Future Service Oriented Networks

www.fusion-project.eu

Deliverable D5.3

Final report on service-centric system use case evaluation

Public report, Version 2.0, 30 July 2016

Authors

UCL David Griffin, T. Khoa Phan

ALUB Frederik Vandeputte

TPSA Dariusz Bursztynowski

SPINOR Mahy Aly, Michael Franke, Folker Schamel

IMINDS Pieter Simoens

Reviewers

Abstract This deliverable covers the final implementation of the prototypes developed within the scope of the project, the integration of the different components, which make up the FUSION platform and the results of the system evaluation. It discusses the architectural changes to existing interactive media engines for supporting FUSION. Next, it covers the features and implementation of the demonstrators, different deployment setups and system integration. The deliverable then reports the measurements conducted and the test scenarios performed to evaluate the FUSION system with a focus on deployment, resolution, session slots, composite services and the evaluator services.

Keywords Service Oriented Interactive Media (SOIM) Engine, Lobby Software, Dashboard, Game, Augmented Reality, Multi-User, Emulation, Evaluation

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

EXECUTIVE SUMMARY

This deliverable covers the final implementation of the prototypes developed within the scope of the project, the integration of the different components, which make up the FUSION platform and the results of the system evaluation. Aspects of the projects, which are not related to the prototypes, for example simulation results, are covered in other deliverables.

The first part of the deliverable is concerned with describing some concepts and architectural aspects for creating interactive media services and the transition phase, which was required to adapt the architectures of existing interactive media software to cope with the requirements of service oriented networking.

Next, it covers the features and implementation of the demonstrators as well as the possible deployment setups of the various service prototypes and the lobby software. The packaging and deployment of the services, the implementation of the lobby interface, which requests these services and the interaction between the FUSION components is then discussed.

The last part of the deliverable is concerned with the measurements conducted and the test scenarios performed to evaluate the FUSION system with a focus on session slots, composite services and the evaluator services.

TABLE OF CONTENTS

1. SCOPE OF THIS DELIVERABLE

This deliverable focuses on the final implementation of the prototypes developed within the scope of the project, the integration of the different components, which make up the FUSION platform and the results of the system evaluation. Evaluation aspects of the projects, which are not related to the prototypes, for example simulation results, are covered in other deliverables.

The first part of the deliverable is concerned with describing some concepts and architectural aspects for creating interactive media services and the transition phase, which was required to adapt the architectures of existing interactive media software to cope with the requirements of service oriented networking.

Next, it covers the features and implementation of the demonstrators as well as the possible deployment setups of the various service prototypes and the lobby software which requests services. The packaging and deployment of the services and the interaction between the FUSION components is then discussed.

The evaluation part of the deliverable is concerned with the measurements conducted and the test scenarios performed to evaluate the FUSION system with a focus on session slots, composite services and the evaluator services.

2. DESIGN OF SERVICE ORIENTED INTERACTIVE MEDIA ENGINES

This chapter sheds light on the redesign of existing interactive media software as a service-oriented media engine on the basis of a case study of the commercial Shark 3D software of Spinor [SHARK]. First, the related work is described in section 2.1, whereas in section 2.2, the workflow for creating scenes using graphs used by modern 3D media engines is illustrated. Sections 2.3 and 2.4 go more into implementation details to build the foundation of how sessions and session slots were implemented to optimize resource usage in Shark 3D. These results were published in [AFKS16].

The transition from single-user locally running applications to internet based delivery models has triggered new service-oriented architectures for software development [CCLE14]. Since engine architectures were typically built assuming single-user applications, therefore, it was assumed there is one single active 3D world and therefore the logic handling assets like geometry, textures and shaders was not adequately prepared for these resources to be shared by different logical users. Accordingly, Shark 3D was previously designed to have several rendering views supporting multiple clients but assuming only one 3D state which is rendered to all output windows. The emerging of cloud gaming drew new requirements. As many users should be handled by fewer instances of engines, this required a suitable architecture of 1:n relationships of the engine software components. To further illustrate this, related approaches are first discussed after which our architecture is explained.

2.1 Related work

Related work on developing cloud-native media applications is mostly situated in the domain of cloud gaming. Our discussion is classified into two categories: alternative developer frameworks and strategies for cloud resource sharing. The best known alternative development frameworks to Shark 3D in the gaming and media market are Unity 3D [UNITY] and Unreal [UNREAL]. Unity has included cloud support for running multi-player servers, as well as for management and organization tasks, but not for rendering in the cloud.

Other related work is found in the domain of cloud resource sharing. PS Now is a cloud gaming service covering technologies of Gaikai and OnLive which were both acquired by Sony. [PLAYS] hints at a separation between user sessions at the hardware level. A full hardware stack (CPU, GPU, memory) is assigned to each player, although realized on the same motherboard. NVIDIA GRID allows for efficient capturing and encoding of GPU output and GPU sharing. It offers two approaches [NVGRID]. The first approach applies a 1:1 mapping between GPU and OS. Although this is more efficient than offering a

full hardware stack per user session, there is still the overhead of running a separate OS instance per session. The second possibility is called shimming and provides isolation through light sandboxes on a single OS.

Hou et al. [NVGRID2] have integrated NVIDIA GRID GPU in the open source Gaming Anywhere cloud gaming platform. The main focus of the authors is to assign each VM its own GPU. Although the goal of the authors is identical to ours, our approach would allow higher optimization of resource consumption, especially texture memory, because of application-level support. A similar approach is followed in [SFL15], however by means of a new framework developed by the authors themselves. In [WXJ15], GPU memory is shared amongst different application instances by detecting already loaded content via hashing. This is performed on the driver level, whereas our approach implements this logic in the developer framework itself. This enables us to exploit not only more detailed information (e.g. resource comparison by file name), but also leverage on additional information unavailable at the driver level. For example, we can assign players having an almost identical set of resources to the same server even before starting the application, thus increasing the number of sharable resources. Another advantage of our approach is that the separation occurs in the application rather than by VM, resulting in the avoidance of duplication of main memory content for multiple instances.

2.2 Hierarchical node-based application definitions using templates

Assets, like 3D models, textures and sounds, play a key role with respect to resource usage optimization as they partly describe a virtual world. These assets commonly have a relatively large memory footprint compared to the program logic. Hence, today's software can possibly consume some hundred megabytes up to several gigabytes of hard disk space. On loading, such data as sounds and scripts has to be moved to main memory, while 3D models, textures and shaders to GPU memory. It might also be the case that some processing is required, for example textures often have to be decompressed and converted to GPU compatible format before moving them to the GPU memory.

In this section, we describe how Shark 3D organizes these assets in a tree structure and how this organization is adapted to cloud deployment.

2.2.1 Graph based editing

Typically, today's professional engines provide graphical tools for the designer to define the virtual world's structure as graphs. In case of Shark 3D, we use a tree for defining the scene augmented by references between nodes of different branches so that at runtime a directed graph is formed. These graphs often include nodes for physics behaviour, artificial intelligence, animations, visual elements and sounds. Moreover, like most other game engines, Shark 3D employs the concept of templates, which allows pre-defining parts of the graph such as complete characters and cars, and reusing it several times in the virtual world, often with the possibility of parameterizing these templates, e.g. by different logic scripts or models.

2.2.2 Runtime and editing

The runtime environment of the Shark 3D software consists of a very thin framework application which more or less loads configuration resources as well as instantiates and configures objects accordingly. This ensures flexibility since it implies that even the "basic" functionality of a game engine like instantiating a renderer is not performed automatically but is rather aligned with these configuration resources which are generated using the editing tools.

In the Shark 3D engine, the editing of scenes is done on a running instance of the engine by connecting it to the editing tools and delivering new configurations to the runtime in the event of changing the graph or node properties. From the runtime perspective there is no technical difference between editing and running the final application. How the graph is translated into runtime configurations is described later.

Any extensions and changes in behaviour necessary for editing (e.g. different behaviour or editor camera and input) is defined on a higher level using either different nodes or a generic socket-based communication protocol. Any changes made in the tree or to node attributes are immediately translated into configuration data chunks (the “outcome” of the translation process) and dynamically loaded by the running engine. The user can also modify or even remove any of the editing features by simply updating or disabling the respective nodes in the graph respectively.

2.2.3 Adapting graph based editing for cloud usage

Spinor chose a bottom-up approach to make the move from locally installed to cloud-hosted media software. This means that each node of the aforementioned graph, which defines the virtual world, was prepared to be cloud proof, with the idea that if each atom is prepared to be used in cloud software, so will be the resulting combination.

Furthermore, in contrast to many other similar software products, the complete functionality of the Shark 3D engine was reworked to be packaged into nodes that are added to the graph on a very fine level of granularity. As a result, there are specific nodes in the virtual world which define the texture managers, sound managers, etc. The end users therefore can adjust every part of the engine to their specific needs by rearranging the relevant nodes and their interconnections. The reason for this is that there is not only one single architecture for Service Oriented Interactive Media (SOIM) Engines, but depending on the specific situation (single-user games, multi-user games, training application, multi-media dashboard etc.) different engine architectures are needed. Since the core functionality of the engine was moved as well to nodes in Shark 3D, these are also accessible to high-level tools to make it easier to create custom engine architectures depending on the specific need. Pre-defined templates with default behaviours in Shark 3D on the other hand allow a quick start without giving up the flexibility to be adjusted later when required.

The following simplified example illustrates the differences. Given a game that can be played by multiple users in single or multi-player mode, it can be seen that sharing assets is valuable even in the single player mode, independent of the actual progress each player made in the game. This sharing involves running a single instance of the software and using the same pointers to memory for accessing textures, 3D models, sounds, etc. The only difference is on which level the logical separation takes place. In multi-player mode, the players share a common simulation of the virtual world, and only their avatars and views, i.e. their virtual camera and input is instantiated per user, while in single user mode the world simulation is also instantiated once per user but there is only one camera and input instance per world simulation.

From a Shark 3D perspective, so called producer nodes can be added to the scene tree. Upon sending a produce message to the runtime representations of these nodes, the whole subtree under the respective node is instantiated. A producer node can therefore be compared to the design pattern “factory”. The difference, which parts of the scene tree are instantiated once for all sessions and which parts are instantiated per session is now made by adding the respective nodes either under the producer node (instantiated once per session) or outside the producer node (instantiated once for all sessions). This adds a lot of flexibility, because the decision whether a node should be instantiated per session or not can be revised by doing a simple drag-and-drop in the tree window.

For the producer node to work it is necessary to add a semantic to the parent-child relationship in the tree, in this example that all nodes under the producer node are instantiated whenever the produce message is sent to the producer. There are also other examples for these semantics, it is therefore added as a generic concept to the Shark 3D editor architecture. The exact meaning of the relationship is determined by the types of nodes involved.

Another example of such kind of relationship is that of the texture manager node. To optimize usage of graphics memory, the so called texture manager nodes can be added to the tree, which will then act as a texture manager for all descendant nodes. Whenever a descendant texture node requests a

texture, the texture manager will be used to look up whether this texture already exists in graphics memory or must be loaded. Therefore, there exists an implicit (invisible to the user) relationship between the texture manager node and any descendant texture node.

More in detail, our implementation of the service oriented media engine uses the following specific concepts:

1) A node system organized as a tree
2) Each node can have properties
3) Depending on the specific node types, the direct or the indirect parent-child relationship between nodes can have a semantic meaning. This is especially interesting in combination with templates, because templates are “inlined” during compilation and therefore the logical relation of the inlined nodes is relevant, not the structure of the tree visible to the end user.
4) Depending on the specific node types, nodes can have cross references, which basically allow circular graphs (not only non-circular graphs as the node tree itself). These references can be used by a messaging mechanism to send and receive messages across the branches of a tree. For example, the instantiation of children of a producer node can be triggered by runtime instances of another node which is defined somewhere else in the tree by sending a “produce” message.
5) Nodes can define scopes for supporting the OO concepts of encapsulation and instantiations.
6) Templates provide a generic mechanism to re-use node sub-graphs. Templates can be nested and parametrized, so that re-using sub-graphs allows customizing them per usage.
7) The resulting runtime objects can be inspected and manipulated, for example to inspect or change the current translation or animation playback state.

One challenge implementing the last point was that the whole editor implementation including the tree and compilation tools is developed outside the runtime engine. As the runtime engine is used both for editing and running the end user application, any connection to the tools pipeline should be as least intrusive as possible so that removing the connection to the tools when packaging the deliverable version should have no effect inside the runtime engine. Therefore, the connection between runtime engine and editing tools uses a very narrow and generic interface that makes it necessary to keep representations of runtime objects in the editor tools.

Spinor developed these mechanisms already before using it for implementing service oriented engine architectures. However, these mechanisms allowed us to implement new nodes to specifically support the creation of service oriented engine architectures. This includes new nodes for managers (e.g. texture loaders), 3D states, view ports, relationships of these components and connection of 3D states and view ports to virtual cameras and users in 3D worlds.

2.3 Translation of definition graphs into descriptions of the run-time engine

One core lesson we learned over time for getting real flexibility for a node based high-level tool was the need for a generic translation process of the node tree into a run-time description for the run-time system, which goes beyond a simple 1:1 relationship.

The Shark 3D node tree offers the possibility of defining templates which can be used anywhere in the tree. Each usage of a template is “inlined” during translation of the tree thus leading to an outcome equivalent to a repeated insertion by the user of all nodes inside the template. Therefore, each node in the tree may be available multiple times, however each time the context may differ depending on where the template is used. This results in a 1:n relationship that has to be managed.

With this growing number of nodes, especially those used in (often nested) templates, it can be concluded that the translation time will be longer when done brute force. In order to shorten the duration of compiling the tree into a runtime description, sub-trees resulting in identical outcome are translated for once and reused. Nevertheless, since the outcome also depends on the context in which the node resides (in the sense that templates can be parameterized and there may be implicit references from nodes to direct or indirect logical ancestors in the tree), it can be noted that combining the 1:n relationship caused by multiple usage of a template is a complex task.

Nodes may also be instantiated at runtime, for example by using the producer node already mentioned. This adds a possible 1:m relationship for each of the above mentioned 1:n relationships. For editing and inspection purposes, the editor logic has to track all runtime instantiations of a node. This yields a two-step process of translating the node tree into run-time instances of engine components, as shown in the following figure.

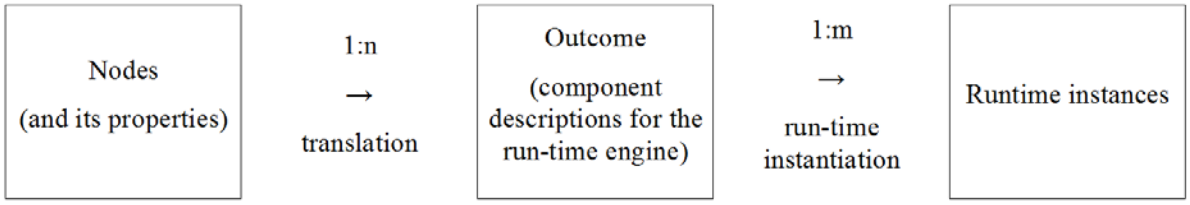


Figure 1. Steps for translating definition graphs into runtime instances.

The following figures show screenshots of our Shark 3D editor tools. The following windows are shown:

8) Node tree

9) Properties of the currently selected nodes. Besides other parameters, it has a cross-reference to another node named “View”

10) The outcome of the selected node. While it is displayed in text format, internally it is optimized by handling it in a binary format.

11) One of the run-time instances of this node. The list of “current runtimes” indicates that we are currently seeing the second of the n times m run-time instances, where in this simple case n times m equals 2. In the “chosen runtime ancestors” you can see how this particular runtime instance originates from the translation and the instantiation process of nested template nodes together with on-demand run-time instances. For example, the “Ports” part refers to a node which included a particular template at translation time, while the “2” part indicates a second instantiation at run-time. In other words, this windows shows you how exactly this particular run-time component instance originated from the translation and instantiation process of the original node tree.

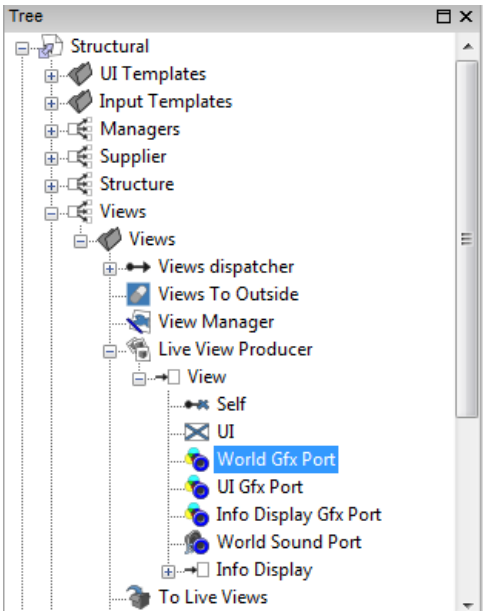


Figure 2. Nodes for instantiating of graph node trees

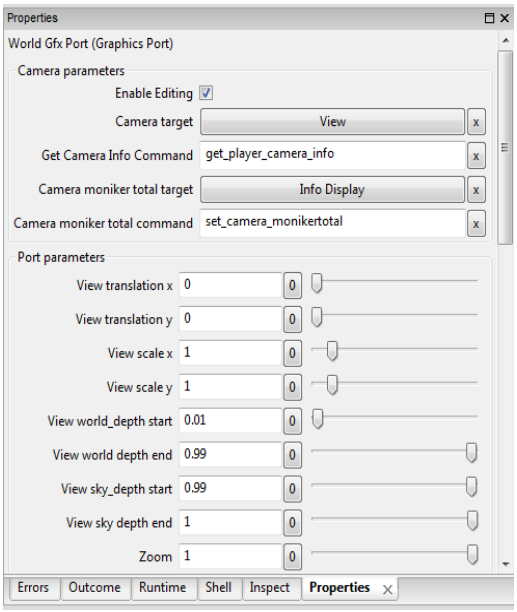


Figure 3. Properties of a node instantiated for creating a view port

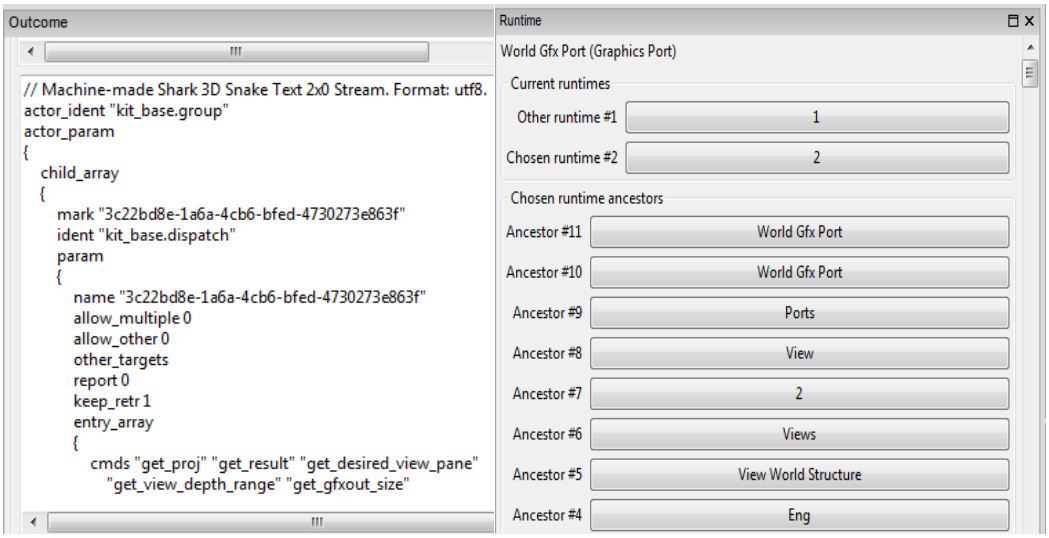


Figure 4. Multiple run-time instances of a view port node

2.3.1 Translation of the tree into a runtime description

The translation step includes the following operations:

- Resolving templates, which may be nested. This means that during tree traversal, the templates are “inlined” when they are used and the parameters are resolved.
- Resolving implicit references of a child to a direct logical ancestor. An example is a delegation controller node operating on a delegation node defined as direct ancestor.
- Resolving implicit references of a child to an indirect logical ancestor. An example is a component loading a 3D object into a 3D state defined in an indirect parent.
- A parent node collecting information from direct children. An example is a node responsible for instantiating a renderer instance demanding a reference to all viewports defined as child nodes.
- Creating the outcome descriptions of individual nodes or groups of nodes based on the properties and all relationships of these nodes.

One possible optimization approach that has been taken is based on the fact, that after a first complete compilation of the whole tree only incremental changes are likely to be made. This constitutes a rather challenging task in view of the fact that all the relations between the nodes (parent to child, child to parent, etc.) have to be taken into account when recompiling only parts of the tree.

2.3.2 Loading and instantiating the runtime description in the runtime environment

The run-time instantiation process involves the following operations:

- Loading dynamic libraries containing the code for different components.
- Instantiation and initialization of objects based on outcome descriptions. For each node in the tree normally a number of runtime objects is created. The initialization parameters are derived from node attributes defined in the editor and passed to the engine runtime using the configuration files.
- Resolving references to other objects defined by name in the outcome into efficient C/C++ pointers for direct access. This is mostly done at loading time but there are also situations

when this is done on demand (when the reference is actually needed) for flexibility reasons and allowing circular references.

- Instantiating engine components building a thread safe parallelized dependency graph based on delayed evaluation and other mechanisms for efficient execution of the engine functions.

This part of the run-time engine is implemented mainly in C++. The before mentioned connection to the editing tools is realized using generic Python bindings, custom Python bindings, an internal network based protocol and a resource watching mechanism. This mechanism reloads resources automatically upon change, including new outcomes of the graph translation process. Most of the changes made in the editor are therefore federated into the runtime system by reloading the respective configuration resources. There are only some exceptions to this rule where reloading the configuration would have a very poor performance.

2.4 Resource sharing using sessions and session slots

2.4.1 Session slots

The maximum number of possible sessions that can be run in one engine instance is limited by the amount of (virtualized) resources allocated to the engine. Besides depending largely on the specific application, the number of available session slots is volatile in a sense that it relies on the resources used by the running sessions. Therefore, it cannot be predefined but must be reported to the cloud management software on a regular basis (e.g. to the zone manager).

On the other hand, this indicates that the application setup defines which components must be instantiated separately for each user. This can be implemented by using different nodes in the tree, which guarantees flexibility that for each application, it can be determined which parts are to be instantiated per user and which parts are to be shared. Putting the texture manager node above the producer node that instantiates the 3D state for each incoming connection for example makes this texture manager a common ancestor for each instantiated 3D state which therefore makes the texture management (along with the texture resources) a shared component.

Therefore, the component-based architecture of Shark 3D engine can be used by the developer to define factories for instantiating session-specific node graphs, which access and use shared data (e.g. assets). Such shared data can then be used by independent 3D states that define the virtual worlds of the services which are, in turn, linked to different views that render the output streamed to the users.

2.4.2 Shared resources

Resources that are possibly shared between different sessions of the same application can be divided into several categories as follows:

On-disk resources: these include the program files and assets which are the graphics artwork as well as the sound effects and music used within games. Shark 3D offers a number of object types to store different kinds of assets such as 3D models (mesh data, skeleton data, skinning information) and textures which can be either static images or video textures.

GPU resources: As the memory on the graphics card is much faster, some resources are loaded to it once and can be used repeatedly. This includes the shaders as well as the vertex buffers which contain the vertices and further attributes of the 3D models drawn on screen.

CPU resources: excessive calculations need to be performed by the CPU especially for simulating physics, collision detection as well as sensor and trigger geometries which determine when certain events should be triggered. Additional computation power is also consumed to run the application logic that can be defined in C++ or different scripting languages.

2.4.3 Session slots implementation

In Shark 3D, the management components for these resources are provided as nodes so that the resources can be shared by all the 3D states which are created as children of these nodes. For each multi-player game session, a 3D state is used to define the virtual environment where the game takes place. Each player however should have a personalized 3D output as each sees his own perspective of the 3D world and should be able to control his character in the game. Therefore, even if the 3D state is shared, each player would have his view, i.e. his own input and output channel. Therefore, it is possible to split multi-user services into two components: the world simulation shared amongst all users of the same session and rendering services which are user specific. These are represented in the editor by the state and view nodes respectively.

As mentioned earlier, Shark 3D provides the possibility to create a factory for session slots necessary to allow resource sharing. In the editor, these factories, known as “producers”, can be used to define which functional components (e.g. unique 3D state) must be created for each session. This mostly relates to relatively lightweight C++ objects in contrast to shared data such as textures, sounds or other assets, which can be created and loaded only once for use by the different sessions.

In order to create multiple states and views, a “State Producer” and a “View Producer” are used, where each contains an “include node” which is a node type that references the appropriate template. The state node defines a new 3D state where the physics properties can be set. Starting with the State template, the main components can be seen in Figure 5. If both the world simulation and rendering services are running as one service on the same machine, then only one state would be sufficient for a session. Otherwise, for each rendering service performed on another machine, a new 3D state needs to be created and synchronized with that of the server. As the state has to be addressable via TCP, a “Network Root” node is added as a child of the State node. This network root gives the possibility to open a port to listen for incoming connections, if the state should be acting as the server. This can be achieved by sending a command to the “Network Listen” node to start listening on a certain port. The network root also contains a producer, to create a new socket for each incoming connection, represented by the “Network Link” node. When a client needs to join a certain session, a connect command is sent to the “Network Connect” node with the IP and port of the server. Whatever needs to be synchronized between the players in a session must be a child of the network root. To achieve this, a “World” capsule is introduced as a child to the network root so that the whole environment can be synchronized.

