depleted.

The following sequence of interactions will be demonstrated:

1) Service requests are generated at regular intervals for the EPG service. The number of session slots reported by the single EPG instance decreases according to the number of users connected.

2) When the number of available session slots drops below a predefined threshold, the zone manager of zone A deploys additional instances.
3) When too many session slots are available, the zone manager will shut down a number of instances to reduce deployment costs.
4) If the maximum number of available session slots is reached, the orchestrator deploys the EPG service in zone B. As long as all session slots of the EPG service in zone A are depleted, service requests are resolved to zone B.

5.3.2 Performance Evaluation of current implementation

The current prototype is deployed on the via the jFED tool on the Virtual Wall. Each zone is deployed on a single machine (Dual INTEL XEON E5645, 24 GB RAM, 250 GB), that runs a complete OpenStack Icehouse version. The OpenStack implementation has been modified and extended with FUSION APIs to support session-slot based scaling.

Figure 11 shows the demonstrator GUI. The GUI shows the running services and offers the possibility to generate client requests The number of session slots is shown per instance, indicated by white squares. If a session slot is occupied, the white square will turn red.

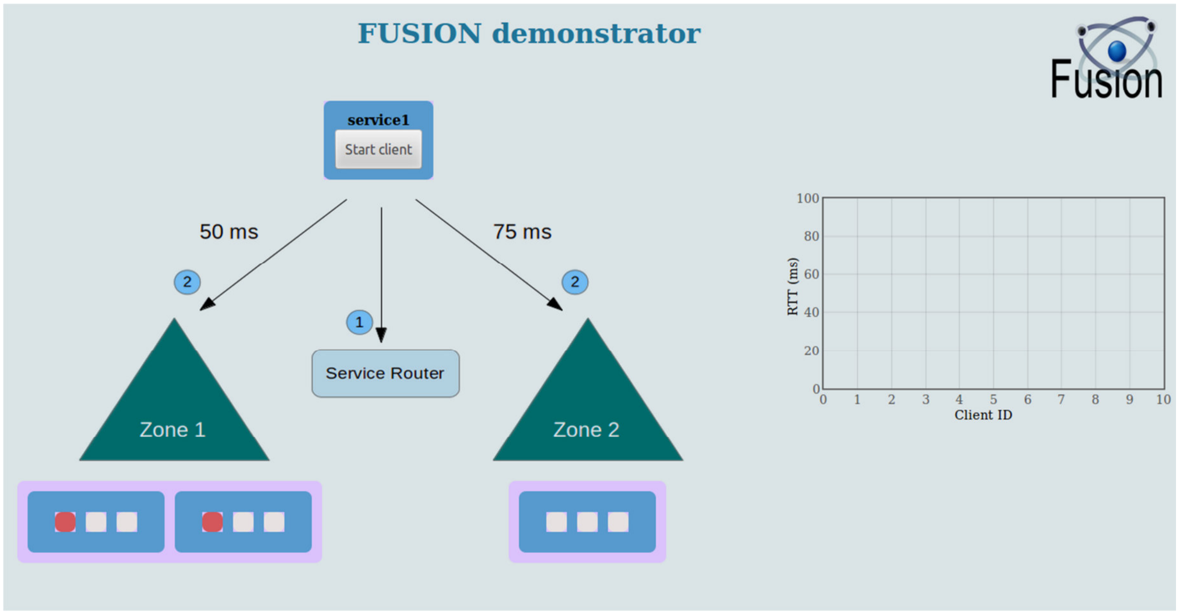


Figure 13 – Demonstrator GUI for slot-based scaling and resolution

The graph on the right hand side of the GUI shows the cost of a client connection. The service resolver uses the network latency between clients and zones for request resolution. In this case, we use the network latency as cost metric. The image shows that the last 5 clients that connected each had a cost of 50 ms. When a client connects to an instance in the second zone, which has a cost metric of 70 ms, you will see a spike at x=6 to y=70.

The services in the current setup are lightweight ncat echo servers running in VMs. These will be replaced by VM images of the EPG and Shark3D software.

The downside to this approach is that scaling VMs is remarkably slower then scaling linux containers for example.

5.3.3 Future plans

On top of that, several components (custom scripts, Heat and Ceilometer evaluators, …) have to work together, which also causes a delay. Therefore, switching to Docker containers and developing custom scaling software appears to be far more efficient.

Scaling results in multiple endpoints for a given service, which guarantees the availability of the service. Another advantage is that the service resolution algorithms can use a number of parameters such as session slots, response time, denial of service numbers, … for each of these endpoints to determine which one to return when a client request is received. Which exact parameters are to be used for these algorithms is up for discussion.

6. REFERENCES

[FFM14] FFmpeg, http://www.ffmpeg.org/, 2014.

[FV09] Vandeputte, F., Vampire parallelization toolchain, IWT Vampire project, Deliverable D.B.4.3, 2009.

[RT10] Richardson, T., The RFB Protocol, http://www.realvnc.com/docs/rfbproto.pdf, 2010.

[SHARK3D] Shark 3D, http://www.spinor.com/goto/shark3d_as_service.html, 2014

[X13] X264, http://www.videolan.org/developers/x264.html, 2013.

[YUV13] YUV4MPEG2 file format, http://wiki.multimedia.cx/index.php?title=YUV4MPEG2, 2013.