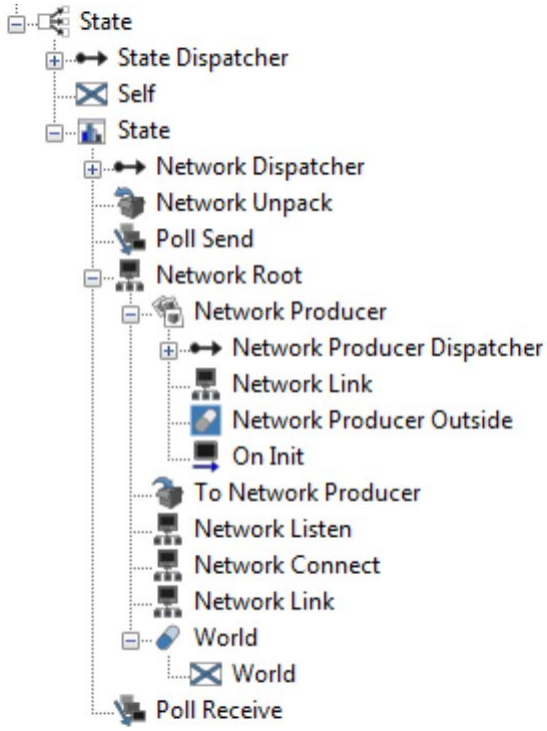


Figure 5. State template nodes

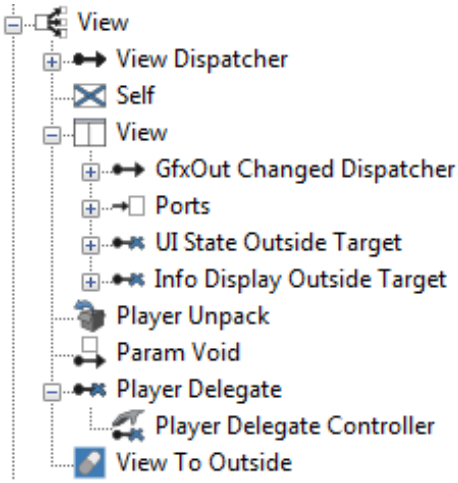


Figure 6. View template nodes

A new view must be created for each client once connected to a state. The view template used is shown in Figure 6 and it contains a view node which is responsible for rendering the output depending on the 3D state the view is bound to. This binding is done by sending the camera and input commands to the appropriate character within the 3D scene. Therefore, the view template also contains a "delegate" node to which the camera and input commands are sent. This player delegate node is bound at runtime with the character actor produced in the 3D state when a client has been connected to it. The demonstrators created by Shark 3D based on this architecture are presented in the following chapter.

3. APPLICATION PROTOTYPES

Prototypes for different application types were used as examples of real life applications that can make use of the FUSION platform and demonstrate its features. These were:

1) Advanced media services Electronic Program Guide (EPG) – section 3.1
2) Augmented reality – section 3.2
3) Thin client 3D game – sections 3.3 and 3.8
4) Media dashboard – sections 3.4 and 3.8
5) Chat service – section 3.5
6) Lobby software – section 3.6
7) Prototype Web UI – section 3.7

This chapter covers the full features and final architecture and implementation of each of these prototypes.

3.1 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 become increasingly important in the upcoming years 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, STB, etc.).

The role of FUSION for such application use cases 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.

3.1.1 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 7.

Figure 7. Screenshot of a 2D EPG service prototype.

3.1.2 3D EPG service

We also developed a 3D EPG service, which has more demanding/specific 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.

Two examples of a basic 3D EPG service that we implemented and integrated in the testbed, are depicted in Figure 8, and represent a rotating interactive cube and sphere, respectively, where people can browse through and interact with media content in various ways. Both represent simplified versions of possible novel 3D UIs for browsing through media.

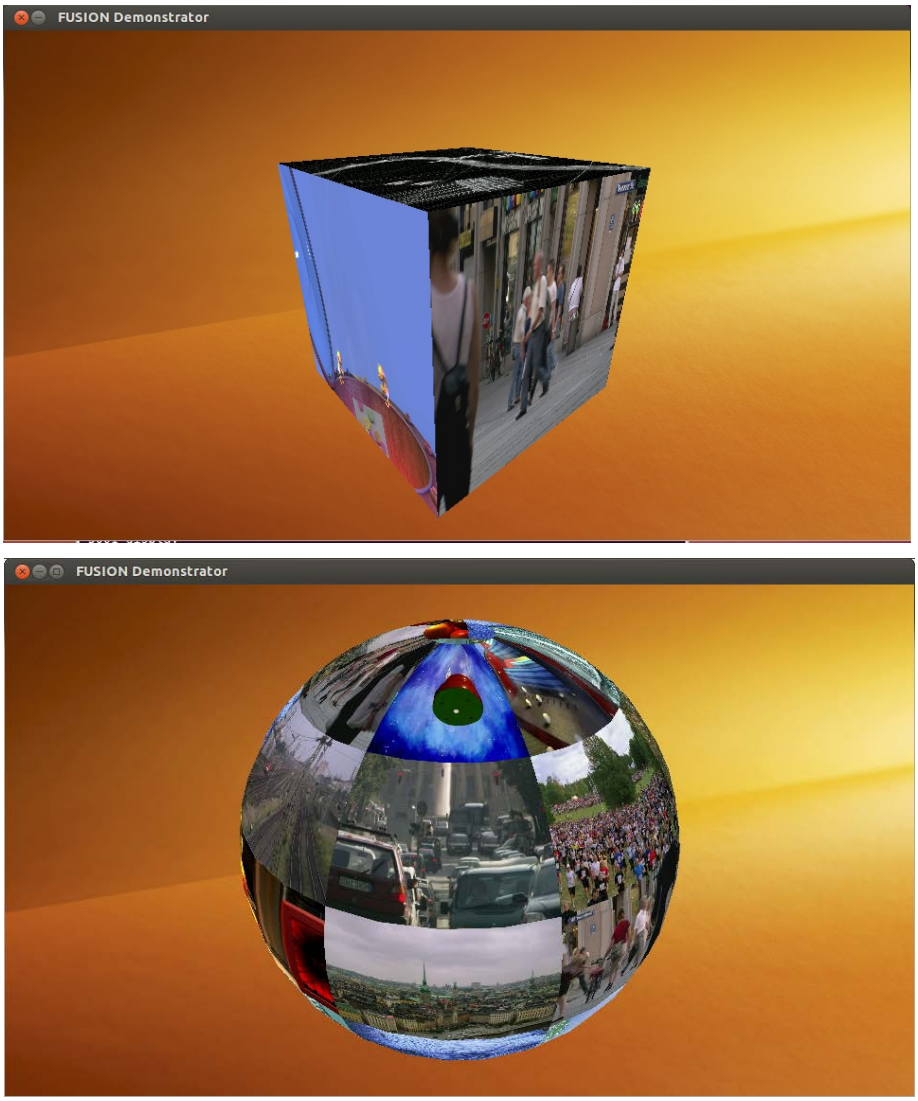


Figure 8. Screenshots of our 3D 'cube' and 'sphere' EPG service prototypes.

3.1.3 Features

This application serves as an example of a personalized single-user streaming service and has the following features:

- This service has been designed using a microservice architecture, where key distinctive functions are developed, deployed and managed as independent service components. Specifically, this service has been decomposed into a decoder, rendering and streaming component.

  Not only allows this to reuse service components across multiple (composite) service types, it also allows better mapping of individual service components onto their respective optimal hardware.

- To support this micro-service architecture, we implemented an efficient shared-memory based inter-service communication mechanism, allowing to efficiently exchange raw frames in between these micro-service components when deployed onto the same hardware environment (e.g., the same host or micro-server).

- We implemented the session slot concept, allowing a configurable number of independent interactive sessions to be active at the same time, sharing as many internal resources as possible.

- We implemented dynamic session slots, where the service components each internally keep track of the actual available resources, calculate the corresponding number of supported session slots and dynamically report session slot availability to the zone manager. Session slot availability can change due to changing actual resource usage for active sessions, as well as possible interference from other services sharing the same physical resources.
- On top of the session slot concept, we also implemented the multi-service configuration concept in all components (i.e., service aliasing concept), allowing different service types to be leveraging the same resource instances. For example, a low-level EPG instance could be used for handling different EPG services (e.g., different resolution, QoS level, features, etc.)
- We implemented a 2-stage evaluator service for these service components, consisting of both a stage 1 evaluator probe for assessing the runtime environment for a particular service deployment request, as well as a stage 2 analyser probe that helps determining the most cost-efficient environment across all zones.
- We integrated an application monitoring into the components, where each service component can report application-specific data to some monitoring framework. This could be used for example by the service provider itself, but also by the heterogeneous cloud platform to learn about the efficiency of a service deployment in some runtime environment.

3.1.4 Architecture

This application has been designed as a micro-service application, consisting of a number of specialized service components. A generic high-level view on the application architecture is depicted in Figure 9.

In this generic architecture, an EPG coordinator component is responsible for finding proper EPG rendering and streaming component instances, allocating some session IDs, and returning them to the client who then can connect directly to both components. This architecture allows to send feedback events directly to the EPG rendering component, and to receive the resulting interactive video stream from the streaming component. The video decoders, being stateless, can be selected implicitly by the rendering component in this use case.

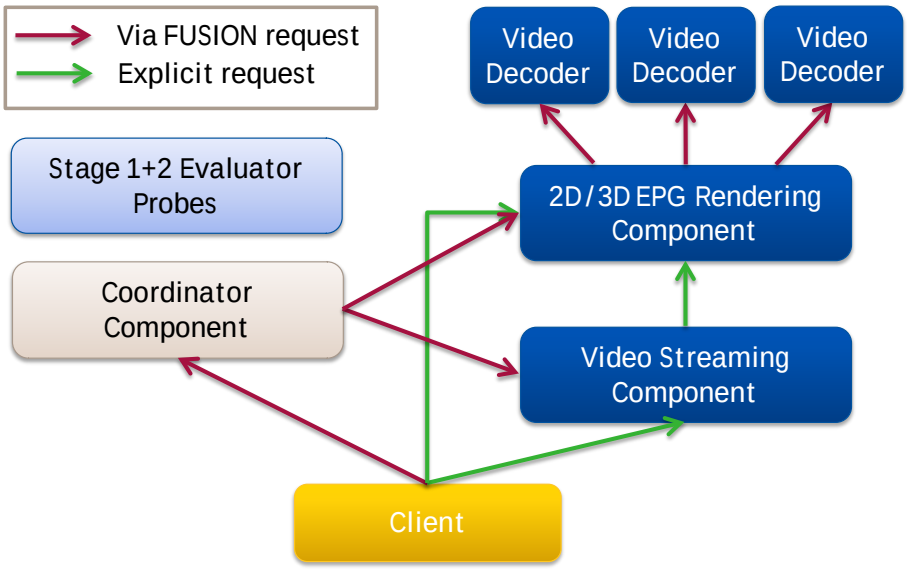


Figure 9. Generic High-Level EPG Application Architecture.

Next to this generic architecture, a more simplified application architecture is depicted in Figure 10. In this architecture, there is no separate coordinator component, and all communication channels are in line, along the path of direct FUSION service requests. In this simplified version, the feedback channel for example is sent through the streaming component to the rendering component.

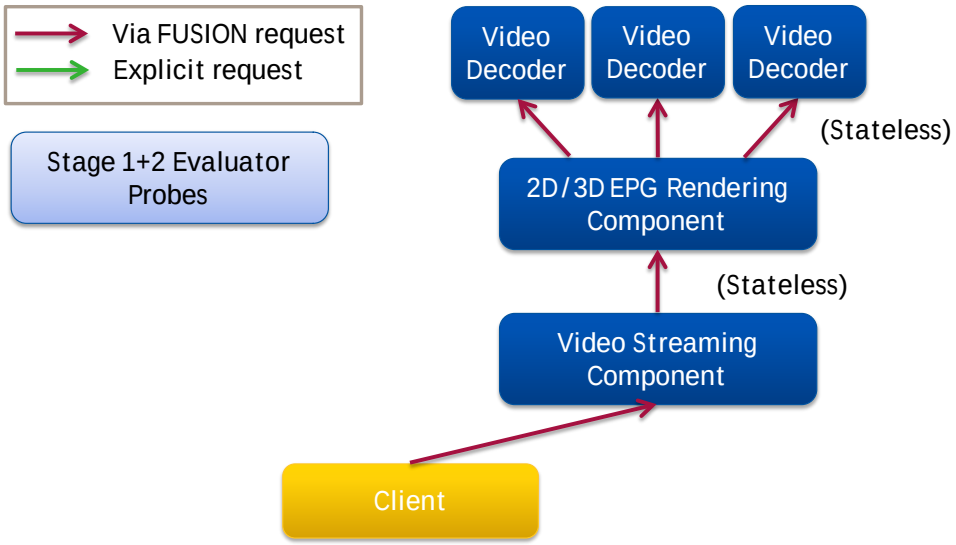


Figure 10. Simplified High-Level EPG Prototype Application Architecture.

A key trade-off is simplicity versus generality and efficiency. The simplicity comes from the fact that all FUSION service requests are in fact stateless and follow the application data plane (i.e., the control plane and data plane are aligned, compared to the generic architecture where the control plane is different from the data plane). However, the streaming component is responsible for forwarding (binary) input from the client to the rendering component, possibly making the component less generic. Also, the feedback channel in this case passes through the streaming component, introducing additional delay, especially in case the rendering and streaming components would be at different locations. This means that for achieving a particular utility (e.g., roundtrip latency), these service components may have to be placed closer to the client to compensate for this additional delay.

In our FUSION prototype implementation, we used this simplified application architecture.

3.1.4.1 Coordinator service component

Although the current prototype implementation does not include a separate coordinator component, which coordinates the communication and connectivity between the other service components, a fully functional EPG FUSION service will likely include such coordinator component, as shown in the generic application architecture shown above.

3.1.4.2 Evaluator service

As part of the EPG service components, we also provided a number of corresponding evaluator service probes. We extended the evaluator probe to support both stage 1 and stage 2 evaluation requests for all three main application service components (i.e., decoder, rendering and streaming), though in real life, each service component could in theory have its own stage 1 and/or stage 2 evaluator probe services. We implemented different types of evaluator services, ranging from very simple feature-based evaluations to running the actual full service itself. See Section 5.2 for more details.

3.1.4.3 Video decoder service component

A new service component (that in the previous implementation in D5.2 was still integrated inside the EPG rendering service component, is a general purpose video decoder service component. The internal architecture of this component is depicted in Figure 11. The key component is the session and

configuration factory component, which is a generic component developed in our Vampire Framework [FV09] that implements the session slot concept as well as the multi-configuration service (i.e., service aliasing) concept. This component will automatically spawn and clean up internal service graphs on-the-fly, resulting in a very dynamic and efficient system.

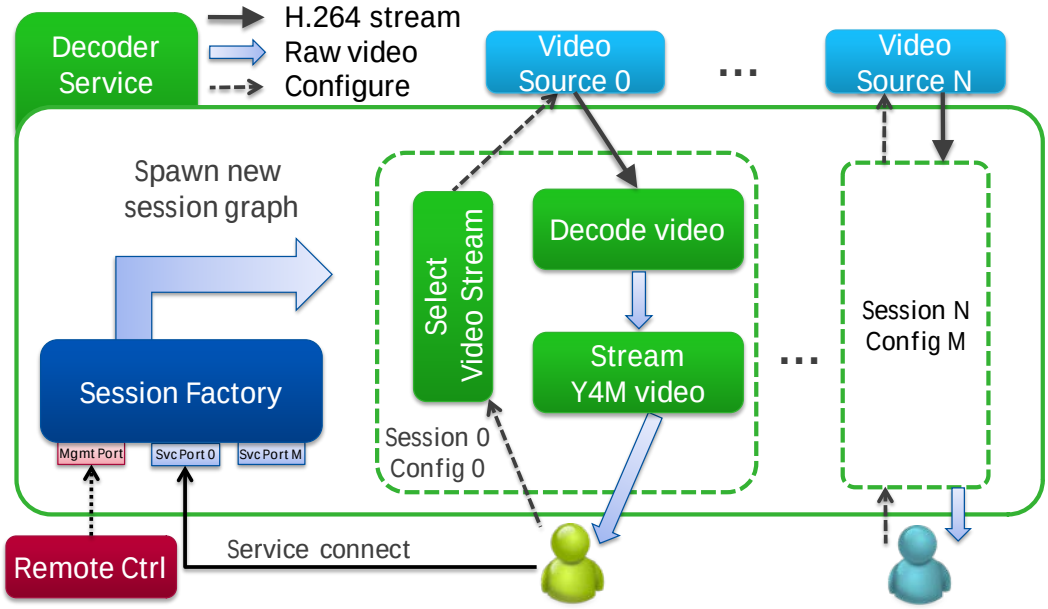


Figure 11. Software architecture of the multi-session multi-configuration enabled decoder service component.

A connecting client (which typically will be another FUSION service) provides its video stream to be decoded (e.g., local file, online video, live stream, etc.), which is subsequently decoded and encapsulated as a raw video stream and streamed to the connecting client.

Two key advantages of having this separate video decoder service component, are (i) easy external reuse of a generic decoder component, not requiring these capabilities to be included in all individual applications, (ii) more specialized and efficient use of specialized hardware (e.g. scalable hardware video decoders), software and real-time constraints, resulting in much better cost-efficiency. Moreover, in case of live streams, the same video only needs to be decoded once, after which it could be efficiently shared across multiple clients using some multi-cast mechanisms.

A key disadvantage is that to make this work, an efficient inter-service communication mechanism needs to be used in order to avoid the overhead of exchanging raw video frames. This also significantly limits the placement of such decoder services with respect to their client services, especially in classical static cloud infrastructures. We studied various inter-service communication mechanisms in Deliverable D3.3, and implemented some of them in our prototype implementation.

3.1.4.4 EPG rendering service component

The existing EPG rendering component has been modified accordingly to only implement the EPG rendering. Its final internal architecture is depicted in Figure 12.

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 sessions available, the factory will create a parameterized session graph, also setting up new or connecting to existing decoded video streams. All this complexity is abstracted via the Vampire framework. Note that the shared input videos are only imported once from a decode service, and are automatically multi-casted via an internal shared memory Vampire protocol.

As with all our other service components, 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.

We also included a monitoring component that monitors the application performance (e.g. frame rendering delay, etc.) and forwards this to an external service.

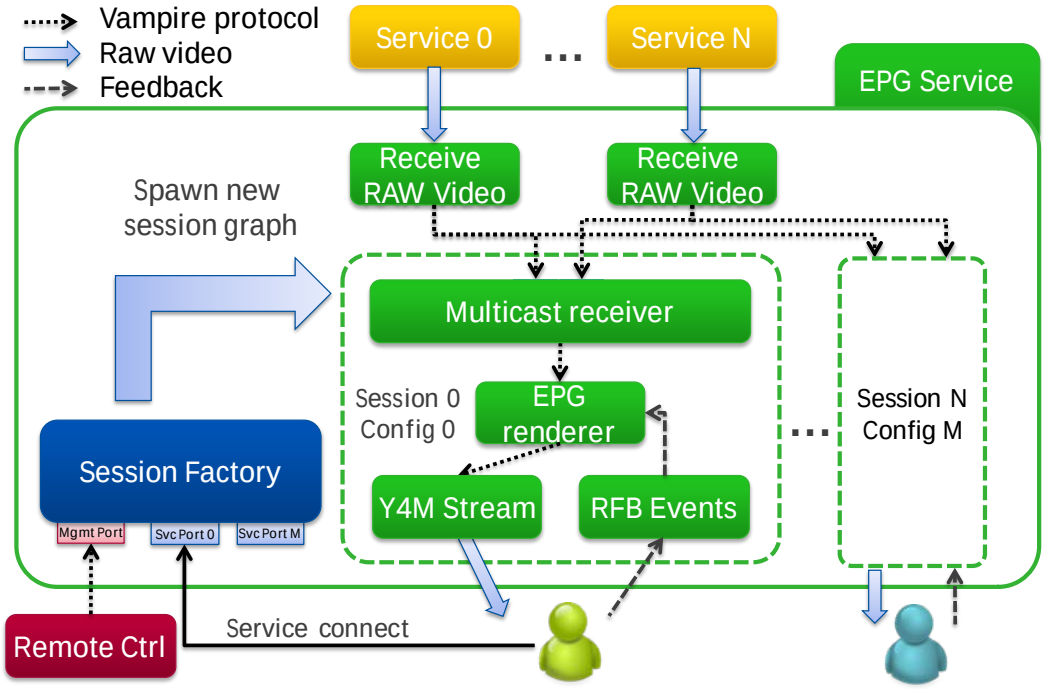


Figure 12. Software architecture of the multi-session multi-configuration enabled EPG rendering service component.

3.1.4.5 Streaming service component

The internal architecture of the streamer service component is depicted in Figure 13. The overall structure is quite similar to the previous service components, leveraging the same generic session factory Vampire component for handling all session and multi-configuration related aspects.

As with the other components, we also added the efficient inter-service communication mechanisms as well as the application monitoring component to measure application QoS metrics.

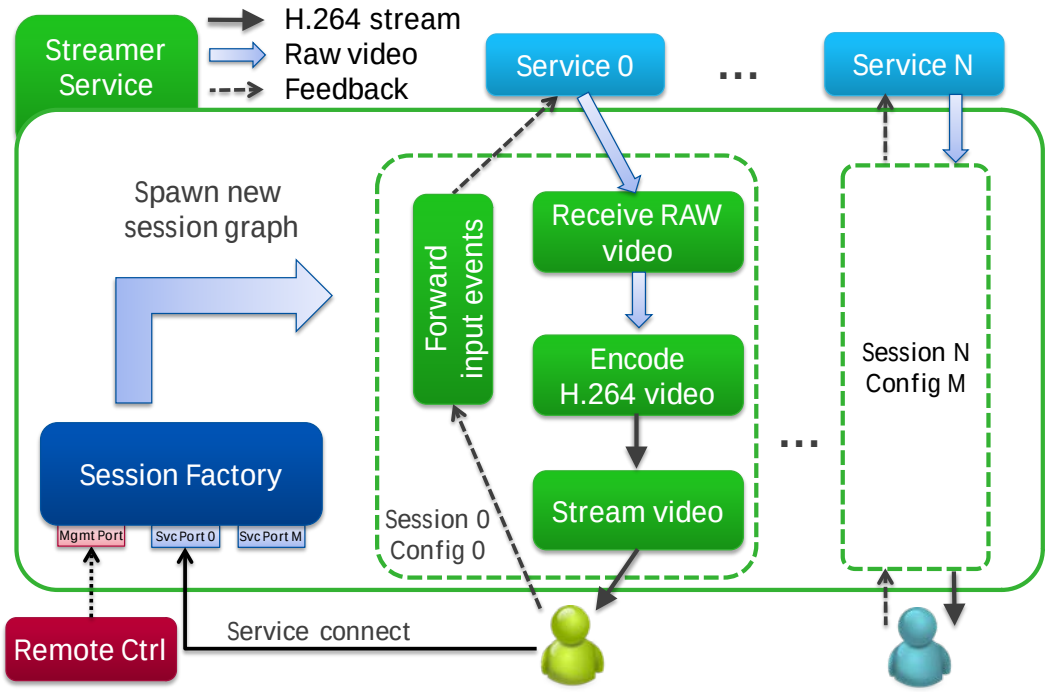


Figure 13. Software architecture of the multi-session multi-configuration enabled streamer service component.

3.1.5 Implementation

As mentioned already a few times, 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. The full Vampire pipeline description is presented in the Appendix in Section 8.1.

All FUSION-specific communication was provided in an external simple 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.

A new implemented feature is the session availability probe, which was also implemented as a separate probe running next to the application service component and which dynamically measures and adjusts the remaining session slots based on available resources as well as the internal application QoS metrics.

As described in previous deliverables, all required application binaries, libraries, scripts and artefacts were wrapped into Docker containers via Dockerfiles, enabling easy and fast deployment across any Docker-enabled runtime environment. Note that the evaluator probes were also packaged in a similar manner as a Docker container. For each service component, one or more manifest files were created to be able to deploy them into the demonstrator prototype.

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. Especially in an on-demand deployment scenario, this methodology is very important to minimize provisioning/deployment time as well as the aggregated size of all container images deployed in a node.

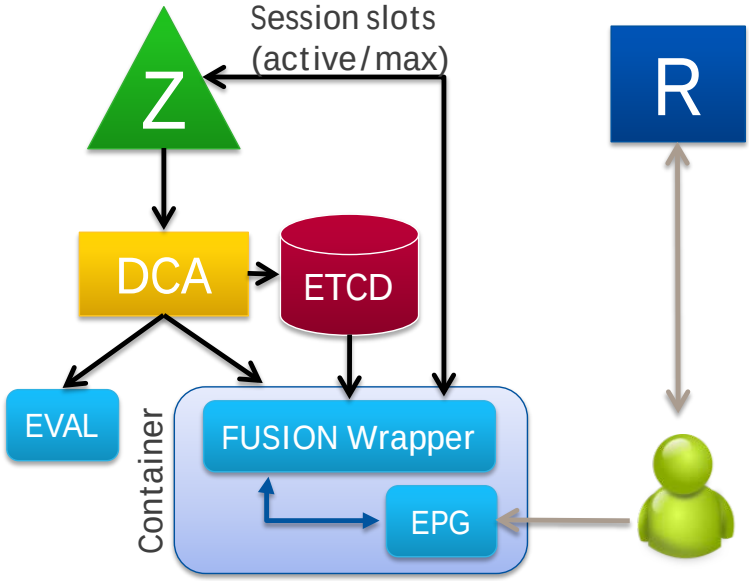


Figure 14. Integration of EPG service component in FUSION prototype.

3.2 Augmented reality

With the advent of next-generation augmented and virtual reality glasses such as Microsoft's HoloLens, Oculus Rift and even simple VR cardboards for a smart phone, new applications, creating virtualized or augmented interactive environments is becoming increasingly popular. These highly interactive applications typically require very high resolutions (more than 4k per eye-frame rates (e.g. 90+ FPS) as well as very low-latency 3D processing in order to be useful. Due to these extremely high requirements, several companies are even launching backpack VR computers to process everything in real-time (See http://www.theverge.com/2016/5/27/11790674/hp-virtual-reality-gaming-pc-backpack).

The FUSION architecture, promoting low-latency distributed heterogeneous cloud environments, could also be an excellent platform to deploy this type of demanding applications, so that people do not need to wear a backpack computer just to be able to walk around with these glasses.

As such, we also created a simple single-user interactive AR test application, where a camera feed is streamed to a FUSION decoder service, processed in real-time and then streamed back to the client. Although in this prototype, the entire augmented frame is streamed back to the client, in reality only the augmented overlay could be streamed back to the client.

A snapshot of this basic augmented reality service is depicted in Figure 15.

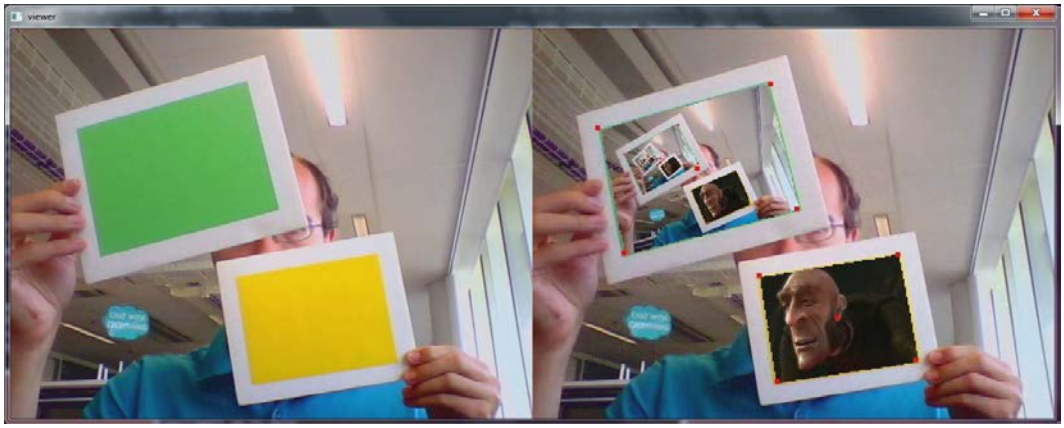


Figure 15. Screenshot of an augmented reality service prototype.

3.2.1 Features

A key difference with the previous use case application, is that a live camera feed is streamed from the client to a FUSION application services, processed in real-time, and streamed back to the client. Due to the very low-latency and high frame rate requirements, where the upstream from the client needs to be taken into account, this application type puts even more constraints on the selection of both the compute resources as well as the location of the services in the network with respect to the requesting clients.

As this test application was developed in the same Vampire framework as the EPG service components, the same FUSION features were implemented and validated. To implement this service as a stateless micro-service graph, we added a service-specific muxing and demuxing mechanism to efficiently stream multiple streams over the same connection.

3.2.2 Architecture

Similar to the EPG service, we designed this application as a micro-service application. A generic high-level view of the architecture is shown in Figure 16.

In this generic architecture, a service-specific coordinator component is responsible for finding and connecting proper decode, rendering and streaming component instances. This allows the client to directly send its live video stream to the optimal decoder instance as well as sending any other interaction events directly to the rendering component.

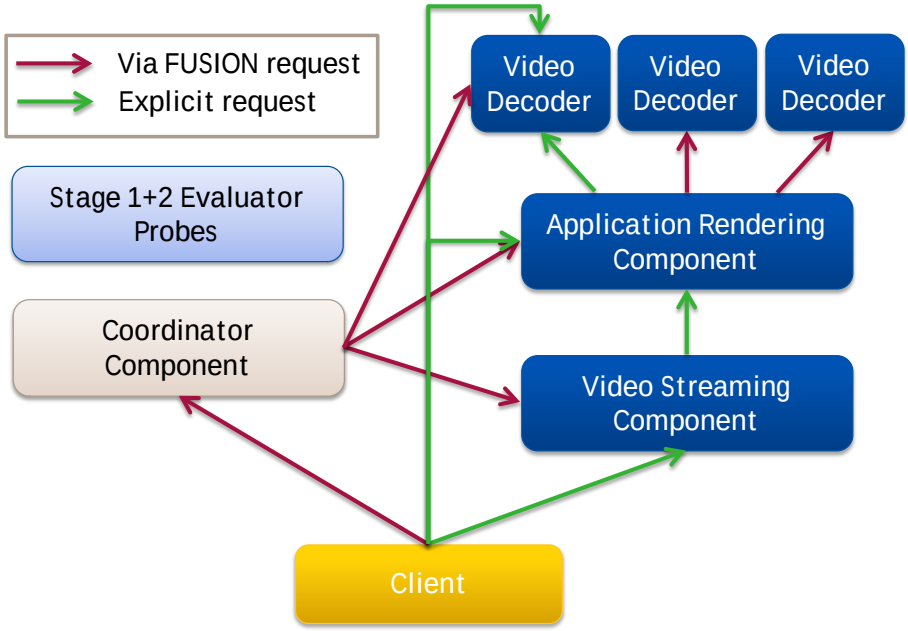


Figure 16. Generic High-Level Single-User Augmented Reality Application Architecture.

For this application prototype, we implemented this generic architecture with coordinator component, who will make third-party service requests to the FUSION resolver for finding optimal instances w.r.t. the requesting client. Each component will dynamically create the necessary server connections, which subsequently are returned to the corresponding components, so that each component knows which stateful session to connect to (cfr green edges in the diagram).

Note that for convenience, the coordinator component is also made available as a FUSION service, accessible via a FUSION resolver, and exposing session slots. However, as this component is not in the critical path nor data path, this component could also be deployed outside of FUSION and accessed as a regular cloud web service.

3.2.2.1 Augmented reality rendering service component

The internal architecture of this service component looks very similar to the internal EPG rendering architecture diagram shown in Figure 12. The main difference is that the internal software components themselves are configured to create a set of session-specific stateful connections.

3.2.3 Implementation

This prototype application service was developed as well using our Vampire framework. For the decode and streamer service components, we reused the same service components. For the rendering component, we reused an existing internal Vampire component and integrated it into a FUSION-enabled pipeline, also including additional functionality to be able to be compatible with the external coordinator software component.

The coordinator service component itself was created in Python, as this component only handles control plane functionality. All additional components were also wrapped into Docker containers, and corresponding manifest files were created.

3.3 Thin client game

As described in D5.2, this is a thin-client single or multi-player game prototype where the rendering of the 3D scenes is done on a separate server instance deployed on the network, receiving live input from the end user's device, and sending the live rendering output as video stream to the same end user's device. The end users only launch an input and viewer application (referred to as thin client), which is

responsible for acquiring the user's input and passing it to the server as well as decompressing the video stream and displaying it.

3.3.1 Implementation of the prototype

The prototype is based on the commercial Shark 3D software of Spinor, which is used for creating media applications in different markets for several years. We decided to use this software for the game prototype for two reasons:

- The results of this prototype including lessons learned and measurements results are closer to the real-world use-cases, since such kind of software (or the Shark 3D software itself) is likely to be used for implementing FUSION services.
- It simplifies exploiting FUSION results, since the FUSION features are integrated into a software used commercially anyway.

As part of FUSION we enhanced the Shark 3D software in three ways:

1) We redesigned the software architecture to be service oriented and supporting session slots. These changes were described in chapter 2.
2) Additionally, we implemented a new revision of low-latency video-in and video-out streaming capabilities to fit into a FUSION service infrastructure of different services like the EPG service discussed above. These features were implemented on basis of existing video-in and video-out features of the Shark 3D software, which were originally designed for different purposes like broadcasting graphics applications and videos within games. See section 3.3.3 for additional information.
3) We added various other features needed for FUSION like evaluator service support and packaging the simulation variant of the software into Docker containers in such way that they can be deployed by the FUSION zone manager prototype. See also sections 3.3.3, 4.2 and 4.3.

We used the Shark 3D software including these changes together with an existing 3D scene as prototype for a FUSION dashboard or FUSION game service component. A screenshot of the game prototype 3D scene rendered by the Shark 3D based service prototype can be seen in the following figure.

Figure 17. Screenshot of the game prototype

3.3.2 Authoring software for creating FUSION services

The changes described in section 3.3.1 extended the Shark 3D software from an authoring software for creating classical media applications like games or broadcasting applications also into an authoring software for creating FUSION services. The main benefit is the integration of the new service oriented networking features with all the existing features for creating interactive media applications, simplifying the production process of creating such applications with service oriented networking and scaling support. The integration also includes the documentation, see the following screenshot of the introduction section of the documentation of the Shark 3D software, linking to the new section about the new service oriented interactive media engine features.

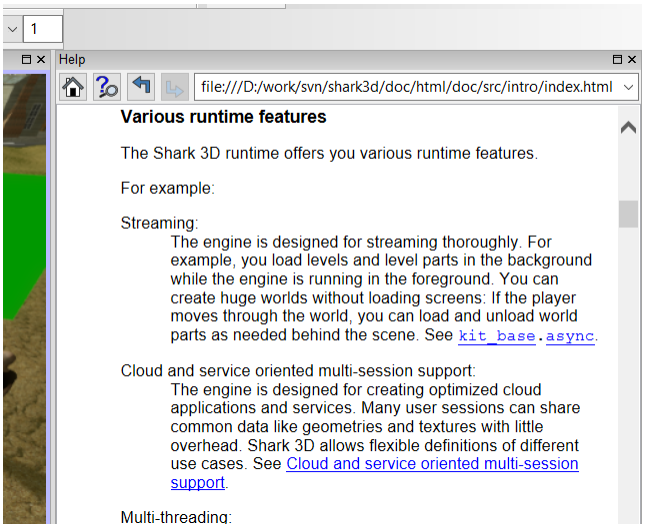


Figure 18. Help window of the authoring software for creating FUSION services

3.3.3 Features

The following describes how the prototype demonstrates the use of various FUSION functionalities:

1) Session slots with resource sharing

   The number of session slots relies on the resources used by the running sessions. Therefore, upon requesting the Shark 3D services (whether for the game or the dashboard, see section 3.4), the number of available of session slots is reported to the zone manager that handles the distribution of incoming client requests for new sessions over the available servers. The available number of session slots is calculated dynamically in the prototype. The average frame time per slot is calculated by dividing the current frame time by the number of slots occupied. Dividing a maximum allowed frame time (for example 16 ms at 60 FPS) by the current average frame time per slot gives the maximum number of slots possible. For example, if currently four slots together consume 8 ms per frame, then four more session slots are possible, totalling eight session slots. This information is updated and reported to the zone manager every 10 seconds. Any incoming request of a user consumes one session slot no matter whether it is a single or multi-player game.

2) Single and multi-user services

   The game service can be requested in either the single- or multi-player mode. In case of the single user game, the world simulation and rendering are offered as a combined service. As for the multi-player game, it can also be offered as two services where the first is a centralized server responsible for the world simulation and the second is a decentralized rendering service for each of the players of a game session. The different setups are further explained in section 3.8.

3) Low latency streaming services.

Since the game is an interactive application and the rendered output is streamed to the user, the network distance and servers' locations should be taken into account when resolving the requests for this service. The experiments done to measure the latency and quality of the service based on the locations where the server is deployed are presented in section 5.13.

4) Integration of existing professional software into FUSION

As explained in section 3.3.2, the architecture of the Shark 3D engine has been adapted to easily create FUSION-enabled services such as these prototypes of the game and the dashboard.

5) Hardware access (GPU)

To stream out the rendered output to the users, Nvidia NVENC [NVCOD], which was introduced with the Kepler-based GeForce 600 series for video encoding, was used as part of the implementation of the output video channel compression in Shark 3D. Therefore, this demonstrator requires access to a GPU whose graphics card supports NVENC.

6) Evaluator service

The evaluator service is a feature integrated into one variant of the Shark 3D software detecting if NVENC is available on the current platform by communicating directly with the NVENC component of the Shark 3D software and reporting the results to the FUSION zone manager.

3.3.4 Implementation of output video streaming

Shark 3D contains a few types of views depending on where the output should be rendered. This includes the "live view" node which renders the output to a viewer in the editor and the "stream view" node which renders the output to a TCP stream. As seen in Figure 19, a node called "Tcp Listen" is used to open a listening socket for incoming connections on the port specified as part of its parameters. Once it receives a client connection, the "Stream Manager" perch script is executed where a stream view is produced and is bound to a player of the 3D state previously created upon the initialization of the editor. If there was no state already available, then a new one will be created. Next a command will be sent to the "Stream Producer" to produce the NVENC node. The desired resolution is also set in the script (the default is 1280 x 720). The context implemented via the NVENC node is then assigned to the view to be rendered and streamed to the user.

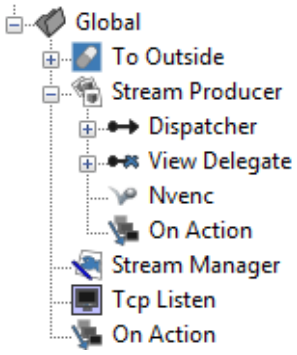


Figure 19. Selected nodes defining the NVENC streaming endpoints in the renderer services

3.3.5 Input handling

We connected the standard Shark 3D input configuration node system with the NVENC node for managing the input back channel. The input configuration nodes connect input events – in this case coming from the NVENC node – with other nodes defined in the project – in this case the game prototype application.

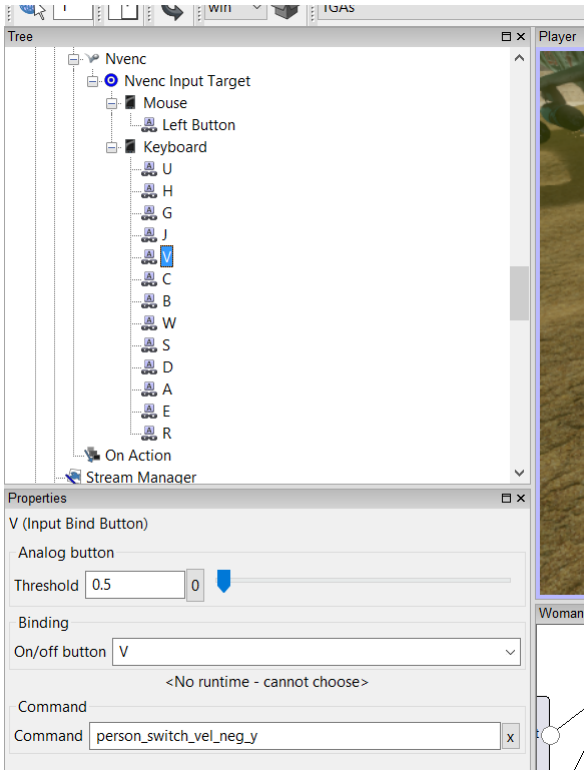


Figure 20. Selected nodes defining input handling in the renderer service

3.3.6 Evaluator service

A new Shark 3D node, implemented in Python, manages the evaluator services functionality within the game prototype. The node communicates with the NVENC node to fetch the availability of the NVENC features and reports it then to the FUSION zone manager.

3.4 3D media dashboard

The 3D media dashboard is an interactive 3D environment combining different media sources statically and dynamically, based on the users' requests. Therefore, it allows the user to explore media and choose one of the services offered such as video streaming of real time data from various sources.

3.4.1 Features

One main difference between the game and the dashboard is that the video streams are rendered on textures within the 3D world, which could be the same or different for each user even within the same session in case of the multi-user dashboard. To implement this, each user must have a logically unique 3D state and a view, where the 3D states of all the users joining the same session are synchronized via Shark 3D networking protocol. See chapter 2 and section 3.8 for additional details.

Other than that, the underlying structure of the dashboard is considered to be very similar to that of the game and so it uses the same functionalities provided by the FUSION platform as those used by the game prototype. Since it also requires to stream the rendered output to the users, a GPU supporting NVENC is needed. Therefore, we can re-use the existing service implementation, including the evaluator service, as that used for the game as they basically have the same service requirements.

Part of the service prototype is also the feature that the service itself can request another service for integrating a video stream into the dashboard. This is a sample for a FUSION service dynamically requesting another FUSION service. For demonstrating this feature we connect to the EPG service discussed above.

3.4.2 Implementation

Technically the dashboard prototype is the same Shark 3D based software as the game prototype described in section 3.3, but controlled in a different way by the lobby software described in 3.6.

3.5 Chat service

A chat service is a sample for a service which does not have tight real-time constraints and transmits less data than video streams. We included such a service as another sample for a heterogeneous set of services. It also demonstrates how the commercial Shark 3D software enhanced by FUSION features can be used for implementing different kinds of FUSION services in the area of media applications quite easily, see section 3.3.2.

3.5.1 Features

Multiple clients can connect to a chat service, exchanging chat messages. A user can type in a chat message. After pressing the RETURN key that message is transmitted to the other clients and displayed together with the name of the user who wrote that message.

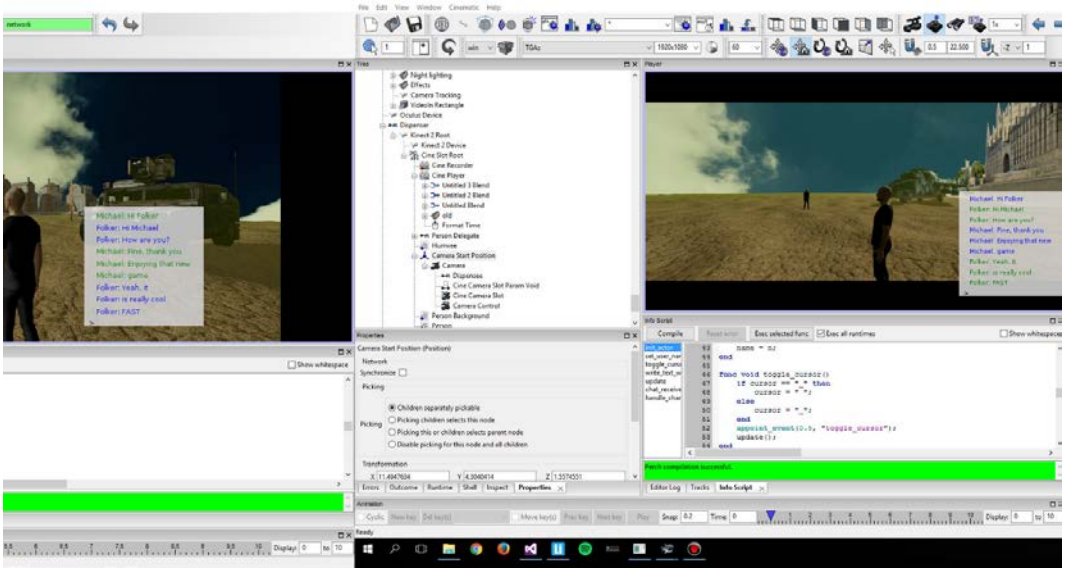


Figure 21. Screenshot of two connected instances of the chat service

3.5.2 Implementation

Technically the chat service is also a Shark 3D based application, consisting of a server and a client part. We based it on the gaming prototype, including the test 3d scene we are using for it, and added the chatting functionality to these applications. The applications then can be packaged as FUSION services in the same way as the game and dashboard prototypes.

3.6 Lobby software

3.6.1 Features

The lobby software itself is not a FUSION service, but a software component which requests Shark 3D based FUSION services described in sections 3.3 and 3.4 on behalf of the user.

In a real-world setup, the lobby software could be for example a Web application where users can meet online and launch a dashboard or game application. In the FUSION application prototype the lobby software is a stand-alone application written in Python. Therefore, a simple lobby interface, pictured in the following, has been developed to be able to start the different scenarios explained in sections 3.3, 3.4 and 3.8.

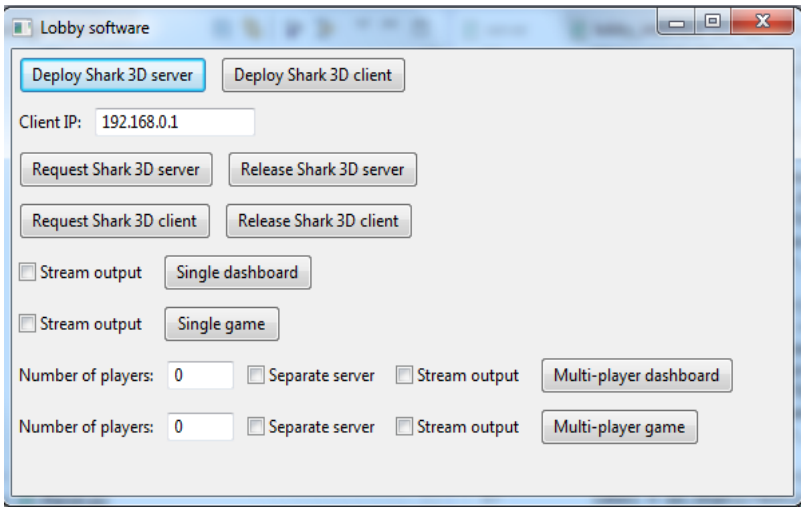


Figure 22. Screenshot of the lobby software

The interface can be used either to deploy raw Shark 3D based services (a server service or a client service, see section 3.8) or to request a full dashboard or game scenario in single or multi-user mode. For the multi-player dashboard or game, the number of players has to be defined first. Also the separate server option allows having a dedicated server which all clients are connected to, to have a centralized world simulation as described in section 3.8.2.

For each scenario session, a separate window is opened to present the logging information about what has been created within the engine for this specific session as can be seen in the following screenshot. The session can be closed by the terminate button at the bottom which destroys all what have been produced for this session.

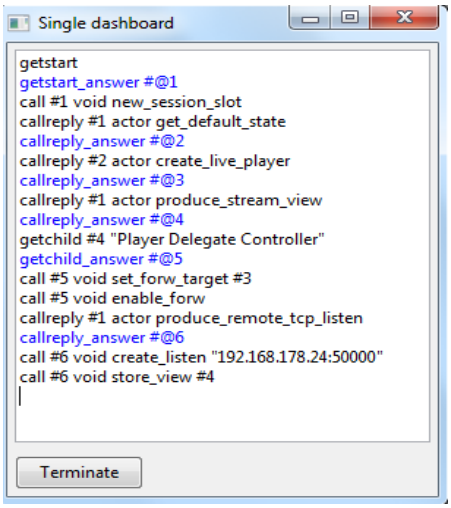


Figure 23. Window for a scenario session.

3.6.2 Implementation

When requesting a scenario in the lobby software, a number of service requests are sent to the FUSION resolver possibly demanding a simulation service (also called Shark 3D server) and rendering services (also called Shark 3D client, not to mix up with the thin client), based on the scenario selected. The communication between the lobby interface and the resolver is based on Rest API as explained in D5.2.

The service endpoints returned by FUSION are used to establish connections between the lobby service and the services deployed by FUSION using the Simple Actor Protocol (SAP) of the Shark 3D software. The SAP is a network communication protocol allowing to execute and receive RPCs to access the Shark 3D based services.

Depending on the scenario (dashboard versus game) and depending on the number of users, the lobby software controls the Shark 3D based FUSION services in the following way, see also section 3.8:

- It controls how many 3d simulation states are instantiated in the Shark 3D software instance.
- It also controls how many views are instantiated containing the Nvidia NVENC [NVCOD] functionality.
- It connects the 3d simulation states with the corresponding views.
- It creates suitable TCP networking endpoints in the Shark 3D simulation and rendering instances, logically connected with the 3d simulation states.
- It directs the Shark 3D software instances to connect the different networking endpoints depending on the kind of application and scenario and based on the endpoints returned by the FUSION resolver.
- It creates TCP networking endpoints for receiving input from the thin client and streaming the video output to it.
- The addresses and ports of these endpoints are passed by the lobby service to the thin client.

The following log output (including log messages and SAP commands) shows the part when the lobby software requests three FUSION services (two rendering services called “shark3dclient” and one simulation service called “shark3dserver4”), FUSION returning the service instance endpoints (192.168.178.202 for the rendering services and 192.178.201 for the simulation service).

```
192.168.178.24
<Response [200]>
shark3dclient.spinor.de
192.168.178.202
192.168.178.24
<Response [200]>
shark3dclient.spinor.de
192.168.178.202
192.168.178.24
<Response [200]>
shark3dserver4.spinor.de
192.168.178.201
SAP send: getstart
SAP recv: getstart_answer #@1
SAP send: call #1 void new_session_slot
SAP send: callreply #1 actor get_default_state
SAP recv: callreply_answer #@
SAP send: callreply #1 actor produce_state
```

Figure 24. Log output of the lobby software requesting services

The following log output shows parts of the SAP communication of the lobby software creating player instances, views and network endpoints in the Shark 3D renderer service in order to configure it as dashboard renderer service:

```
Listening at
SAP send: callreply #2 actor create_live_player
SAP recv: callreply_answer #@4
SAP send: callreply #2 actor create_live_player
SAP recv: callreply_answer #@4
SAP send: getchild #2 "Network Connect"
SAP recv: getchild_answer #@5
SAP send: callreply #5 int connect "192.168.178.201:60003"
SAP recv: callreply_answer 1
Connecting to 192.168.178.201:60003
SAP send: callreply #1 actor produce_stream_view
SAP recv: callreply_answer #@6
SAP send: call #6 void assign_player #4
SAP send: callreply #1 actor produce_remote_tcp_listen
SAP recv: callreply_answer #@7
SAP send: call #7 void create_listen "192.168.178.32:50006"
SAP send: call #7 void store_view #6
SAP send: getstart
SAP recv: getstart answer #@1
```

Figure 25. Log output of the lobby software configuring the service instances

After the creation of the required components of the dashboard, the thin client can be automatically started and it connects to the view within the Shark 3D engine for the rendered output to be streamed. In case of a multi-player service, each TCP stream created is bound to its own view, which in turn is bound to one of the players in the scene, thus each user gets his own perspective view within the session.

3.7 Prototype web UI

3.7.1 Features

For managing, coordinating and visualizing all prototype components, we developed an interactive web UI. A screenshot is depicted in the figure below:

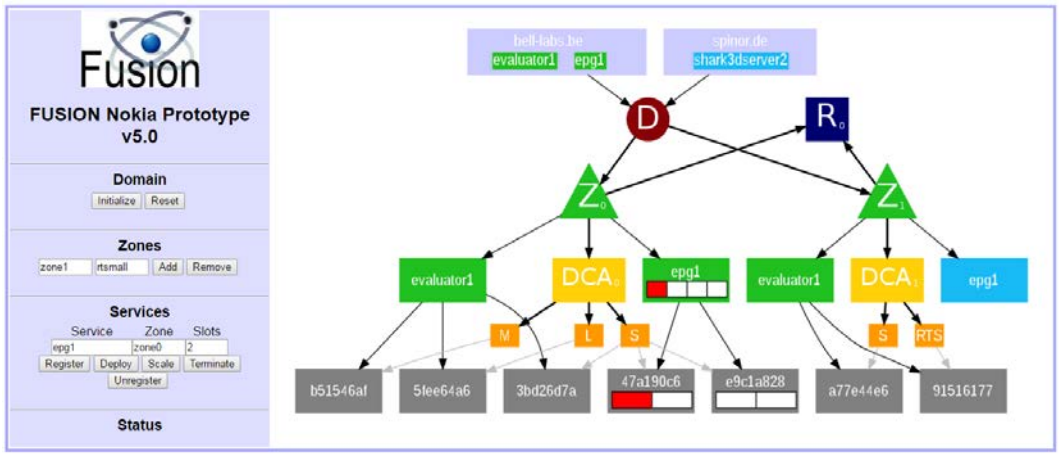


Figure 26: Interactive Web UI for controlling the integrated FUSION prototype

The left pane provides a set of basic controls to interact with the deployed testbed, whereas the right pane depicts a live generated overview of the entire infrastructure. At the domain level, one can completely reset the entire setup, as well as initialize a basic setup, including a domain orchestrator (i.e, the dark red circle), basic resolver (i.e., the blue square), as well as a single zone (i.e., the green triangle) on top of a particular DCA layer (i.e., the yellow rectangle), consisting of one or more environment types (i.e., the orange rectangles).

At the zone level, one can add or remove zones with a corresponding DCA type. At the service level, there are basic controls for registering or unregistering a service (in a domain and/or specific zone), manually deploy or terminate a service (in a domain and/or specific zone), as well as manually scale up or down the number of session slots in a particular zone.

In the right pane, all registered services are depicted at the top, and are grouper per service provider. In this example setup, there are two service providers, each having one or two registered service. Registered services that also deployed in one or more execution zones are depicted in green, whereas services that are only registered in a zone or domain are depicted in blue. The grey rectangles at the bottom represent actual deployed (container or VM) service instances.

The FUSION prototype can handle two types of services, namely evaluator services and actual public FUSION services. The latter case typically will announce session slots availability to the zone manager. The number of available slots are depicted by the white rectangles; the number of used session slots are in red. In our example, there are two instances of the EPG service deployed in zone 0, of which one session slot is currently in use.

3.7.2 Implementation

This interactive Web UI has developed on top of the Python Flask framework, and encapsulated in a Docker container. The FUSION REST APIs were triggered on the actual testbed from within this framework. The actual running state of the entire demonstrator setup (i.e., the domain, zones, services, etc.) was rendered using the graphviz software package, by a software module that queried the entire FUSION state by triggering key FUSION REST APIs implemented by the prototype.

3.8 Typical setups

The game and dashboard services, which are both based on the Shark 3D software, can be set up in different ways according to both their hardware requirements and the users' locations. Examples of the former are represented in the world simulation that requires high CPU power and also in the rendering services which require access to the GPU. Such requirements put limitations on where the service instances should be deployed and whether or not they can run on the same machine. The users' locations on the other hand, determine where the rendering instances need to be deployed as these should be as close to the users as possible. The following reflects the possible scenarios whether or not the services are split for each of the use cases of the dashboard and the game.

The implementation of these scenarios is based on the architecture described in chapter 2. The different scenarios are controlled by the lobby software as can be seen also in the screenshot in section 3.6.1.

3.8.1 World simulation and rendering running on the same machine

3.8.1.1 Single-user dashboard / game

In this case, an instance of Shark 3D engine is instantiated, offering a number of session slots where each slot is equivalent to a new dashboard or game session comprised of a 3D state and a view. The rendered output is then streamed to the thin client running on the end user's device. The following diagram shows the single-slot situation for this scenario. In case of multiple session slots, the respective component instances are duplicated.

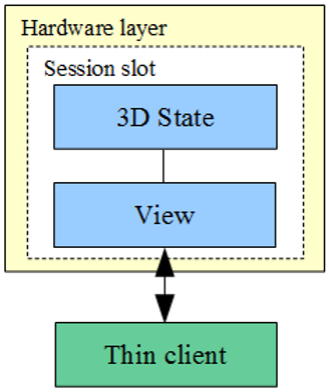


Figure 27. Single-user dashboard / game

3.8.1.2 Multi-user dashboard

A 3D state and a view are required for each user, hence is the consumption of one session slot per player to allow users to have different output streams as explained earlier. For each group of users sharing the same session, the 3D states must be synchronized via Shark 3D networking protocol. The following diagram shows the single-slot situation for that scenario. In case of multiple session slots, the respective component instances are duplicated.

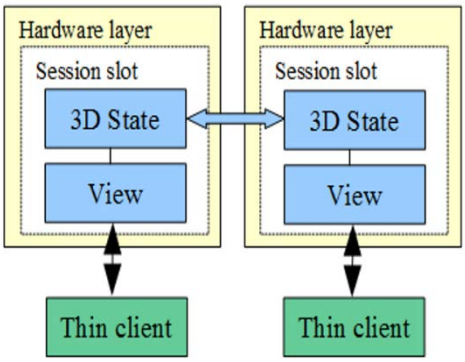


Figure 28. Multi-user dashboard

3.8.1.3 Multi-player game

As for this use case, only one 3D state is required per game session to which all players of the same session are connected. A new view is created for each player. The following diagram shows the single-slot situation for that scenario. While there is no objective criteria to which session slot the shared 3d state is belonging to, the diagram contains it in the left session slot.

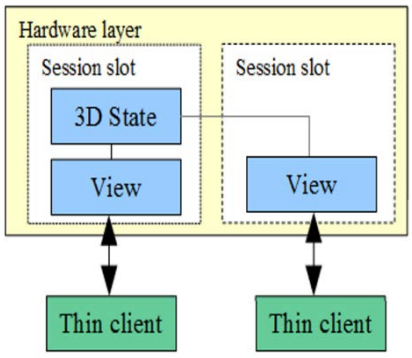


Figure 29. Multi-player game

3.8.2 World simulation and rendering services running on different machines

3.8.2.1 Single-user dashboard / game

Considering a single-user dashboard or game where a new slot is accorded to each user and no synchronization between states is required, assigning a separate instance for a server and another for a client consumes two session slots instead of one. Hence, this scenario requires the same setup as the scenario mentioned in section 3.8.1.1.

3.8.2.2 Multi-user dashboard / game with centralized world simulation

Similar to the multi-user dashboard requirements, once the rendering services are running on different machines, a new 3D state will be required for each user in the multi-player game. Therefore, it can be safely admitted that the same setup is applicable to both, the dashboard and the game. In this scenario, a session slot of a server instance containing the decisive 3D state is requested per session. Additionally, a session slot of a client instance containing a state replication and a view are also created (consuming one session slot) for each of the players within the session. All of the clients' states should be synchronized with that of the server using Shark 3D networking protocol. The following diagram shows the single-slot situation for this scenario. In case of multiple session slots, the respective component instances are duplicated.

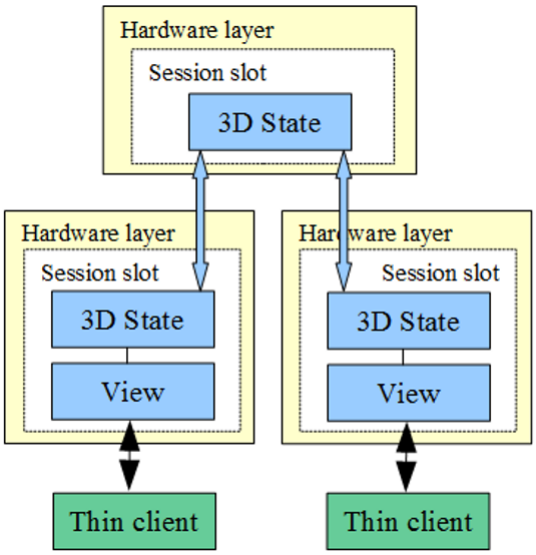


Figure 30. Multi-user dashboard / game with centralized world simulation

3.8.2.3 Multi-user dashboard / game without centralized world simulation

The last scenario is a combination of the previous two categories where the rendering services are instantiated on different machines depending on the users' distribution, while the world simulation is not separated from the rendering services. In this case here, one of the 3D states created for one of the users will be acting as a client as well as a server by opening a listening port for the rest of the clients who would like to join the same session to connect to. The benefit of this approach is that it uses less number of session slots against the previous scenario. It also reduces the number of states that need to be synchronized with each other. The following diagram shows the single-slot situation for this scenario. In case of multiple session slots, the respective component instances are duplicated.

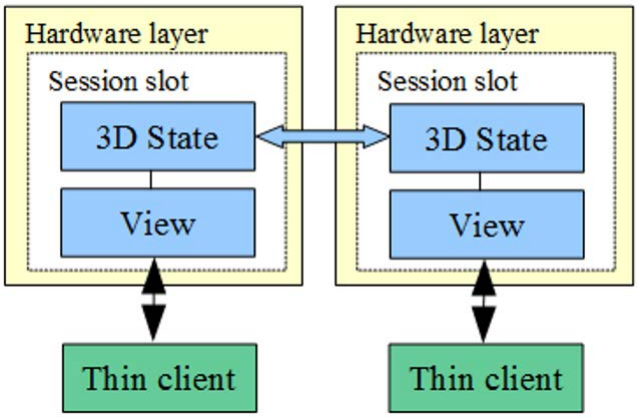


Figure 31. Multi-user dashboard / game without centralized world simulation

4. END TO END INTEGRATION

4.1 EPG integration

As mentioned in section 3.1, all EPG service components as well as evaluator services have been containerized as Docker containers and pushed to a private FUSION Docker registry that was set up on the Virtual Wall testbed. This enabled the efficient and easy provisioning and deployment on a number of FUSION testbed locations.

4.2 Simulation services integration

The Shark 3D based simulation services, which are used both for the game prototype (see section 3.3) and the dashboard prototype (see section 3.4), are packaged into Linux Docker containers, which can be deployed and managed by FUSION.

4.3 Rendering and chat services integration

The Shark 3D based rendering service, which is also used both for the game prototype (see section 3.3) and the dashboard prototype (see section 3.4), including the chat service (see section 3.5) uses a different setup due to constraints of the existing Shark 3D software and the testbeds. This situation is a good sample for a situation which may also occur when porting other existing software to a service and cloud oriented infrastructure like FUSION. This is also the reason why we explicitly chose to build the rendering service on the basis of the Shark 3D software instead of avoiding such challenges by a different prototype which is farther from existing real-world software.

More in detail, the rendering functionality of the Shark 3D software requires running under Windows, which conflicts with the Linux based Docker approach required by FUSION. It also requires access to a NVENC capable NVidia hardware, which is not available in the Virtual Wall testbeds, but only on a PC in the Spinor network. Therefore, we split the rendering service into two separate components working together transparently. The first component running under Linux is packaged into Docker containers deployed and managed by FUSION in the same way as the simulation service. With the help of a special server component, the first component then instantiates and controls a second component on a different computer. This second component contains the rendering functionality, runs under Windows and requires the NVENC access. This splitting of the service implementation is managed internally in the service implementation and is transparent to FUSION, which sees a service instance as one single unit. Note that each such service instance offers multiple session slots, which are managed internally by the rendering service implementation.

This specific setup for the Shark 3D based rendering service does not affect the functional evaluation of FUSION. It also does not affect the performance measurements as long as we ensure that FUSION

deploys the Linux part of the rendering service in the Spinor network where the NVENC-powered computer is located. For a real-world deployment it would be preferable to create a single component version of the Shark 3D rendering service running under Linux and packaged completely into a Docker container. This would require a) porting the Shark 3D rendering service to Linux, and b) using datacentres having NVENC support.

Such approaches of a more complex internal structure of a service may be also necessary for other existing software ported into such a service oriented infrastructure. Possible reasons may include:

- The software may have specific operating system or hardware requirements as with the Shark 3D rendering components.
- The existing software is already split into multiple components running on separate hardware, but for some reason it is not feasible to manage these components as individual services by FUSION because of a special relationship of these components hardwired into that software, but not known by FUSION. In case of the Shark 3D software this wasn't the case: The two parts of the software (the simulation component and the rendering component) could be nicely mapped into two different services, managed by FUSION.

4.4 Lobby software integration

The lobby software is running on a Windows computer and connecting to the FUSION orchestrator via the FUSION deployment and resolution network protocols.

4.5 FUSION orchestration platform Integration

Prototype implementations of the FUSION orchestration (i.e. domain orchestrator, zone manager, dca, etc.) have also been containerized and pushed to our private FUSION Docker registry.

4.6 Service container integration

To facilitate the easy deployment and integration of all prototype software components from all partners, we wrapped every component into different Docker containers, while reusing as many intermediate container layers as possible, as this can significantly impact the provisioning time and cost when deploying service containers. The full graph of all Docker containers for all application components as well as FUSION orchestration components, is depicted below, generated using the dockviz [DOCK16] visualization container for automatically generating graphs.

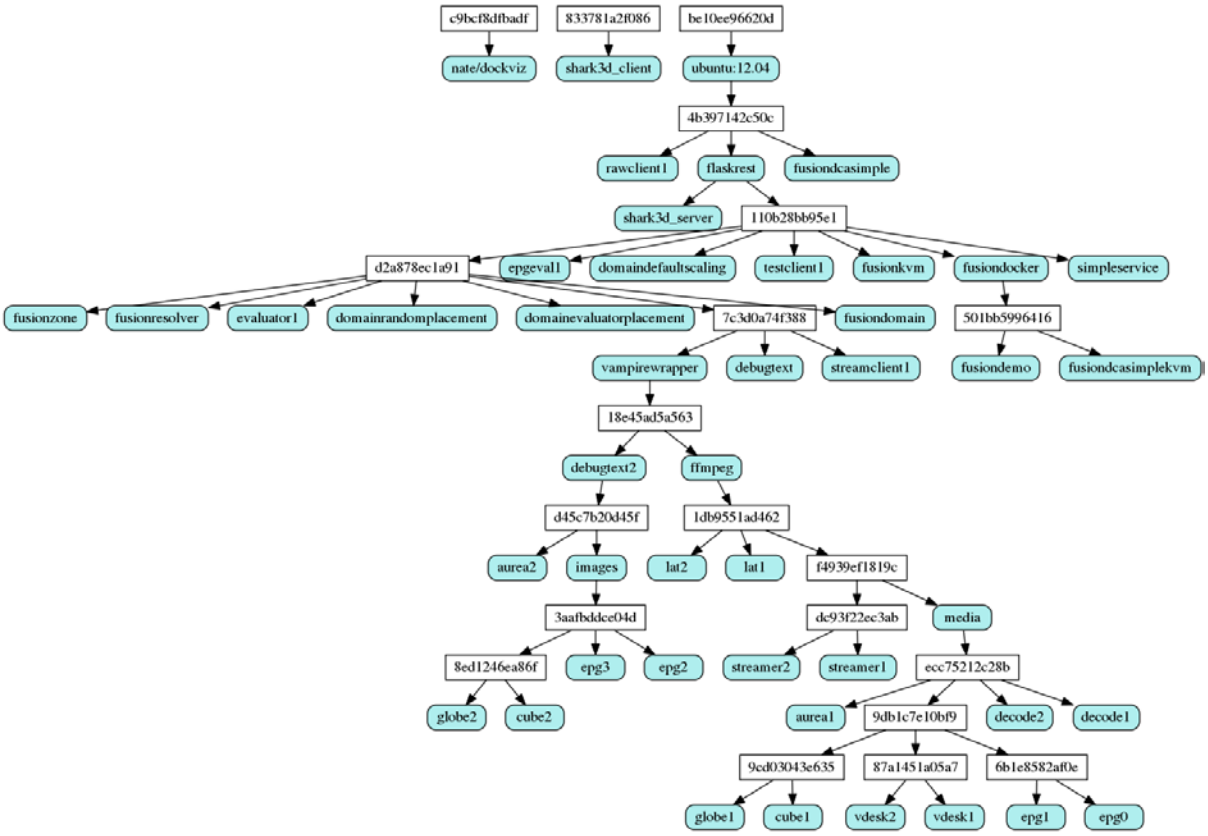


Figure 32. Diagram of all Docker images used in the prototype testbed

In this graph, intermediate layers have been removed to reduce the graph, so in real life, there are many more additional layers. The effective number of layers per container, is depicted in the graph below. On average, the containers have about 25 layers; many of the EPG services and variants have the most layers to enable optimal reuse per layer.

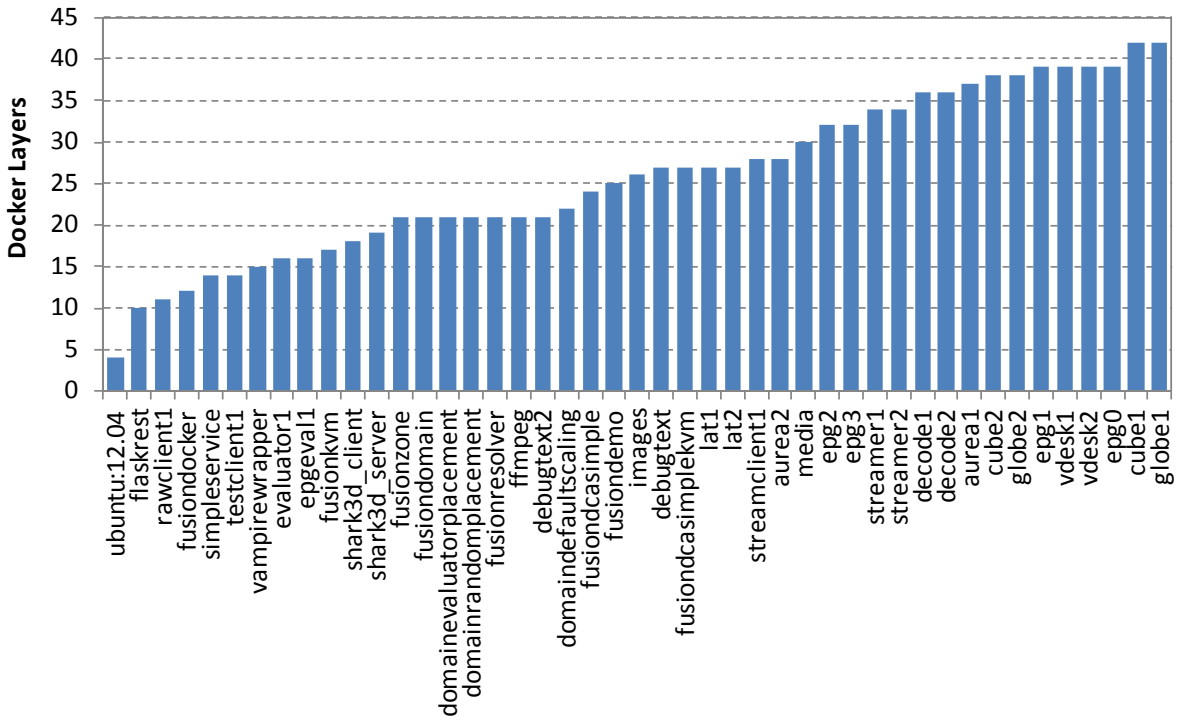


Figure 33. Number of Docker image layers per FUSION service container

The globe1 service container, representing the 3D sphere EPG as described in Section has the most layers (i.e., 42 layers). In total, there are 1161 layers, of which there are 327 unique layers, resulting in an average reuse of about 3.5. Note that 154 of these layers are not metadata layers. The reuse histogram is depicted below.

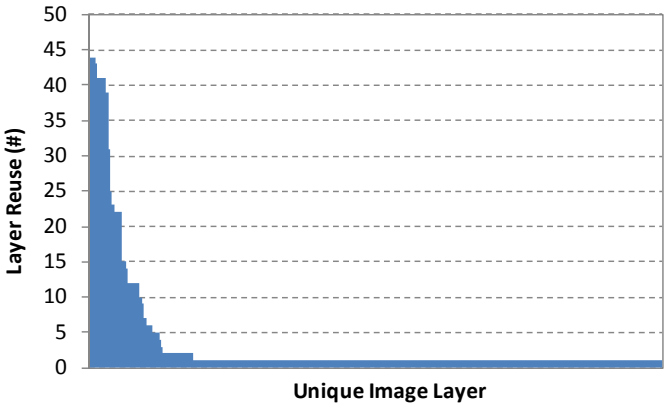


Figure 34. Reuse histogram of all unique Docker image layers

An overview of the total aggregated container size for each container is depicted below. Note that the largest service containers at the right-hand size are children from the *media* container, where a number of test video files (about 400 MB in total) are statically encapsulated in a separate container layer for easy testing.

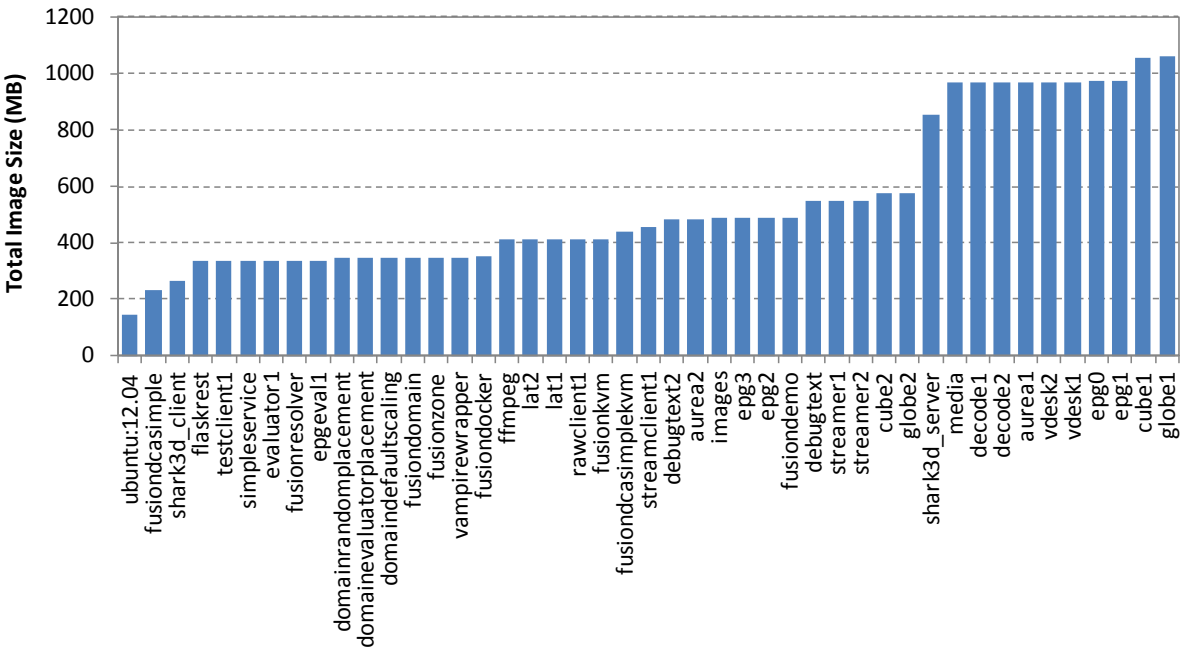


Figure 35. Number of Docker image layers per FUSION service container

The respective image layer sizes for the largest container, namely *globe1*, which is a fat non-composite service, is depicted below. In this figure, the final container image layer has number 1, whereas the base layer as number 42. Most layers are relatively small; the *media* test layer is layer 13.

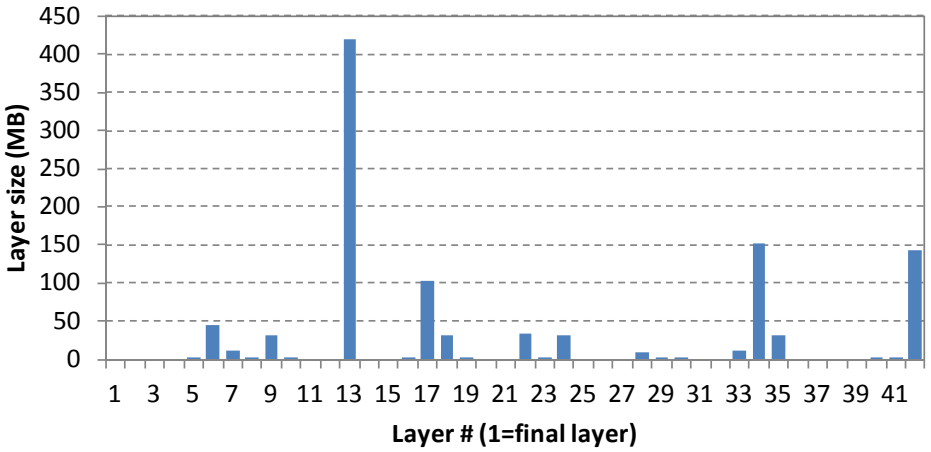


Figure 36. Docker image sizes for all image layers for the *globe1* service container

The respective sizes of all unique image layers is shown below. About half of the unique image layers are metadata layers of size 0; about 50 (i.e., 15% of all unique layers, or 33% of all unique non-zero layers) of them are larger than 1 MB.

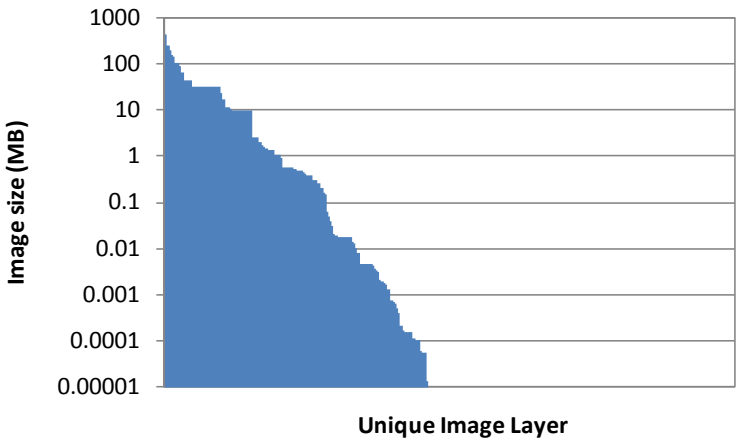


Figure 37. Docker image sizes for all unique image layers

4.7 Testbed integration on the Virtual Wall and Spinor nodes

On each testbed node, we installed Docker (version 1.9.1) on an Ubuntu 14.04 64-bit server OS; we configured the basic bridging network in such way that all containers are accessible from outside the hosts.

On the main node on the Virtual Wall, we subsequently deployed a private Docker registry, where all partners could push their containers to. This private Docker registry was accessible from all other testbed nodes and facilitated easy distribution and deployment across a wide range of nodes.

To facilitate the deployment of the testbed and all components, we created a main configuration script, which enables easy deployment of the web UI as well as individual prototype components, such as adding or removing a (remote) execution zone to some FUSION testbed. These scripts have been developed to enable multiple parallel demonstrator deployments. These scripts enabled easy the deployment the web UI, additional zones as well as starting various thin clients.

These master scripts themselves also have been Dockerized in a container, and have been made available by a single basic bootstrapping script that starts a temporary container on-the-fly to execute the command. As such, only this generic script had to be copied on the various testbed nodes; everything else was automatically fetched from the central private Docker registry as containers. When a new version of the main scripts was made available as a container, this simple bootstrapping script would automatically fetch the latest version and execute the command. This bootstrapping script is shown below:

```
#!/bin/bash
FUSION_DOCKER_REGISTRY=${FUSION_DOCKER_REGISTRY:-
10.2.33.228:5000}
sudo docker run -it --net host --rm \
  --env ETCDCTL_PEERS=$ETCDCTL_PEERS --env
FUSION_DOCKER_REGISTRY=$FUSION_DOCKER_REGISTRY \
  -v $PWD/manifest2:/www/manifest2 -v
/var/run/docker.sock:/var/run/docker.sock \
  -v /tmp/.X11-unix:/tmp/.X11-unix -e
DISPLAY=$DISPLAY -v
$HOME/.Xauthority:/.Xauthority \
  $FUSION_DOCKER_REGISTRY/fusiondemo ./fusion.sh
$*
```

4.8 Testbed integration on Orange datacentre

Orange lab facilities were described in deliverable D5.2. They were used for the deployment of FUSION resolver and ALTO server integrated in the tested prototype.

Regarding the reference architecture of FUSION resolver described in D4.3, Service request handler and Forwarding/resolution table were hosted by Orange datacenter as separate applications using common virtual machine. The following components were used to build their runtime environment:

- OS: Debian GNU Linux 8.0 amd64 (64-bit architecture)
- Application server: Wildfly 8.2.0 Final
- Database: PostgreSQL 9.4
- Java: openjdk-8-jre version 8u45

ALTO server function (Network/cost map client in the reference architecture of resolver in D4.3) was implemented on a dedicated virtual machine using the following technologies for its main functional blocks (ALTO Web Service, ALTO Administration Portal, database):

- ALTO Web service (implements interface N4 as of the reference architecture of the resolver in D4.3)
    - Java application using JAX-WS Web Service library with EclipseLink MOXy
    - Run environment – Wildfly (Jboss) 8.2.0 Final application server
- ALTO Administration portal (implements configuration functions to manually set ALTO maps for testing purposes; not shown in the reference architecture of the resolver in D4.3)
    - Vaadin framework for Java
    - Run environment – Wildfly (Jboss) 8.2.0 Final application server
- Database:
    - PostgreSQL 9.4 with Hibernate provider for Java

For the integration of all components used in the prototype testing both Service request handler and ALTO server were accessed by respective functions (end clients and Service-instance-client mapper) at public IP addresses through RESTfull interfaces. The interface to Service request handle is described in D4.3 and Internal Report I2.1 Final Specification of FUSION Interfaces. The interface to the ALTO server is described in D4.3.

5. SYSTEM EVALUATION

5.1 Service registration and deployment

5.1.1 Testing plan

We set up two zones:

- The Virtual Wall testbed of iMinds as described in section 4.1 of D5.2.
- The private network of Spinor. In this network we used a single Linux VM for FUSION.

We register the following services:

- EPG (Electronic program guide) video streaming service, see section 3.1.
- Streamer service for encoding the EPG video stream, see section 3.1.
- Shark 3D simulation service (also called Shark 3D server), based on the Shark 3D software of Spinor extended by FUSION features, see sections 3.3, 3.4 and 3.8.
- Shark 3D rendering service (also called Shark 3D client, not to mix up with the thin client), also based on the Shark 3D software of Spinor extended by FUSION features, see sections 3.3, 3.4 and 3.8.

- A dummy evaluator service
- For requesting services, we used the lobby described in section 3.6. That software can run on any Windows computer. For convenience, we run it also on the computer having NVENC support.
- Last but not least, we used different computers as end-user devices running thin clients. We mainly used one Windows PC for this, and sometimes also some other computers.

5.1.2 Involved components

- Orchestrator
- EPG service
- Streamer service
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.1.3 Results

5.1.3.1 Storage space usage

Since the rendering and simulation services are based on the commercial Shark 3D software and use an existing – although small – 3D test scene (see section 3.3 and 3.4), they provide exemplary storage space usage numbers for real-world services. Note that in our specific setup the binaries and assets of the rendering service are deployed under Windows (see section 4.3). For memory usage and load see section 5.6.3.

Service		**Simulation**	**Rendering**
Storage usage	Service binaries	27 MB	95 MB
	Service assets	365 MB	365 MB
	Docker image & libraries without service binaries and libraries	399 MB	252 MB
	Total	791 MB	712 MB

These numbers show that the overhead of packaging an application into a service, which is basically the underlying Ubuntu Docker image and libraries, is smaller than the application code and assets. While the test scene we are using is an existing real-world sample, it is still a small scene compared to larger real-world applications, e.g. games, videos or other media assets. Therefore, the overhead via packaging an application into a service will be smaller.

Our conclusion is that the storage space overhead of packaging applications into FUSION services is reasonable and gets quite small for larger real-world applications.

5.1.3.2 Deployments

The following are screenshots of the FUSION prototype software displaying the registered and deployed services live. The first screenshot is from the web UI (see section 3.7) of the zone manager of the Virtual Wall, while the second is from the Spinor zone.

Note that the boxes which don't contain session slots are only registered but not deployed (e.g. evaluator1), while boxes having session slots are also deployed (e.g. shark3dserver4).

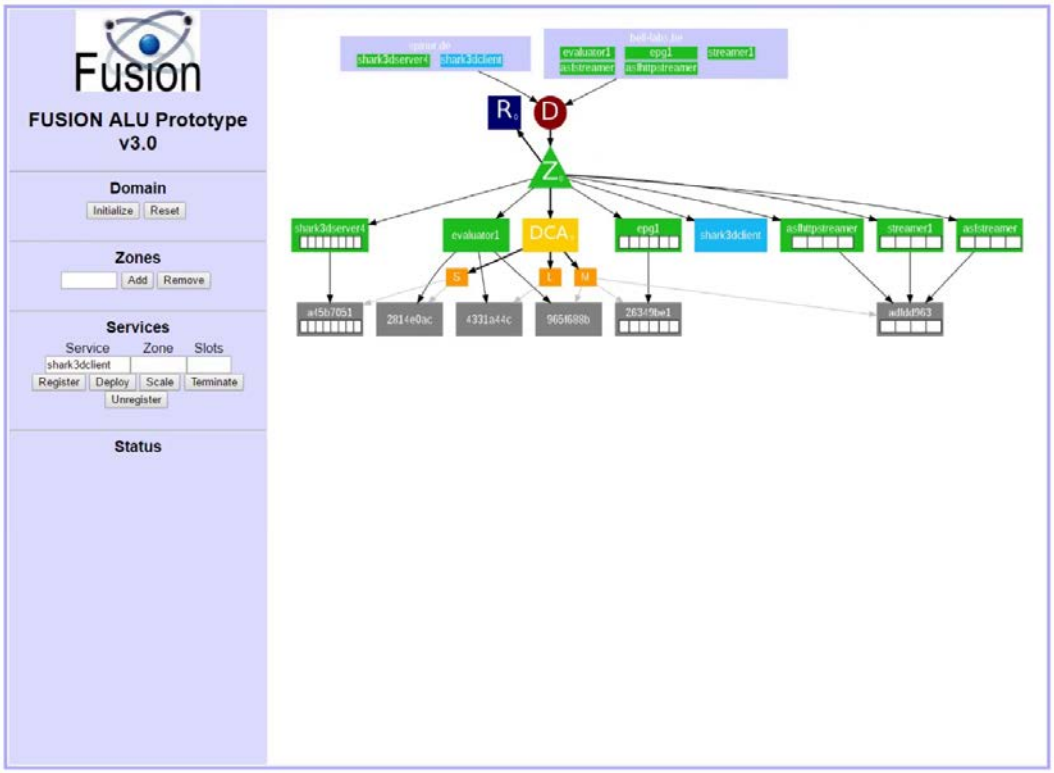


Figure 38. Screenshot of the deployment in the Virtual Wall zone

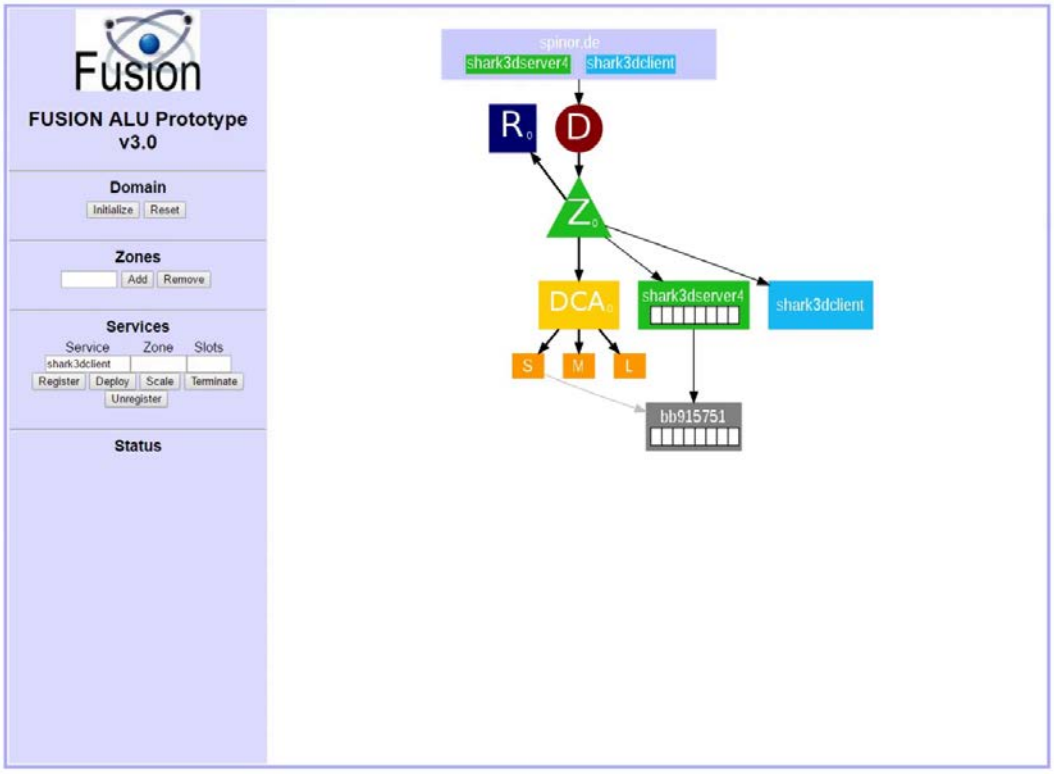


Figure 39. Screenshot of the deployment in the Spinor zone

5.2 Runtime environment evaluator services scenario

5.2.1 Testing plan

In this scenario, we validate and evaluate the evaluator services concept for assessing the feasibility and effectiveness of a particular runtime environment. In this section, focus on assessing the performance and cost-efficiency of a particular runtime environment, whereas in the next section, we specifically assess the presence of a particular hardware acceleration feature, through evaluator services.

For assessing a particular runtime environment, we will deploy three types of (type 1) evaluator services:

- Type 1: a simple generic feature-based evaluator, which mainly looks at the various runtime environment features as well as high-level service requirements, and provides a rough estimate on the feasibility as well as expected performance of a particular runtime environment.
- Type 2: a basic EPG-specific evaluator service, consisting of a significantly simplified version of the service, basically only containing the kernel operation. This will be used to profile a particular runtime environment by actively running this EPG kernel at maximum speed on the environment, and based on the resulting throughput provide a more accurate assessment of the performance (apart from the available runtime features).
- Type 3: a full EPG service, wrapped and configured to run in an evaluator service mode. In this version, the actual full service is being used in some evaluator mode for assessing the performance of a particular runtime environment.

We will assess these evaluator services on a number of testbeds and for a number of runtime environment types.

For the stage 2 cost-benefit analyzer evaluator service, we implemented a simple service that ranks all evaluations based on cost-effectiveness. We did not include networking features into this stage 2 evaluator services.

5.2.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple Evaluator Service
- Basic EPG Evaluator Service
- Full EPG Evaluator Service
- Video Decoder Service
- 2D EPG Rendering Service
- Streamer Service
- Thin Client

5.2.3 Results

5.2.3.1 Tradeoffs

The choice between a particular type of evaluator service is a matter of making a tradeoff between accuracy, simplicity and efficiency on the one hand versus runtime, development and deployment overhead and complexity on the other hand. Needing to create, provision and deploy full application-

specific evaluator services has an overhead compared to deploying a simple lightweight multi-service evaluator.

In the graph below, we show a break-down of the different evaluator services used in our scenario.

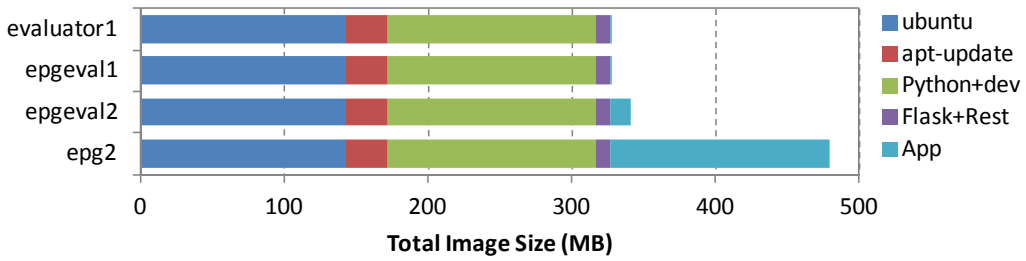


Figure 40. Breakdown of the different evaluator service container images

In the graph, *evaluator1* is an example of a type 1 evaluator service, *epgeval1* an example of a type 2 evaluator service, *epg2* an example of a type 3 evaluator service, and *epgeval2* somewhere in between a type 2 and type 3, being a stripped-down version of epg2, but will almost full functionality.

Image layers that are shared across the (evaluator) services include the base *ubuntu* layer, some package mgmt update, the python libraries as well as the Flask and Rest Python modules for easily implementing a RESTful server. Note that these layers could be optimized by using a more lightweight framework; on the other hand, when the Docker image layer building was done correctly, one could expect all these base layers to be present, resulting in no provisioning overhead for these layers.

The application-specific layer size grows significantly as more of the actual service implementation is being used, meaning that provisioning service-specific evaluator services may introduce provisioning overhead as well as provisioning delay. On the other hand, in the type 3 case, no additional provisioning delay would have to be accounted for when effectively deploying the service on the selected environments.

Next to a provisioning overhead, the type 2 and type 3 evaluator probes also have a higher runtime overhead, as they run various probes on the actual runtime environments for a period of time. Obviously, the longer the probing, the more precise the actual average and tail latency behaviour. On the other hand, if a probe needs to be able to produce an evaluation fairly quickly after deployment, one may want to provide an intermediate (but less accurate) result. Consequently, we implemented our type 2 and type 3 probes as background services that incrementally improve their results by running for increasingly longer periods of time. In our implementation, we also probe using 3 different rendering resolutions as an example of an application specific probe that is still capable of providing accurate results for different service configuration parameters.

As an example, we demonstrate for our type 2 evaluator probe the resulting achieved frame rate when running them for a particular period of time. This test was done on the following testbed platforms in our lab, running both native on the host as well as within a VM in our private OpenStack cloud environment:

Name	Host Hardware Platform	Guest Hardware Platform
Xeonv2	Intel Xeon E5-2690v2 2x10 core @ 3.0 GHz	Native host
Opteron	AMD Opteron 6174 2x12 core @ 2.2 GHz	Native host
Avoton	Intel Avoton C2750 1x8 core @ 2.4 GHz	Native host
XeonVM	Intel Xeon E5-2680v3 @2.5 GHz	4-core VM, 8 GB RAM

OpteronVM	AMD Opteron 6174 2x12 core @ 2.2 GHz	4-core VM, 8 GB RAM

Table 1. Various hardware platforms on which we deployed the probes

On each hardware platform, we created the following container software runtime environments:

Name	CPU Specifications	Guest Runtime State
Small (S)	1/2 vCPU	Idle
Medium (M)	1 vCPU	Idle
Large (L)	2 vCPU	Idle
Loaded (M)	1 vCPU	Loaded
RTLoaded (RTM)	1 vCPU	Loaded

Table 2. Various hardware platforms on which we deployed the probes

The first three represent three environments with varying CPU quota capabilities (ranging from half a CPU core to 2 CPU cores per container), with no other applications running in the background (i.e., no noisy neighbours). The latter two represent a "medium" runtime flavour with some significant background load, the latter of which has real-time capabilities (i.e., higher CPU priority).

For brevity, we only show the results of the type 1 probe for a single rendering resolution on the xeonv2 environment. As shown, running each probe for about 2-4 seconds already gives representative results for this application-specific kernel benchmark, especially for the idle and real-time environment types. The best-effort loaded environment type gives most variability. Note that the performance of this oversaturated environment is less than half of its idle or RT variant.

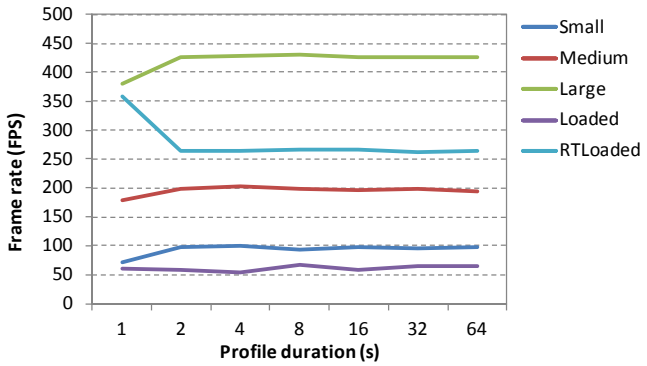


Figure 41. Reported frame rates for different application-kernel profiling durations for different container environment types on the xeonv2 hardware platform.

Below, we subsequently depict the profiling results for the 'Loaded' environment type for the various hardware testbed platforms. As expected, more variation is present on most platforms, especially the virtualized environment running on the Xeonv3 node. Notice also the performance different between the native Opteron and virtualized Opteron. This will be discussed further in Section 5.18.

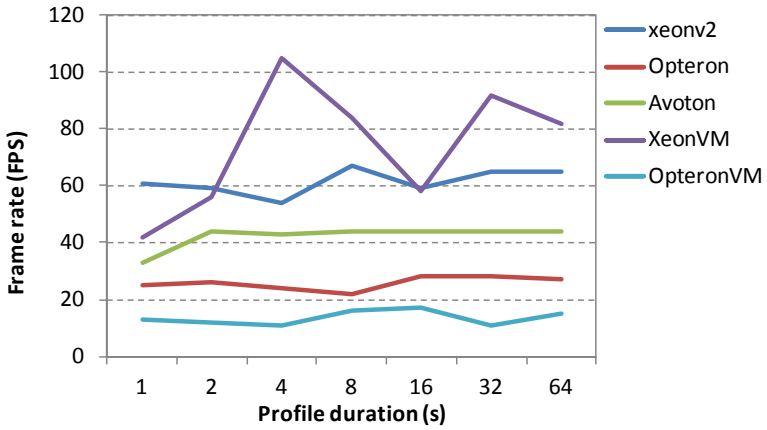


Figure 42. Reported frame rates for different application-kernel profiling durations for the 'Loaded' container environment type for different hardware platforms.

5.2.3.2 Baseline performance

In order to have a baseline, we first deployed the actual EPG service component on the various hardware platforms, to determine the actual maximum session slots. This is depicted below.

	Small	Medium	Large	Loaded	RTLoaded
Xeonv2	3	6	13	4	5
Opteron	0	1	2	0	1
Avoton	0	1	2	0	1
XeonVM	3	6	13	4	5
OpteronVM	0	1	2	0	1

Table 3. Actual supported session slots per hardware platform and environment type.

As can be observed, some hardware/software environments are too limited to even deploy a single session. In the following sections, we briefly evaluate each of the three evaluator types, and compare them with these results.

5.2.3.3 Type 1 evaluator results

The basic setup with a simple type 1 evaluator service, is depicted in the figure below, and involves the small, medium and large environment types, as well as all three service components of the EPG composite service graph. Although this evaluator provides a probe result for all three service components, the main focus in these scenarios is the EPG rendering service component.

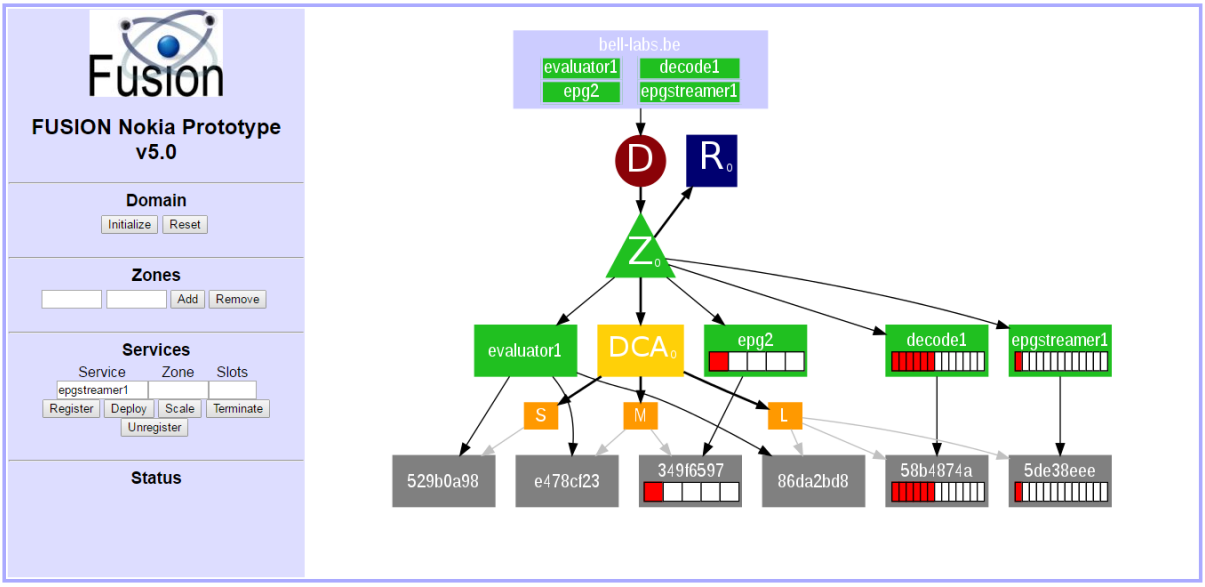


Figure 43. Basic scenario with a type 1 evaluator service.

This evaluator type is a general purpose static evaluator without any active probes for assessing the runtime environments. It uses basic knowledge about the environment provided by the zone manager while triggering an evaluation request, combined with some information fetched directly from the runtime environment (e.g., CPU type, clock speed, etc.), and uses this information for providing a rough estimate of supported session slots.

In this simple prototype, we use the clock speed to extrapolate based on some precalibrated data. Based on this, the evaluator decided to deploy the decode1 and epgstreamer1 services on the large environment, whereas for the epg2 service, the medium environment appeared most cost-efficient, with an estimated available 5 session slots.

Using clock speed and available vCPUs however is not accurate enough for estimating how many session slots a particular environment could support. For example, on the Opteron and Avoton environments, which have a much simple processor architecture, this results in an overestimation of session slots: based on clock speed alone, these environments should support about 3 session slots for the medium environment type, whereas in real life at best one session slot is feasible. External aspects such as noisy neighbours or other bottlenecks are also not considered. This could all be included in the extrapolation engine, but in the end, it may be simpler to use a type 2 or type 3 evaluator instead.

5.2.3.4 Type 2 evaluator results

In this scenario, an application kernel is deployed on the respective environments and its results are being used to better estimate the true potential of a particular environment for a particular (range of) service(s). Either an application-specific kernel could be used (which ideally should hit the same bottlenecks as the real application), or one or more generic application benchmarks could be used, in which case an additional interpolation step is required. In our prototype setup, we developed an application-specific kernel. The setup is depicted below. The type 2 evaluator is called *epgeval1*. Notice that we still use the type 1 evaluator for the other two service components.

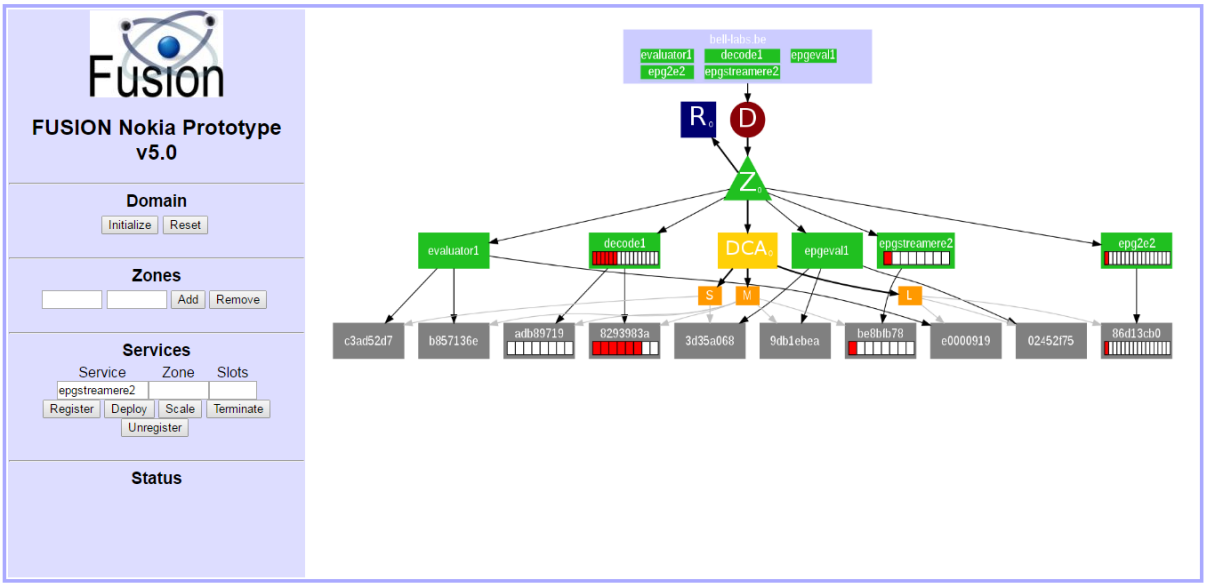


Figure 44. Basic scenario with a type 2 evaluator service.

To estimate session slot availability in each environment, it uses the results shown in the previous section, using the most recent stable frame rate results, dividing it by the target frame rate (e.g. 25 FPS), possibly subtracting some absolute or relative overhead. In this setup the *epgeval1* evaluator service determined the 'Large' environment to be most cost-effective, with an estimate of 16 session slots (i.e., floor(427/25)), which is a slight overestimation compared to the true value (i.e., 13 slots).

However, as this application kernel does not contain the full features of the EPG service component, it is logical for the value to be overestimated. In this prototype, we did not try to calibrate this result based on reference measurements, though this should obviously be done in a real implementation. Note that for the 'Small' and 'Medium' environments, the raw output would be just about 4 and 8 session slots, respectively, also slightly above the actual number.

Alternatively, this type of evaluator services could also be used as a probable upper-bound, or possibly even a lower-bound, depending on the behaviour and characteristics of the application kernel compared to the actual application. In fact, in case one could create or use two types of probes, one acting as a probable upper-bound, and the other as a probably lower-bound, this could provide very valuable information as well.

5.2.3.5 Type 3 evaluator results

The final type is to use the actual full service for probing a particular environment type. Two possible operation modes here are to either or not allow this evaluator probe to also trigger additional FUSION services via the FUSION resolver. For example, in case of the EPG rendering service component, the corresponding evaluator service could be configured to also connect to decoder services for receiving actual raw video streams. Although this would result in very accurate measurements, there is clearly also a potential huge performance and cost overhead in case the evaluator needs to be deployed on a wide range of environment types. An alternative approach, also adopted in our prototype implementation, is to build an evaluator mode into the service component, that uses internal dummy raw video sources instead.

As with the type 2 evaluator probe, we run the probe service as a background service, doing several measurements under different conditions (e.g., different resolutions, frame rates as well as active session slots). To estimate the supported number of session slots, we currently leverage the DynSlots script that dynamically updates session slot availability based on used slots and available resources. This reduces the number of tests to be done, but may be less accurate compared to actually monitoring the performance for different session slots.

The basic setup for validating this evaluator implementation in our prototype is depicted below.

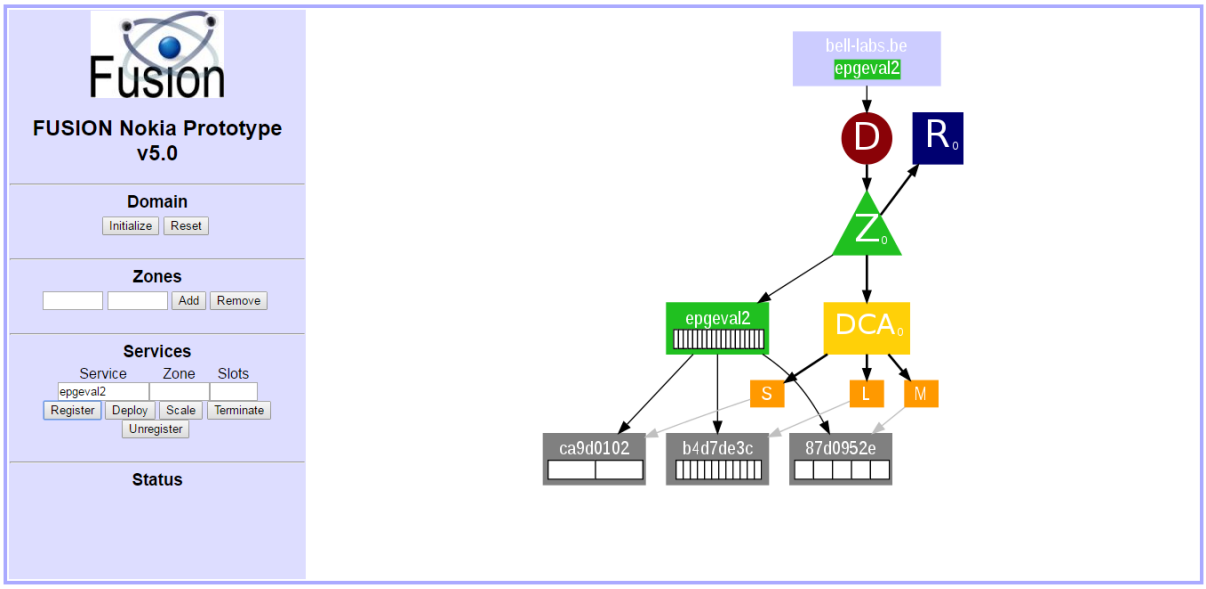


Figure 45. Basic scenario with a type 3 evaluator service.

The predicted session slot availability in this case is visualized as actual session slots here. For the three example environment types, this probe estimates 2, 5 and 12 available slots, respectively, which seems to be a conservative lower-bound for the xeonv2 hardware platform, where actual full measurements indicate 3, 6, and 13 available session slots in total.

5.3 GPU rendering acceleration evaluator services scenario

5.3.1 Testing plan

In this scenario, we assess the capabilities of both the FUSION orchestrator and DCA prototypes for being able to support GPU-accelerated execution environments, as well as the evaluator service for selecting an appropriate runtime environment based on features (and performance).

5.3.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple Evaluator Service
- Video Decoder Service
- 3D cube/sphere EPG Rendering Service
- Streamer Service
- Thin Client

5.3.3 Results

The final state of this scenario is shown below. We first configured a DCA environment consisting of a regular node as well as a GPU-enabled node. Then we deployed a 3D EPG composite service graph onto this simple environment.

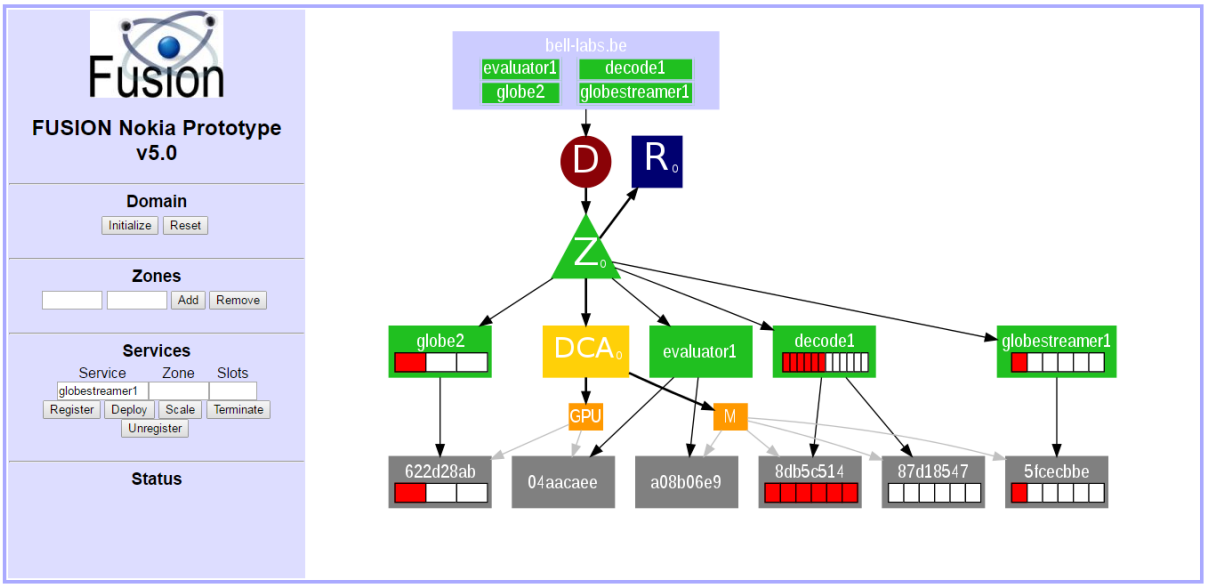


Figure 46. FUSION zone with a GPU-accelerated execution node, mixed service components, and a simple evaluator service.

For this scenario, we used a simple evaluator service, which assesses basic environment capabilities and makes a rough session slot estimation. As the GPU environment type ('GPU') is much more expensive than the default medium environment type ('M'), both the *decode1* and *globestreamer1* service components are deployed on the default node. The *globe2* service component however requires a GPU-accelerated execution environment, and as such is deployed on the GPU-accelerated node.

5.4 Hardware encoding acceleration evaluator services scenario

5.4.1 Testing plan

The evaluator service is a slightly modified version of the thin-client game service as described in section 3.3.

For this test the evaluator service is now prepared in the following way:

- One actual evaluator is deployed to check a machine which supports NVENC video encoding
- One actual evaluator is deployed to check a machine which does not support NVENC video encoding
- The actual test now deploys the evaluator services and queries the evaluation results: A first evaluator service for the first machine is started, and a second evaluator service for the second machine is started. Then evaluation requests are sent to both evaluation services.

5.4.2 Involved components

- Orchestrator
- Shark 3D evaluator service
- Lobby software

5.4.3 Results

Depending on which evaluator service was addressed, the number of returned session slots was different. The evaluator service which targeted the machine not supporting the NVENC encoding

reported zero available session slots while the other machine with NVENC support available reported a non-zero number.

The following is a sample response of the Shark 3D based evaluator service request reporting zero session slots, because the node has no NVENC capable hardware.

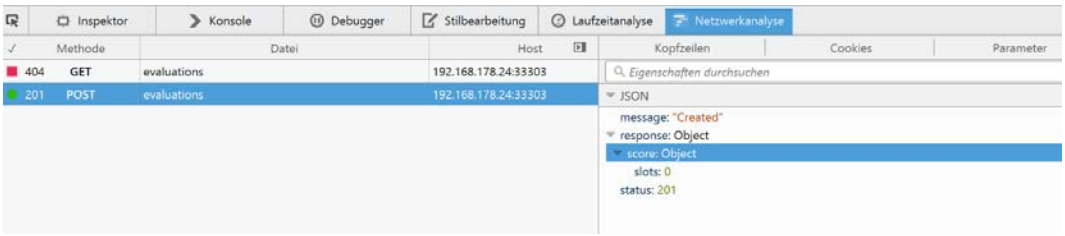


Figure 47. Evaluation request returning zero session slots

The following is another sample response of the same evaluator service reporting eight session slots, because the node has an NVENC capable hardware.

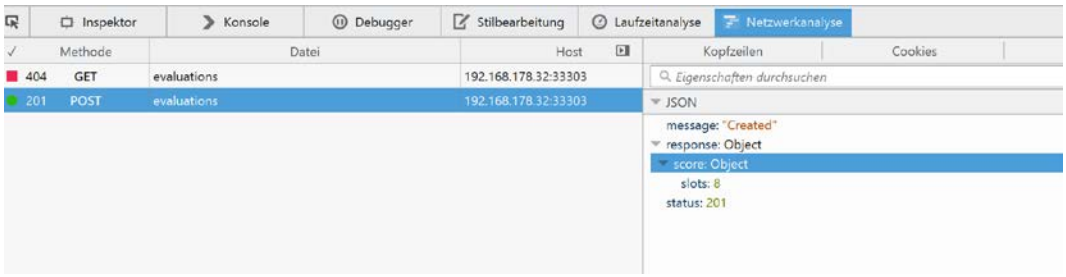


Figure 48. Evaluation request returning eight session slots

5.5 Service deployment optimization

5.5.1 Testing plan

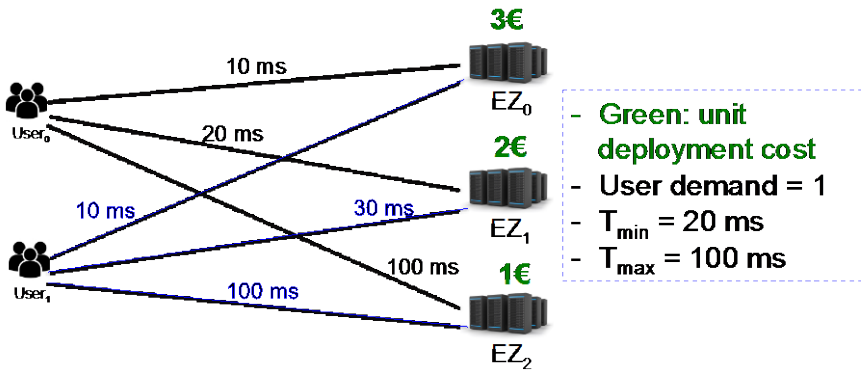


Figure 49. Placement deployment scenarios

We have set up an environment to deploy EPG service with our placement optimization algorithm. The testbed includes 3 execution zones and two users. Network latencies between users and EZs are shown in Figure 49. We assume that each user requires only 1 session slot and the cost of deploying a session slot in each EZ are different – the cheap EZ is located far from the users (Figure 49). We use our utility function (see deliverable 4.3 for more detail) to convert latency into a utility score and evaluate QoS based on this score. We use Tmax = 100 ms to force users to only choose EZs which have latency less or equal to 100 ms. On the other hand, Tmin = 20 ms would mean that for this service, users cannot see any difference in QoS if the latency is less or equal to 20 ms. That is, in this testbed, user0 sees no difference if he/she uses the service in EZ0 or EZ1. For the deployment scenarios, we need to specify the maximum budget cost, the placement algorithm will then try to maximize the utility score (maximize QoS) given that cost as a constraint.

5.5.2 Involved components

- Orchestrator
- Placement component
- EPG service

5.5.3 Results

5.5.3.1 Scenario 1: budget cost = 2€

For each scenario, we simply go to terminal and type a command which states how much is the budget cost as the Figure below:

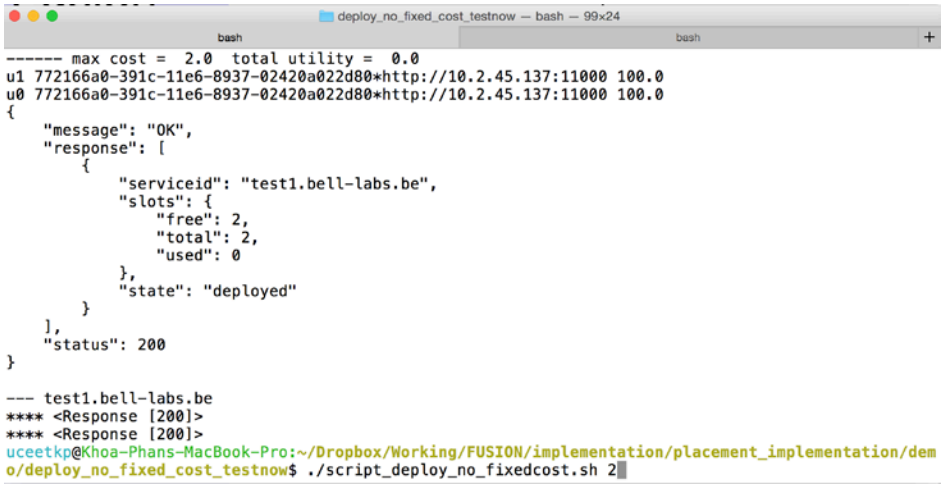


Figure 50. Deployment command with budget = 2€

Given the budget of 2€, we can easily see that we can only deploy 2 session slots for the two users at EZ2. With 100 ms latency, both two users will experience a not-so-good QoS. We capture a screen shot of Web GUI to monitor service deployment status as below:

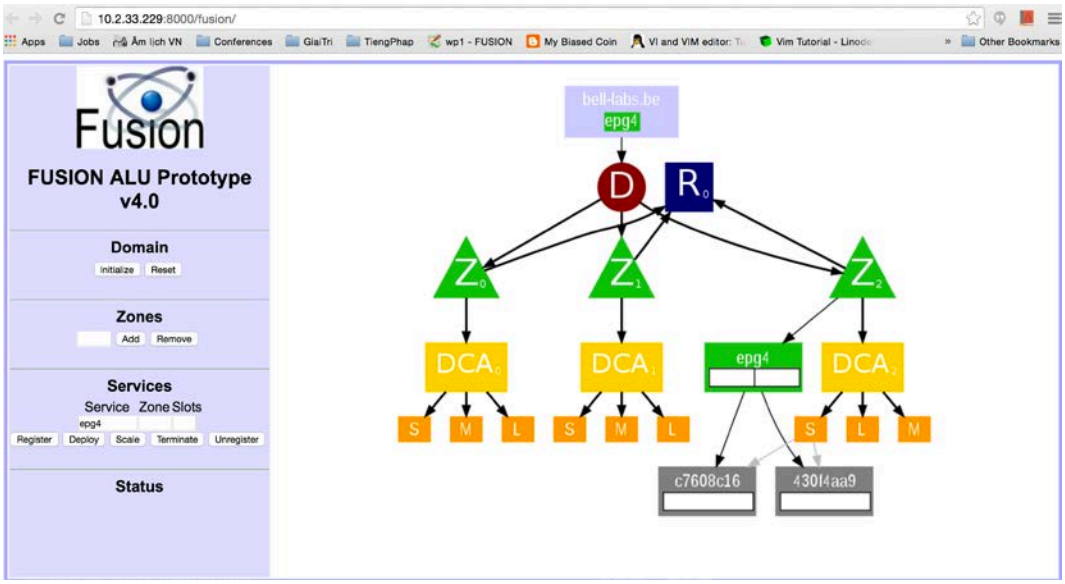


Figure 51. Deployment solution with budget cost = 2€

5.5.3.2 Scenario 2: budget cost = 3€

In this scenario, we increase the budget to 3€, which has more chances to have a better deployment solution. A screenshot of the solution is shown below:

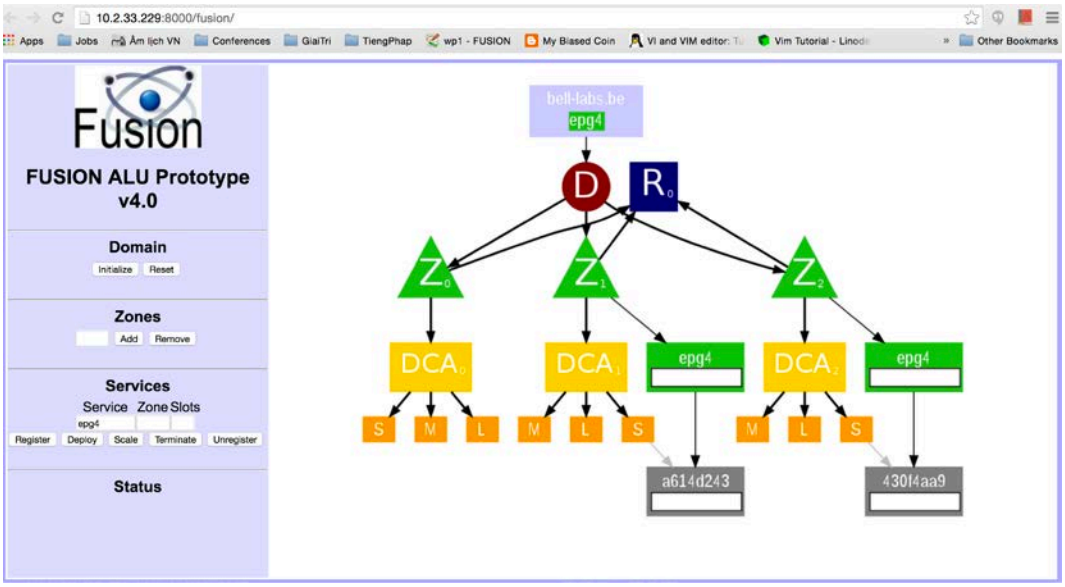


Figure 52. Deployment solution with budget cost = 3€

With more budget, we now can deploy one session slot is EZ2 and the other session slot in EZ1. As EZ1 is closer to both two users, we will see better QoS for the user connecting to EZ1.

5.5.3.3 Scenario 3: budget cost = 5€

With the budget cost = 5€, we can see even a better placement solution for the two users.

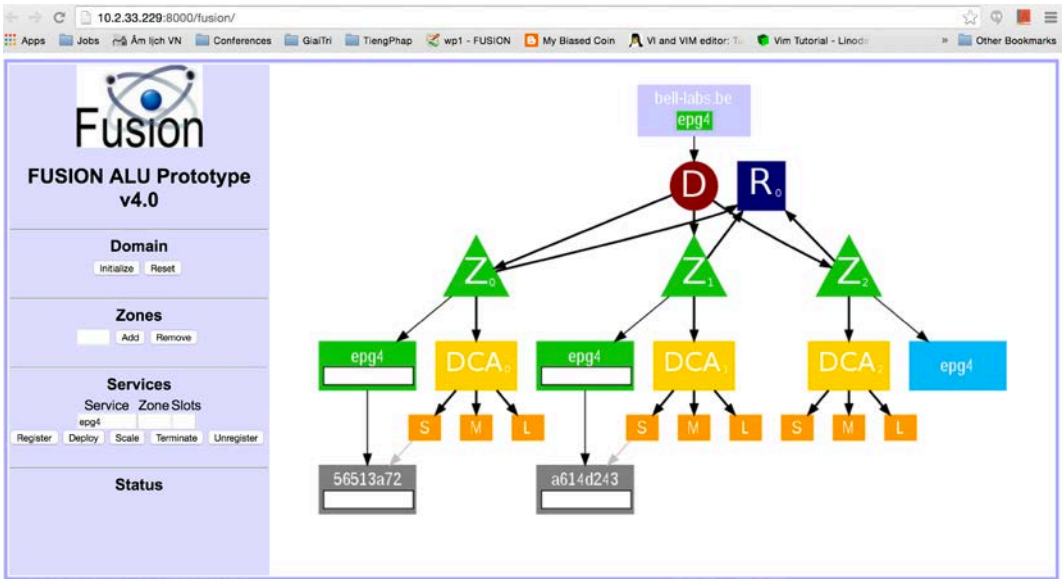


Figure 53. Deployment solution with budget = 5€

We can see now the service is deployed at EZ0 and EZ1. For this case, the best QoS for both users will be: user0 connects to EZ1 (20 ms) and user1 connects to EZ0 (10 ms).

5.5.3.4 Scenario 4: budget cost = 10€

We continue to increase the budget cost to 10€. However, the placement solution we have is the same as in the scenario 3. In fact, we can deploy both session slots in EZ0 which costs 6€ (still less than the

budget). However, as this solution does not help to improve QoS but cost more than in the scenario 3, our placement algorithm decides to keep the same solution as in the scenario 3 (minimizing the deployment cost with the same (maximal) utility score).

5.6 Session slot resource sharing for media applications

5.6.1 Testing plan

To test improvements achieved by the Shark 3D engine redesign discussed in chapter 2, a few experiments were designed. The single-user game scenario is employed for testing as being a typical media application. The setup was as follows: One PC acting as server, instantiating a state and a view for each incoming client.

The lobby interface described in section 3.6 was used to start the scenario. This interface acted as a management service to execute the logic normally handled by a portal application, i.e. processing incoming requests for new services from the clients and distributing them over the servers by returning the respective IP. At the level of our experiments, this service routed all client requests to the server where the measurements were performed.

Two series of experiments were carried out: In the first, for each client, a separate instance of the Shark 3D engine was started to measure the base line; in the second the optimization was used and each client was assigned his own private session but within a common instance of Shark 3D. Both graphics memory and main memory consumption were measured for one to eight clients connected to the server.

5.6.2 Involved components

- Orchestrator
- Resolver
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.6.3 Results

5.6.3.1 Memory usage

The results of run-time memory usage are shown in the following figure. As the memory of the graphics card used was about 4000 MB, it was able to handle up to five individual instances of the Shark 3D engine. It is noticeable from the graph that for the fifth instance, the limit of the GPU memory was being approached, swapping some data into the CPU memory instead. Therefore, the consumption of memory for this specific instance was less than the previous ones, with respect to GPU memory but higher in terms of that of the CPU, resulting in the same total amount of memory used. Launching a sixth instance of the application simply failed. As for the shared instance, it was able to handle at least eight sessions, with relatively much smaller increases in both CPU and GPU memory usage.

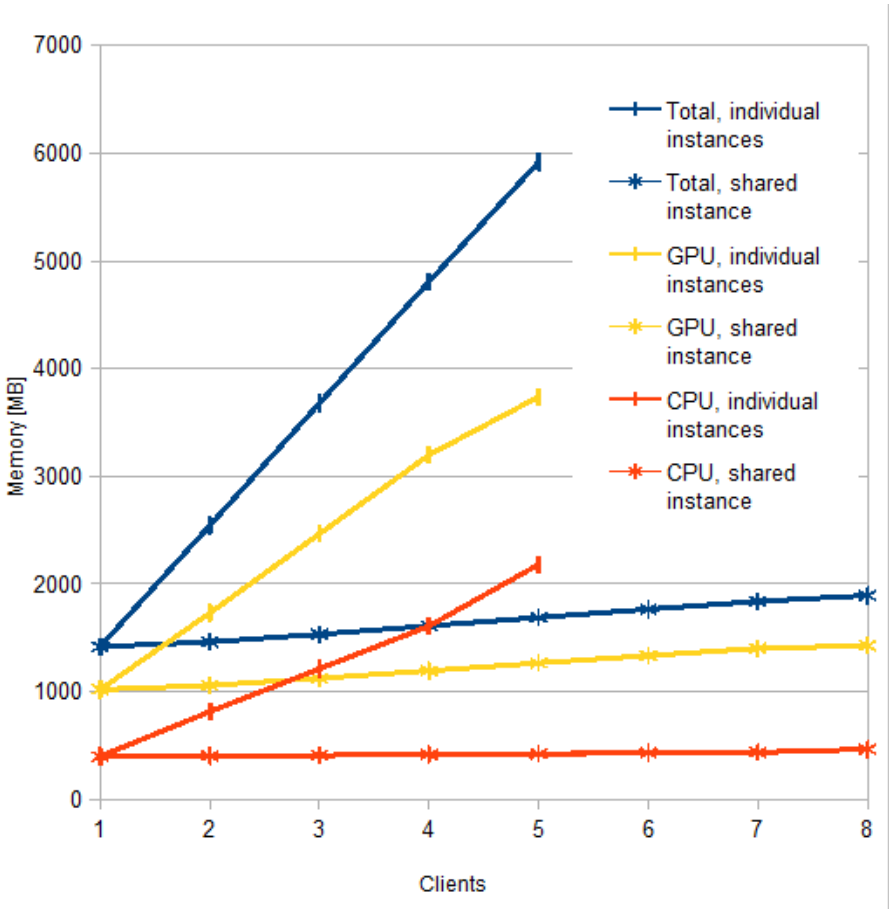


Figure 54. Memory usage of session slots with vs. without SOIM resource sharing architecture

For information about storage memory usage see section 5.1.3.1.

5.6.3.2 CPU load

The Shark 3D based services run always with the full load of one (virtual CPU core), independent from the number of session slots used. A higher number of used session slots does not increase the load, but reduces the frame rate available to all users. See also section 3.3.3 for additional details.

5.7 Automatic session-slot based scaling

5.7.1 Testing plan

As part of the FUSION orchestration prototype, we also implemented a single automatic service scaler at the domain orchestrator level. The autoscaling policies can be defined in the service manifest and currently controls a two key aspects:

1) Define the minimum and maximum number of session slots that should be available in a domain for some service;

2) Define the distribution across environments and zones;

5.7.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple Evaluator Service
- Video Decoder Service

- 2D EPG Rendering Service
- Streamer Service
- Thin Client

5.7.3 Results

In this scenario, we functionally validate the basic scaling features of the prototype and manifest. We start by registering an evaluator service. As in the manifest of this evaluator, it is specified to automatically deploy in all available runtime environments, by registering this service in the domain, the scaler, who is notified automatically during registration, will automatically start deploying an instance of this service on each environment. The final state of this registration step is depicted in the figure below.

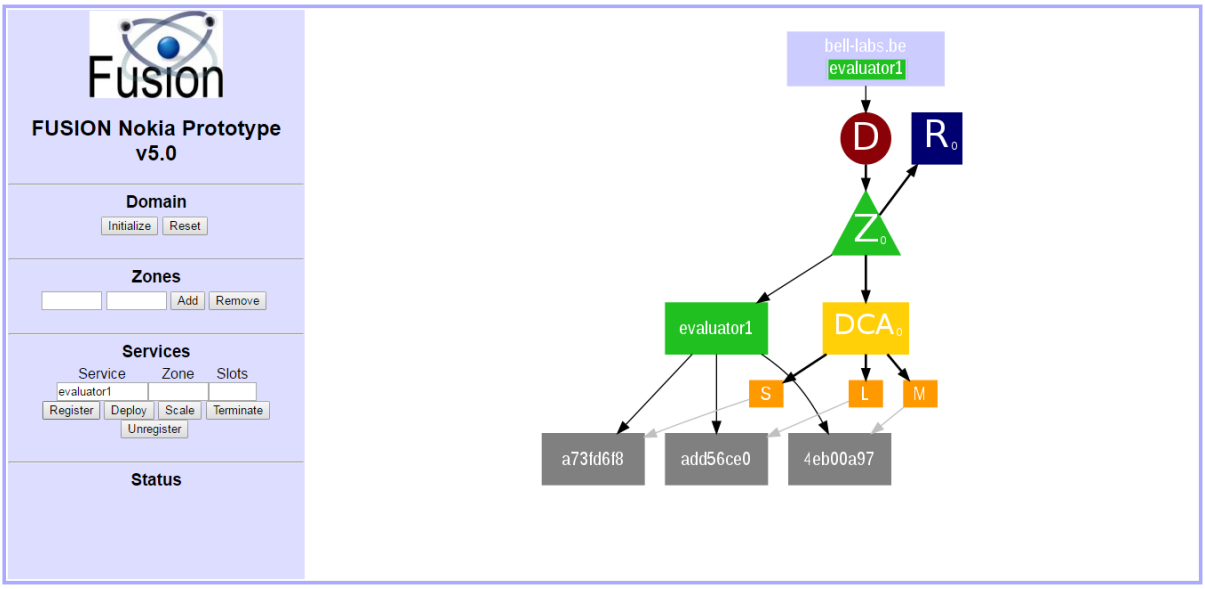


Figure 55. Registering an evaluator service manifest automatically triggers the deployment across all available execution environments.

Next, we register a new execution zone, this time with a single runtime environment. The domain scaler service is also notified of this event, and as the evaluator policy is to deploy in all environments, the scaler service will automatically launch an additional instance of the evaluator in the environment(s) of this new zone, as depicted below.

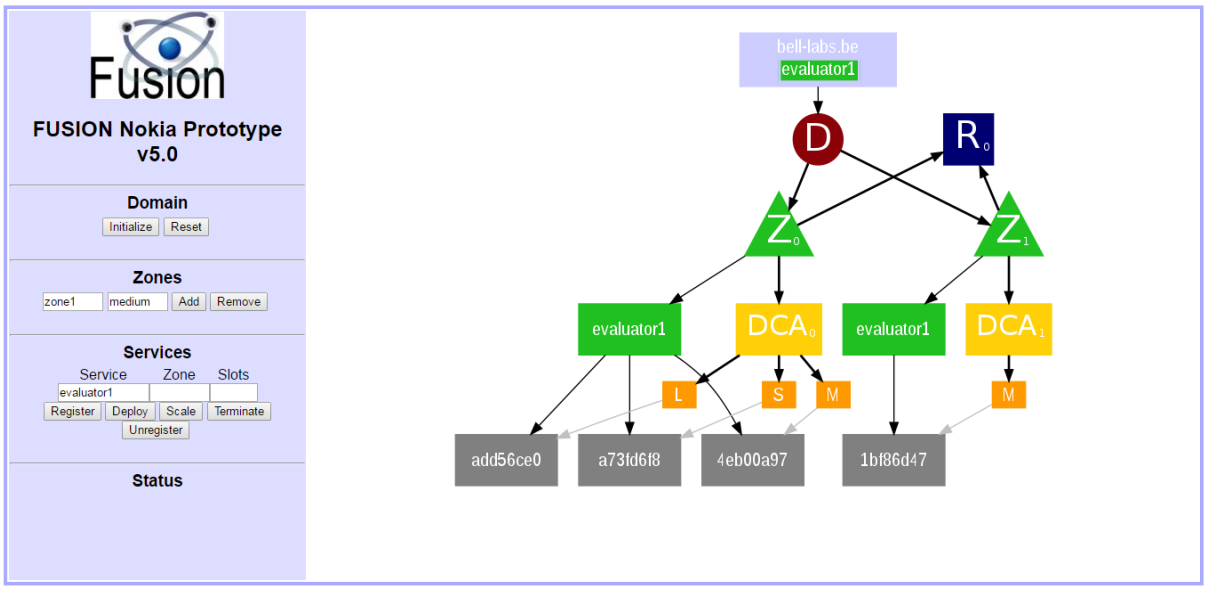


Figure 56. Registering a new execution zone automatically triggers the deployment of an additional evaluator service instance in the new execution environment.

Next, we unregister the zone again, which also causes the evaluator instance to the destroyed again. We return back into the earlier state (not shown here). Next, we register a video decoder service. As in the manifest it is specified to have at least 6 available slots, the scaler will automatically deploy a single decoder instance (not shown here). We then also register an EPG rendering service and streamer service. As the EPG rendering service currently automatically allocates and consumes 6 session slots of the decoder service, the scaler will automatically deploy new instances as the available session slots get depleted. The final state of all this is depicted below.

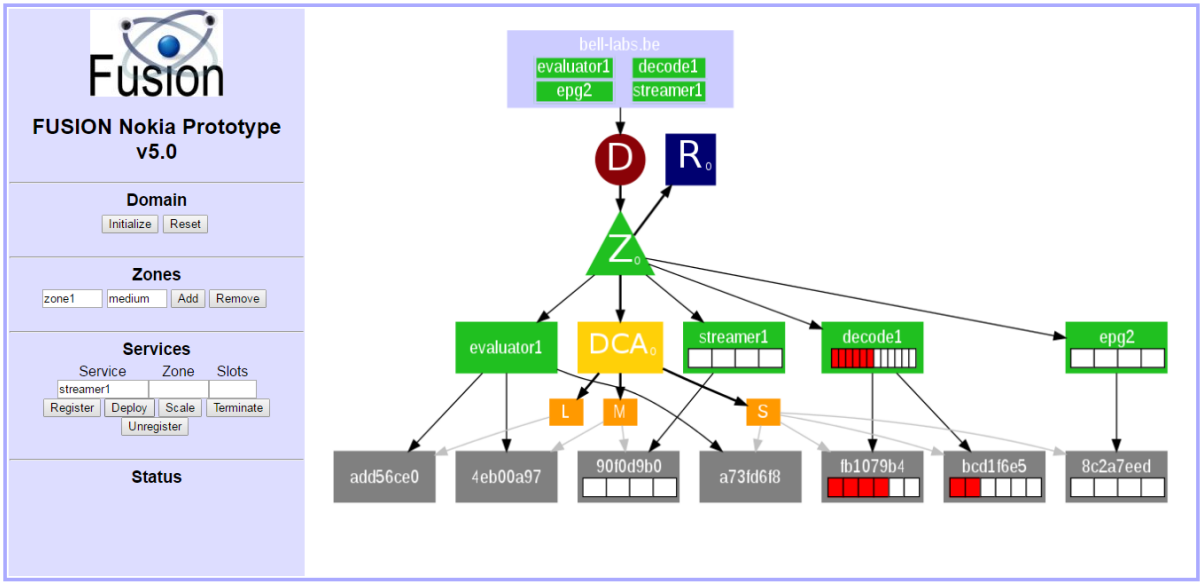


Figure 57. Registering an EPG service and streamer service triggers the autoscaling of an additional instance of the decoder service.

Finally, when 3 clients connect to the composite service, 3 out of 4 session slots of the EPG service are consumed. As in the manifest it was specified that at least 2 slots need to be always available, the scaler automatically deploys an additional instance of the EPG service. This however in its turn triggers the scaler to also deploy an additional decoder instance, as fewer than 6 slots are available. The final state is depicted below.

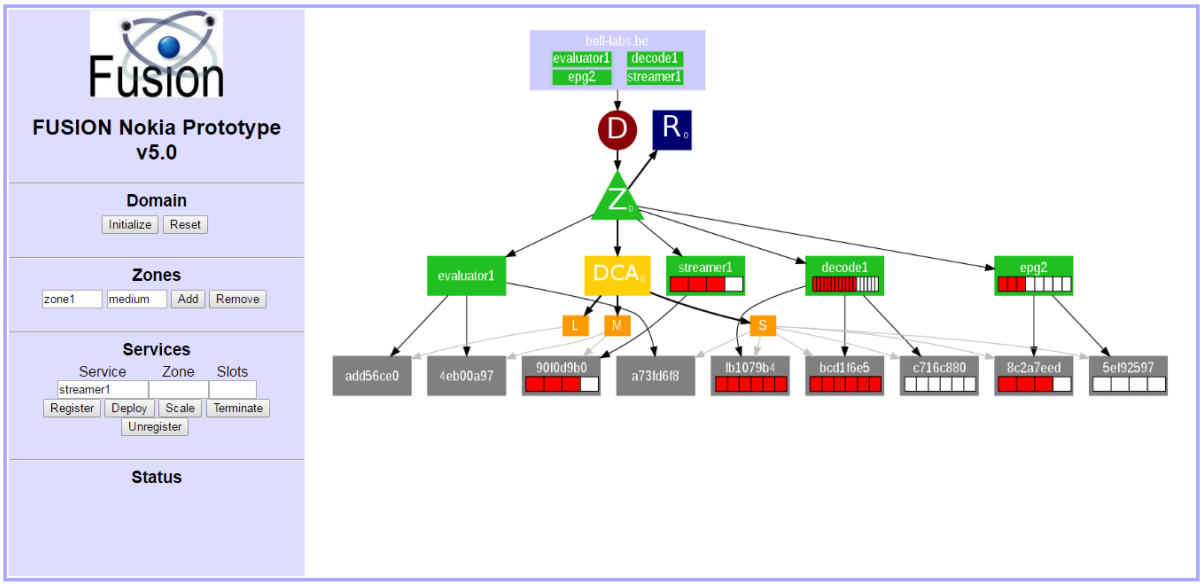


Figure 58. Connecting 3 clients to the composite service triggers an auto-scale-out on the EPG service components, which in its turn also triggers a second auto-scale-out on the decoder service component.

5.8 Stateless single-user EPG composite service

5.8.1 Testing plan

In this first composite service test scenario, we assess how FUSION can handle simple stateless composite services such as the simple single-user EPG composite service graph described in Section 3.1.4, consisting of a decoder, rendering and encoding/streaming service component.

As many of these service components stream raw video frames in real-time, efficient inter-service communication mechanisms can have significant impact on the overall performance (i.e., resource utilization & latency).

As such, in this scenario, we will both functionally validate the interworkings of the end-to-end prototype, as well as perform particular performance tests w.r.t. inter-service communication acceleration.

5.8.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple Evaluator Service
- Stateless Video Decoder Service
- Stateless 2D EPG Rendering Service
- Stateless Streamer Service
- Thin Client

5.8.3 Results

In this scenario, the client first requests an EPG streamer service from FUSION, which in its turn requests and connects to an EPG service instance, also by making a FUSION service request. The EPG

rendering service instance in its turns makes several FUSION service requests to an appropriate video decoder service for each video stream to be decoded.

The following screenshot shows the prototype state after all respective components have been deployed and a single client is connected to the epgstreamer service component. This streamer requires one EPG rendering component. The EPG rendering session requires 6 video streams to be decoded, and as such requires 6 session slots from the decoder service.

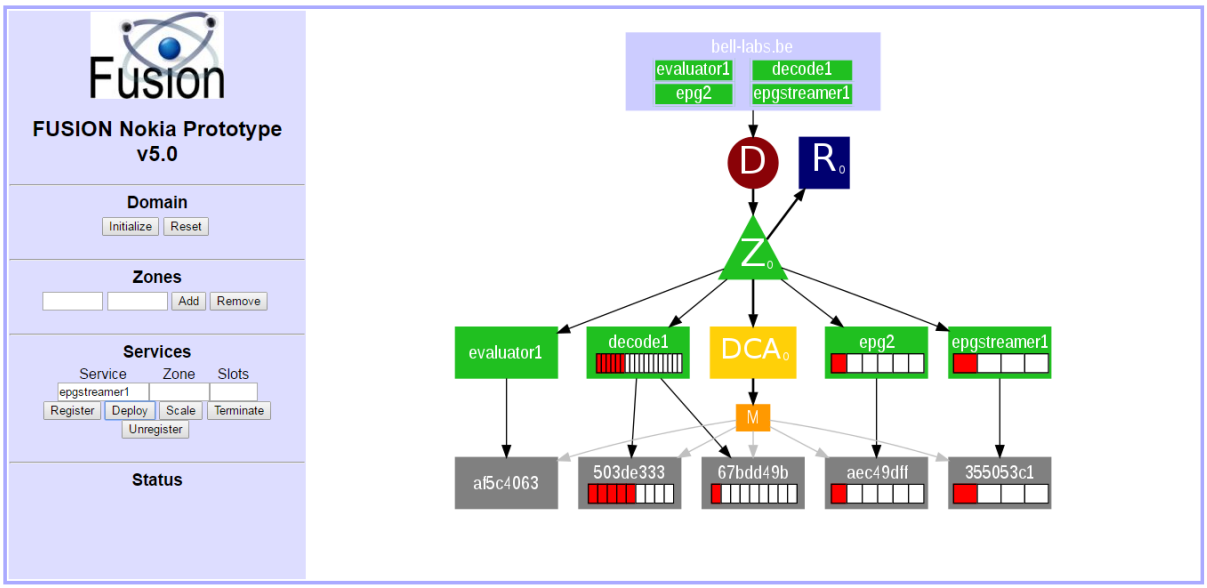


Figure 59. Simple 3-stage composite service

Next to this functional validation, we also performed a number of performance tests, with and without shared memory (SHM) inter-process communication (IPC) acceleration support. More information on this can be found in Deliverable 3.3.

Without SHM support, each service component, wrapped in its own Docker container, will send and/or receive raw video frames over a standard TCP socket. At 720p and 25 FPS, this results in an average throughput of about 400 Mb/s per stream (and in our example, we have 7 of these raw streams). As demonstrated in Deliverable D3.3, sending the entire raw video frame in a single *send()* command rather than per scanline, can result in significant throughput and latency improvements, which is what we used as baseline results.

With SHM support (assuming each connected service component is running on the same machine), these raw frames be streamed by exchanging them via shared buffers. As we demonstrated in Deliverable D3.3, this can be done via a true zero-copy mechanism (i.e., SHM-0), with some built-in support of the applications. In this scenario, we used the somewhat less efficient SHM-1 variant, where still at least one copy is required. As such the results presented below can still be improved.

First, the CPU utilization results are shown below for all three active service components, with and without SHM acceleration, see 'SHM-1' and 'TCP', respectively. For comparison, we also compare with deploying the entire service as a single fat service component (i.e., 'FAT').

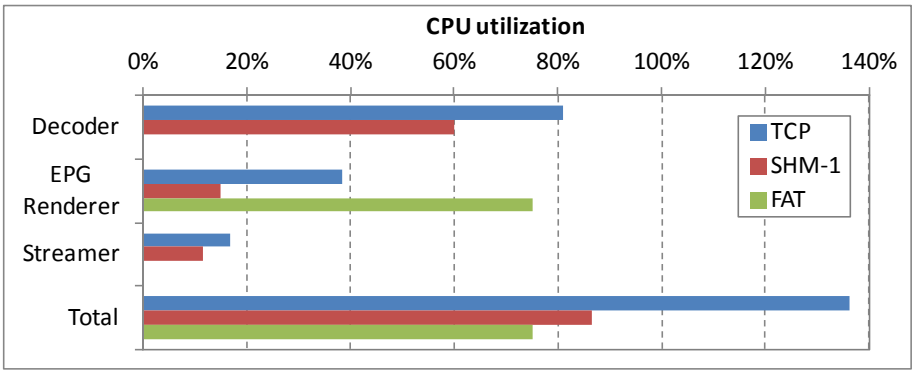


Figure 60. CPU utilization of the EPG composite service with and without SHM acceleration, and compared to a non-composite service

For the Decoder service, the CPU utilization can be reduced by about 25%. For the EPG rendering service, the CPU utilization can even be reduced by 60%. This is because this service component has 6 RAW input video streams, and the SHM-1 method specifically avoids memory copies for reading the raw video frames; the decoder service however in the SHM-1 method used in our test setup still has the overhead of an explicit memory copy (though without the TCP stack overhead); in case of the SHM-0 method, the CPU utilization benefit for the Decoder service would be even higher. Secondly, the EPG rendering itself is very lightweight compared to video decoding and/or encoding. As such, for the streaming component, the reduction in CPU utilization is only about 12%, as the overhead of receiving the raw frames is small compared to real-time encoding the frames.

Notice also that the SHM-1 composite service implementation is almost as efficient as a single fat implementation. Consequently, for the additional flexibility, reuse and possible explicit hardware specialization, we do not lose much efficiency, even with this default software implementation.

Using the Intel Performance Counter Monitoring Toolkit [Intel16], we also measured the impact on the actual total memory bandwidth utilization. The results are depicted in the Figure below. About 50% reduction in system memory pressure is observed with SHM-1 versus TCP. In case of SHM-0, the memory throughput would reduce even further. This potential for improvement can be seen when comparing with the fat non-composite implementation.

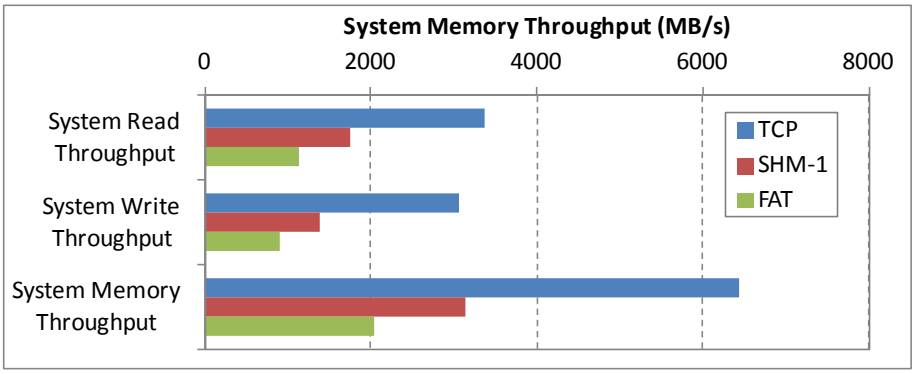


Figure 61. System Memory Throughput of the EPG composite service with and without SHM

The final results shown are with respect to the overall roundtrip latency between the user pressing a key and the result being visible after rendering, encoding and streaming. At 25 FPS, the average roundtrip latency is dominated by the half-frame delay (i.e., 20 ms), as discussed in more detail in Section 5.20. As can be observed, the roundtrip latency is reduced by about 1 ms via the SHM-1 method compared to TCP streaming. Running the service as a single fat service has a similar (though slightly worse) roundtrip latency. We did not investigate further.

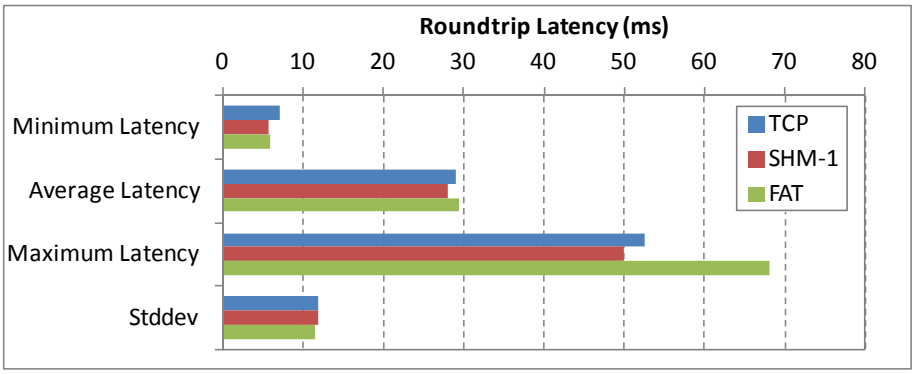


Figure 62. Interaction roundtrip time of the EPG composite service with and without SHM

5.9 Stateful augmented reality composite service

5.9.1 Testing plan

In this scenario, we investigate how our prototype orchestrator and service components can handle more complex stateful composite services. We call a composite service stateful when specific session slots of particular service components need to be connected to each other for implementing the overall service. This contrasts with the first composite service test case, where an EPG streamer session slot could connect to any EPG rendering slot and any decoder session slot. The composite service graph is detailed in Section 3.2.2.

5.9.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple Evaluator Service
- Stateless Video Decoder Service
- Stateful Video Decoder Service
- Stateful Augmented Reality Rendering Service
- Stateful Streamer Service
- Thin Webcam Client
- Optional: External Coordinator Service

5.9.3 Results

Being only a single-user application, the coordinator component functionality could be either integrated as part of the thin client application, or as an external service (to hide the complexity at the thin client). In this use case, the bootstrapping of such composite service can be split into four main steps:

1) In the first step, the client connects to the coordinator component, either locally, via a direct connection or via a FUSION service request.

2) Next, the coordinator component makes three subsequent FUSION service requests as a third party entity, in the following order: (i) the streamer, (ii) the rendering and finally (iii) the decoder. Alternatively, if the resolver supports resolving an entire service graph, it can try to resolve the

entire graph at once. Note however that the client should be considered for every service component, as it will directly communicate with all three components:

- The Client sends its live webcam video stream directly to the stateful decoder service. Note that you could also treat the client as a streaming server, but then you run into TURN/STUN issues.
- The Client sends its feedback events (e.g., key presses, etc.) directly to the rendering service component.
- The Client receives the resulting (video or overlay) stream directly from the streamer component.

3) Once the coordinator received the most optimal endpoints for all service components, it will start a stateful session slot with all three components by connecting to each endpoint. During this brief connection, it will configure each session (thus making them stateful) of each service component. During this configuration step, each service component will properly set up itself. This includes allocating e.g. a stateful TCP port onto which it will listen on for handling the actual stateful data connection later on. This information is returned to the coordinator. As the coordinator needs this information for setting up the other components, the order of configuration is important. Consequently, the coordinator will configure the components in the reverse order: first it configures the decoder session, then the rendering session (passing along the stateful TCP port that the decoder allocated for this session), and finally the streamer session (passing along the stateful TCP port that the renderer allocated for this session).

   As the client itself also directly connects to the stateful sessions of each service component, the client as the last component is returned the stateful TCP ports that are allocated for its session.

4) The client can now directly connect to all three components, thereby effectively starting the actual stateful client session. When the client disconnects, the stateful session is deallocated.

We implemented this functionality and validated this with our prototype. The final state is shown below. Components *decode2*, *aurea2* and *streamer2* are stateful versions of the corresponding service components. The *aurea2* service component also itself connects to a stateless *decode1* service component for decoding a generic video stream, as part of the service.

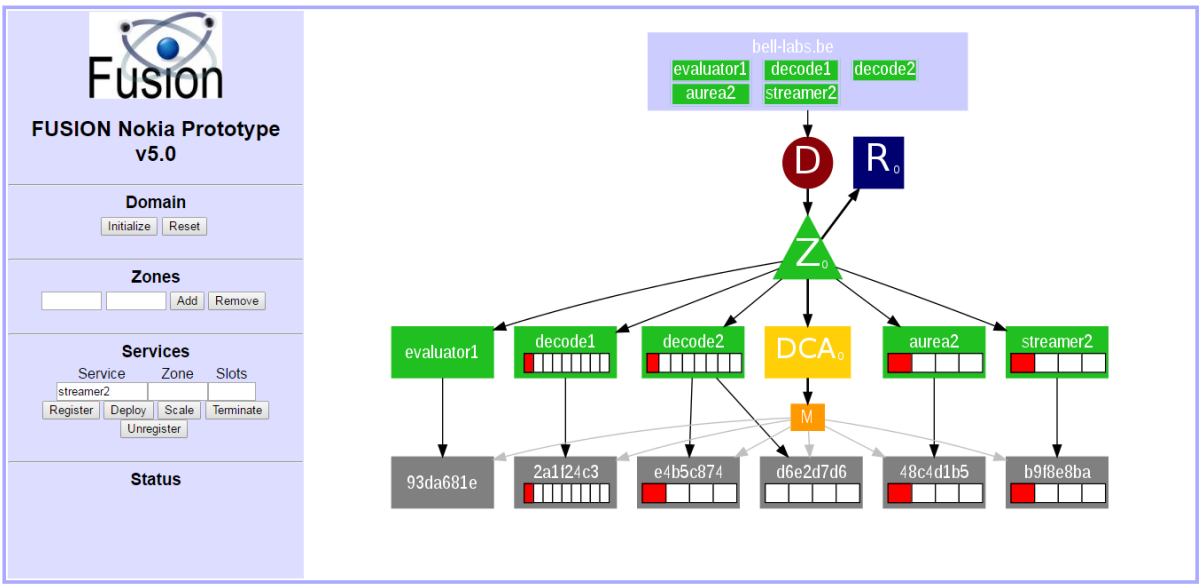


Figure 63. Stateful Single-user Composite service

Apart from this functional validation, we also performed a number of performance evaluations. Again, we compared the efficiency three different implementations, namely with SHM inter-process

communication acceleration ('SHM-1'), with standard TCP streaming ('TCP'), and implementing the functionality as a single fat service rather than a composite service ('FAT'). Note that in case of the single fat service, we used an internal muxing and demuxing process to be able to implement this service as a stateless service, reusing the single socket for multiple data streams.

First, the CPU utilization results are shown below. The overhead of the additional demuxing in the service is visible in the CPU efficiency results.

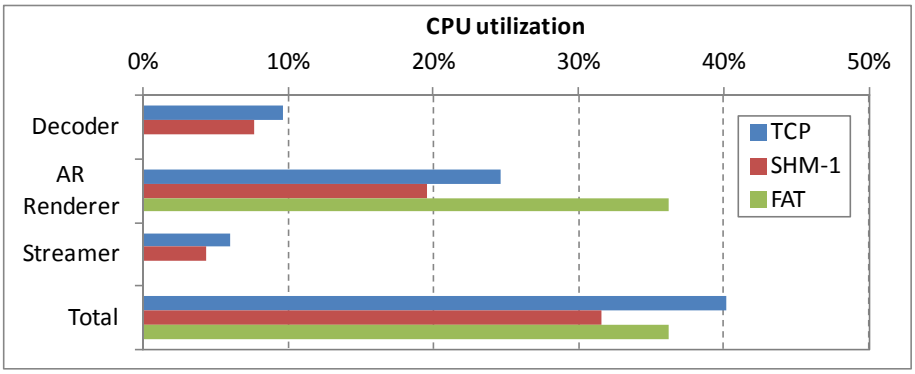


Figure 64. CPU utilization of the stateful augmented reality composite service with and without SHM acceleration, and compared to a non-composite service

Next, the overall memory throughput results are shown below. As expected, SHM-1 performs much better than TCP, and relatively close to the fat service implementation that does not require these additional copies.

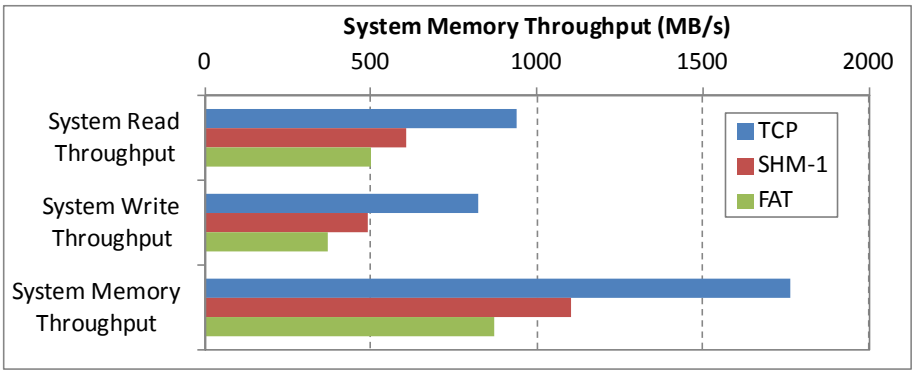


Figure 65. System Memory Throughput of the Augmented Reality composite service with and without SHM

Finally, the roundtrip latency results are depicted in the figure below. SHM-1 here performs much better than TCP; the fat service here appears to suffer substantially from the additional muxing/demuxing. An alternative implementation of this fat service would be to use the same stateful service mechanism, implemented in the individual service components, but applied to the fat service and client.

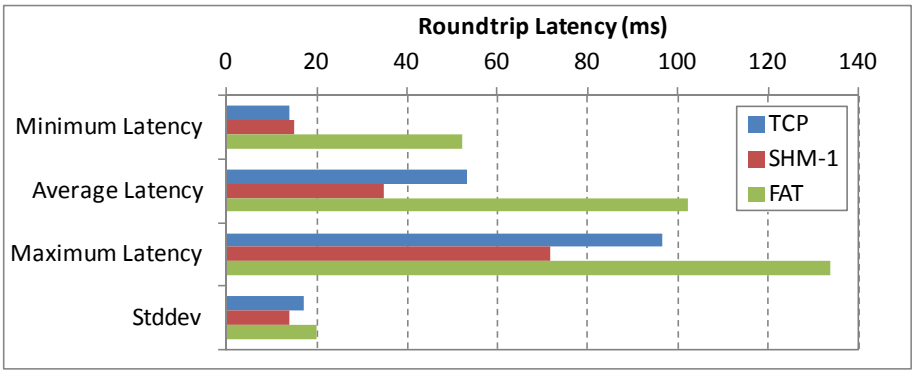


Figure 66. Interaction roundtrip time of the AR composite service with and without SHM

5.10 Lobby controlled multi-user stateful composite service

This scenario is based on the setups described in 5.1.

5.10.1 Testing plan

As discussed in section 3.6, the lobby software can launch different composite service scenarios:

- A game simulation server service connected with multiple game renderer services.
- A dashboard synchronization server service connected with multiple dashboard renderer services.

5.10.2 Involved components

- Orchestrator
- Resolver
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.10.3 Results

Based on the scenario requirements, the lobby software first requests suitable FUSION Shark 3D simulation and rendering services.

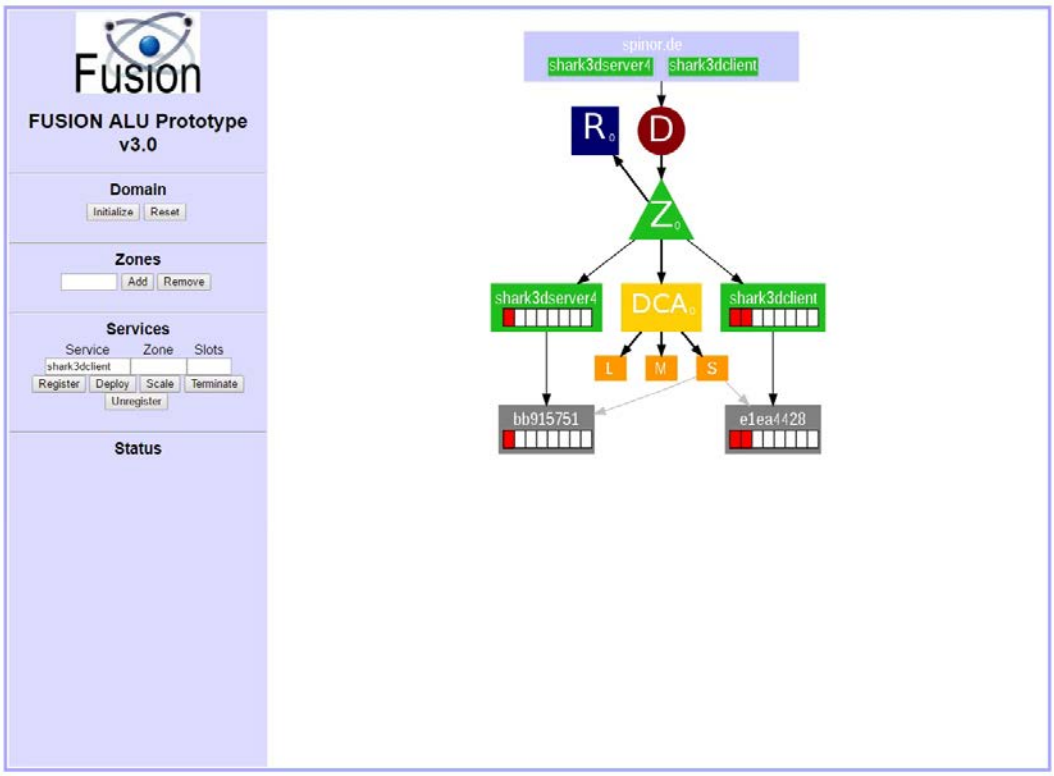


Figure 67 Usage of one simulation and two rendering session slots

Then the lobby software configures the service slot instances of the services by instantiating simulation states and rendering views, connecting them internally and establishing the required network connections between them. For this the lobby software communicates with the service instances via the SAP protocol (simple actor protocol) of the Shark 3D software, using the service endpoints returned by FUSION.

For example, in the following screenshot, the “produce_remote_listen” command together with the “create_listen” command are used to establish a listening socket for the inter-service-communication.

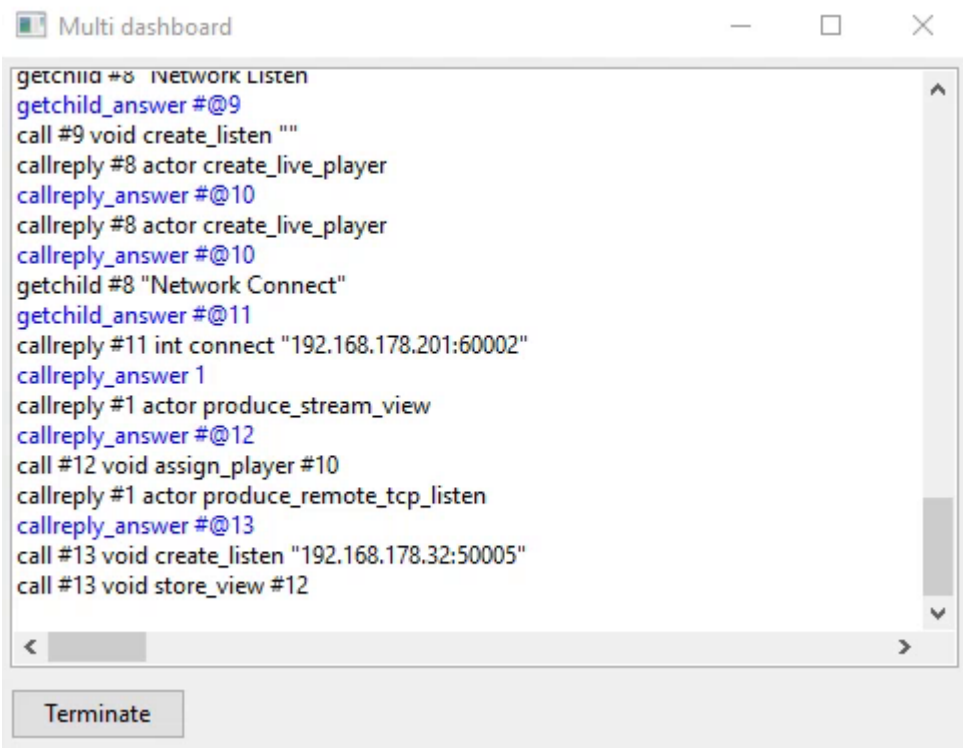


Figure 68 Lobby software creating and configuring the service endpoints

5.11 Multi-configuration services

5.11.1 Testing plan

In this test scenario, we evaluate two different mechanisms for configuring individual service components for adapting e.g. the frame rate, resolution, or what other service component to connect to, for example as part of a dynamic service graph. The first mechanism is the multi-configuration (a.k.a. the service aliasing) feature inherently supported by FUSION. This was described and evaluated in Deliverable D3.2. The second mechanism is leveraging the stateful application feature, where individual service components are configured via a coordination service as part of a stateful service session.

We will validate this with the EPG and streamer service components, and focus on the configurability of two types of service configuration parameters:

1) Service quality parameters, such as frame rate, resolution, encoding and streaming format and encoding quality;
2) Service composition parameters, such as which other service(s) to connect to.

5.11.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple Evaluator Service
- Video Decoder Service
- 2D EPG Rendering Service
- Stateless Multi-Configuration Streamer Service
- Stateful Streamer Service

- Thin Client

5.11.3 Results

In FUSION orchestration (as well as the FUSION prototype), we support the concept of multi-configuration services or *service aliasing*. This concept has been discussed in detail in Deliverable D3.2. Basically, this allows FUSION to reuse and reconfigure particular service (or resource) instances for multiple service types. Imagine for example a particular game service, or encoding service, then a service provider could provide different versions of the same service to clients, such as for example a basic standard-resolution version with best-effort QoE at a lower cost, as well as a premium high-resolution version with excellent QoE at a higher cost. Implementing this flexibility can be done in three ways:

1) Using completely separate service instances. Sufficient session slots for each variant need to be provided, resulting in a higher fragmentation overhead (see Deliverable D3.2).

2) Using one a single service type, and handing this diversity at application level. By doing this without support of FUSION orchestration and resolution however, two key disadvantages are lack of service-specific resolution and scaling, based on different policies.

3) Using the FUSION service aliasing feature, mapping session slots of different service types onto the same physical service or resource instances. This effectively combines the benefits of the previous approaches while mitigating their key disadvantages.

For this setup, we will provide two variants of the 2D EPG service, namely one rendering at 720p@25FPS and one rendering at 360p@100FPS, and implement them using the three options listed earlier. We will call the first option ‘stateless’, the second option ‘stateful’ and the third option ‘multiconf’ in all further graphs and evaluations. The runtime prototype states of each respective option are depicted in the following three figures.

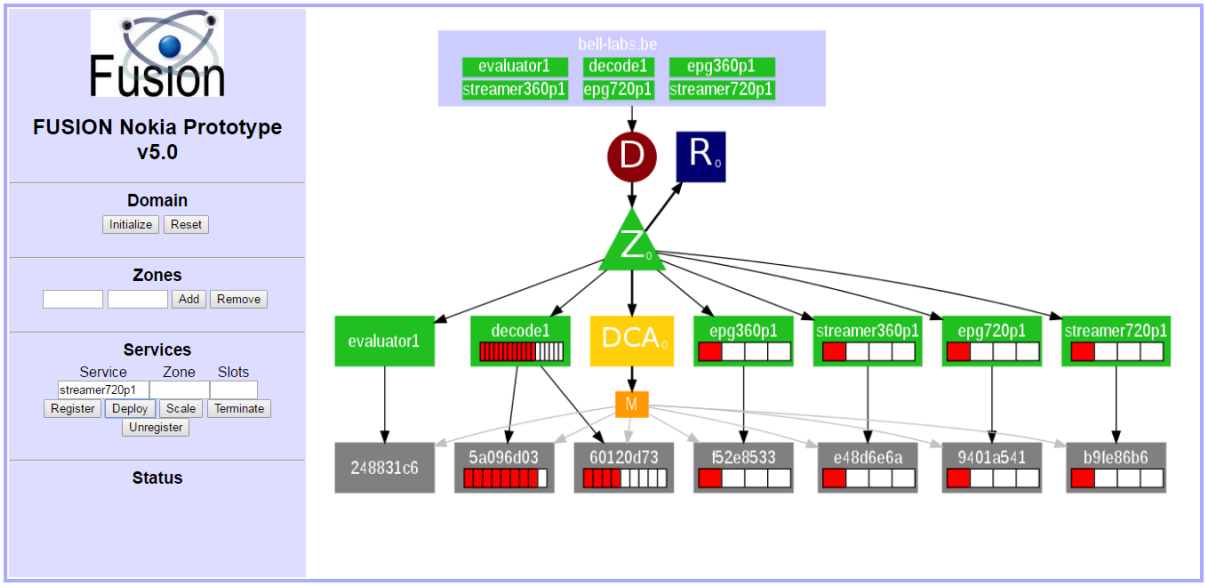


Figure 69. Stateless deployment version of the 360p and 720p variant of the same EPG service.

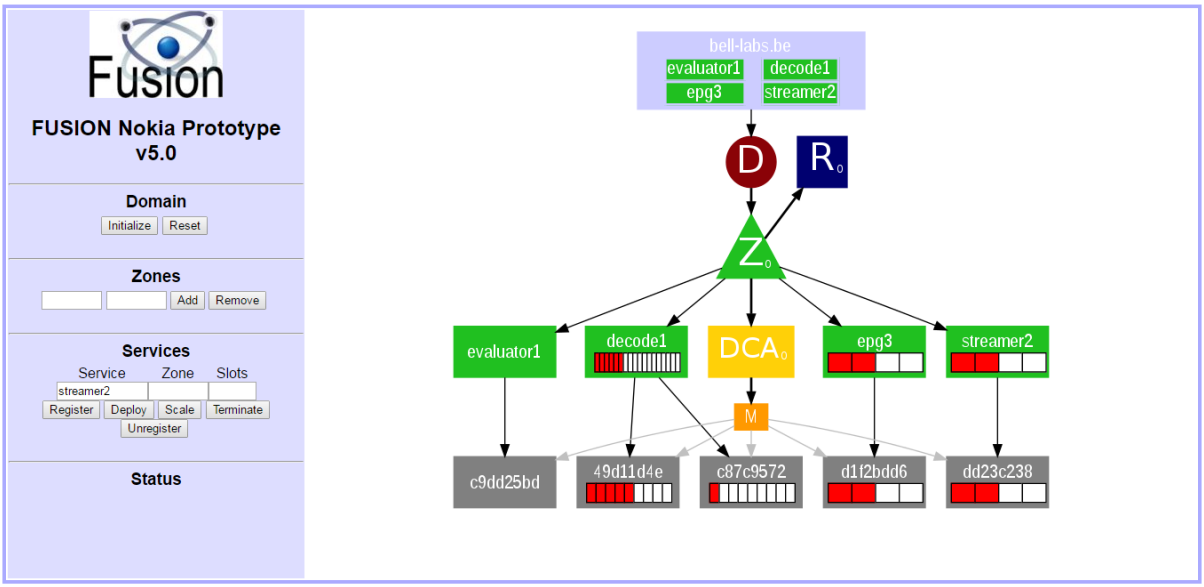


Figure 70. Stateful deployment version of the 360p and 720p variant of the same EPG service.

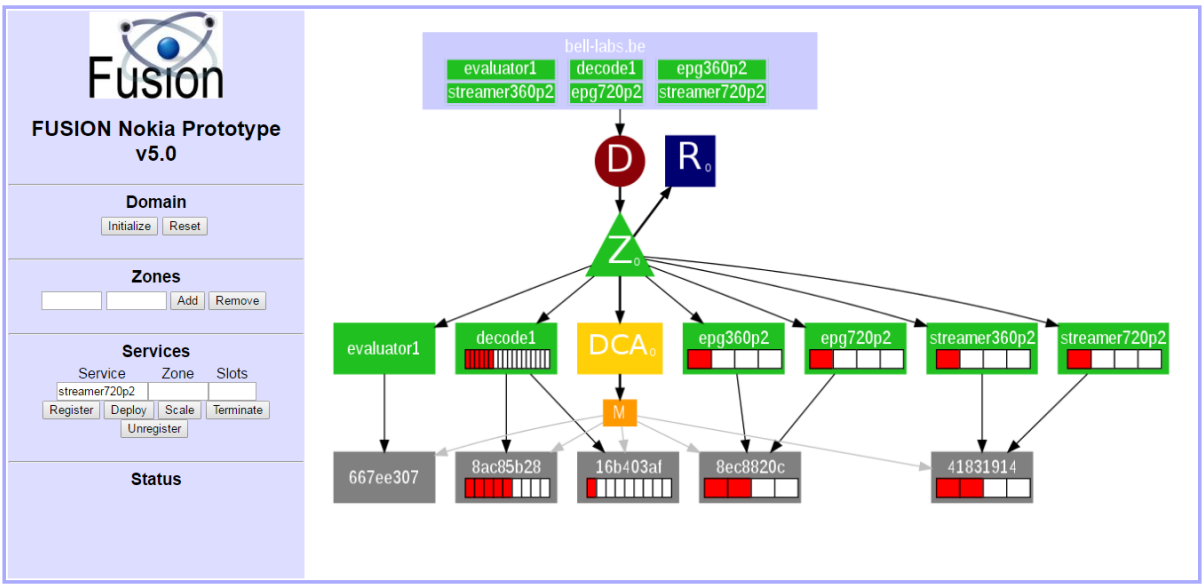


Figure 71. Multiconf deployment version of the 360p and 720p variant of the same EPG service.

The stateless option has most active service types and service instances; the stateful option has the least of both, whereas the multiconf is somewhere in between, combining deployment efficiency with FUSION-level orchestration and resolution flexibility.

Performance wise, the biggest difference is w.r.t. the reuse of decoded video streams. In the stateless variant, there is no reuse of decoded video streams across both EPG rendering service instances, though this intelligence could have been added into the decoder service. As a result, the CPU utilization, the memory bandwidth as well as the total amount of allocated memory, is significantly higher, as shown in the figures below.

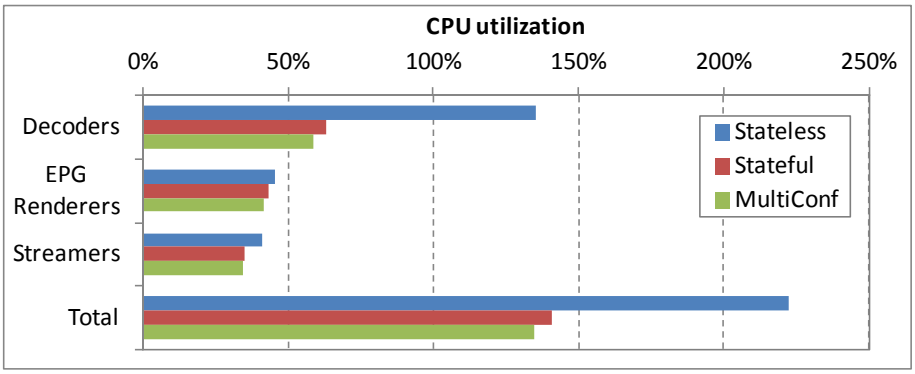


Figure 72. CPU utilization for the three deployment options

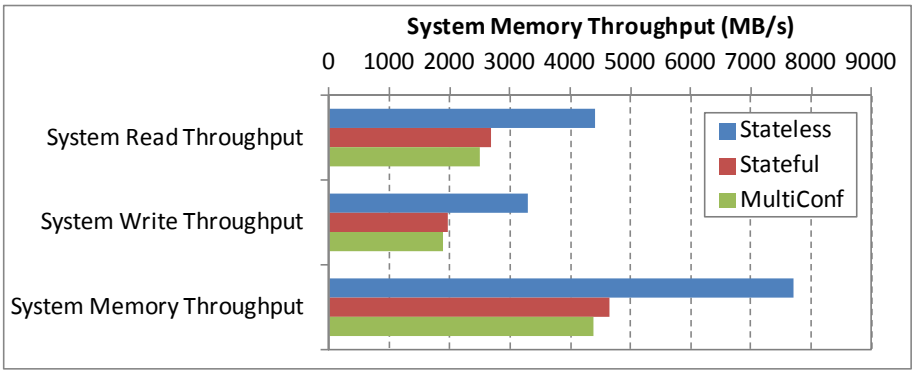


Figure 73. System Memory Throughput for the three deployment options

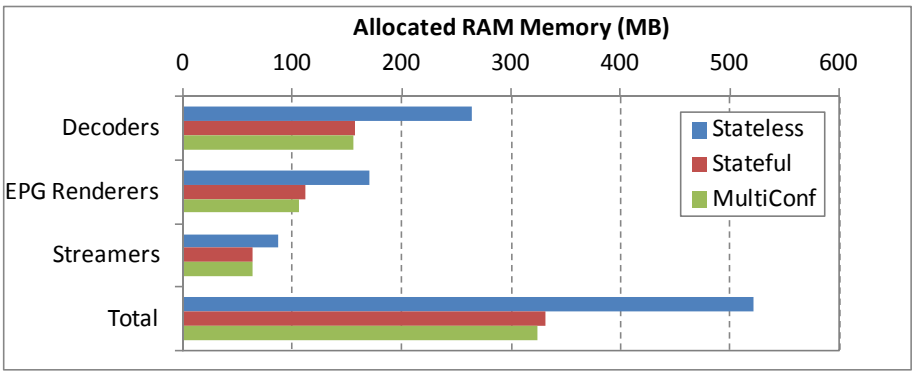


Figure 74. Total Allocated System Memory for the three deployment options

The application roundtrip latency on the other hand is almost identical, as expected in this symmetrical deployment, as show below.

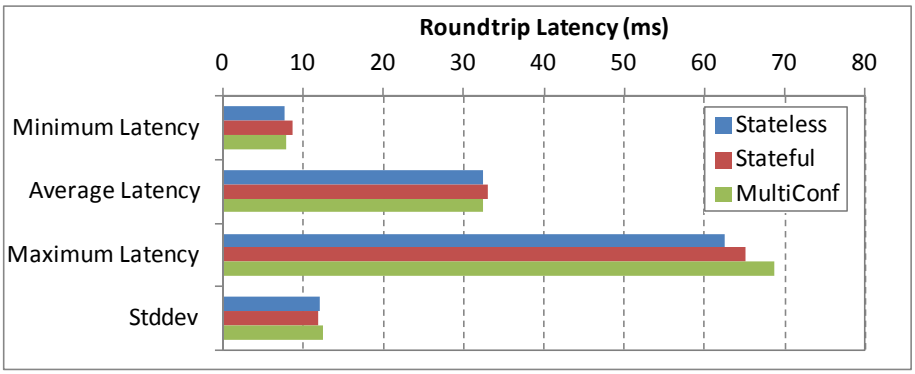


Figure 75. Application Roundtrip Latency for the three deployment options

Which deployment option to use for implementing this configurability depends on the application requirements. Having separate FUSION service types has the advantage of being able to associate different orchestration (e.g., scaling & placement) and resolution policies with each service type. For high-level key configuration differences, this amount of flexibility may be a necessity. Other configuration options on the other hand are less critical and could perhaps be dealt with at application level via stateful services. As shown in Deliverable D3.2 (using standard queuing theory), opting for the multi-configuration option over the truly independent/stateless option has the benefit of reduced fragmentation of available session slots across service instances, resulting in a higher efficiency.

5.12 Dynamic session slot availability updates

5.12.1 Testing plan

In this scenario, we evaluate the optional functionality integrated into our application service prototypes for monitoring resource availability and dynamically adapting session slot availability based on these measurements. There are two main reasons why the session slot availability may change over time:

- Internal variability: under- or over-consumption of runtime resources by active session slots: sessions can consume a varying amount of runtime resources over time; different sessions may also use different amounts of resources based on their respective usage patterns (e.g., passive vs active user).
- External variability: varying available runtime resources due to resource oversubscription and/or interference because of noisy neighbours.

We evaluate how our application service can deal with this variability in our prototype, and how the concept of session slots helps in summarizing true service-specific resource availability.

5.12.2 Involved components

- Orchestrator
- Greedy Resolver
- 2D EPG Service with DynSlots enabled
- Thin client

5.12.3 Results

A break-down of application resource utilization over time is depicted in the drawing below. Note that each runtime resource has a similar graph that may have to be taken into account, depending on the application resource bottleneck. For example, for memory-intensive applications, the available memory throughput may be crucial, whereas for other applications, monitoring memory bandwidth is irrelevant. Due to internal and/or external runtime variability, the application resource bottleneck may change over time.

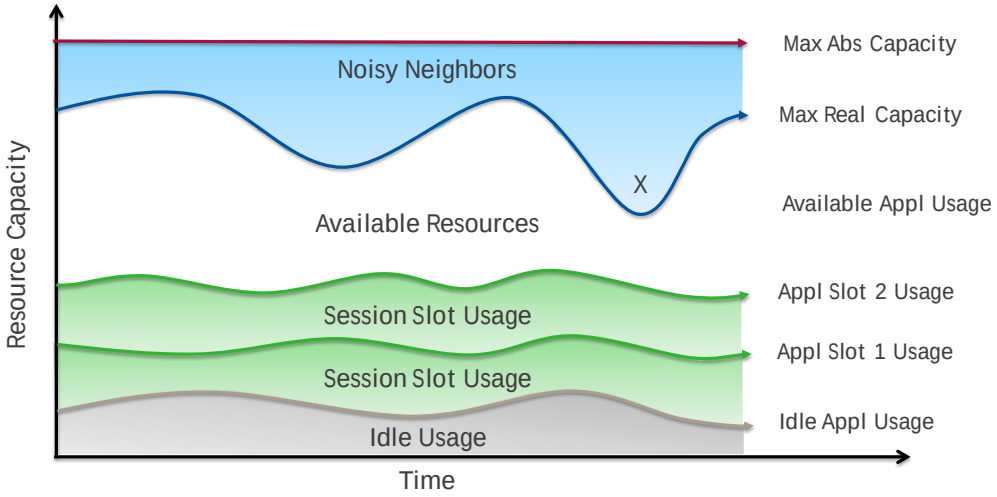


Figure 76. Break-down of application resource utilization over time.

When deploying an application on a particular system, the maximum capability will typically be limited by the amount of resources available on the target system and/or allocated to the application. Note that in a dynamic and/or heterogeneous environment, this maximum absolute capacity could change over time (not shown in graph). For example, a cloud platform could decide to allocate more (or less) CPU cores or CPU time to a particular application; similarly, memory, storage and network capacity allocated to a particular cloud application may change over time.

However, in the cloud, even when an application theoretically has a particular maximum resource capacity (e.g., 2 virtual cores), the actual real capacity typically will be lower due to a combination of oversubscription as well as loss of useful resources because of noisy neighbors (e.g., cache trashing). For typically cloud IT services, this may not be a huge deal, but for (near) real-time applications such as in FUSION, with long-lived sessions and a particular QoE requirements, this can be a serious issue. Specifically peaks like the one shown in the figure around 'X', can be quire harmful, and should be dealt with. This section allows to deal with these from an application perspective by dynamically altering session slot availability (e.g., in a conservative manner), whereas in Section 5.18, we handle this issue from a heterogeneous cloud perspective.

Note that for time-critical applications, resource latencies typically are equally if not more important that available capacity. For example, an application not being scheduled for several tens of milliseconds, or a disk read taking much longer than expected can easily result in multiple deadline misses. As such, this should also be taken into account while estimating session slot availability.

As our EPG prototype application is mainly CPU bound, we mainly focussed on measuring effective CPU availability as well as rendering delay for estimating session slot availability. For this, we created an application-external "DynSlots" script running next to the main EPG application service in the same service container, for handling this complexity. The script currently monitors a number of things:

- The maximum CPU capacity, as far as it can detect this, by querying various cgroups settings (i.e., /cpuset/cpuset.*, /cpu/cfs*, etc.).
- The idle application CPU usage, measured when 0 session slots are available, by querying both cgroup metrics (/cpuacct/*) as well as proc settings (/proc/*/stat, etc.).
- The per-slot application CPU usage, by measuring the average utilization per session slot. Note that due to self-interference, the average per-slot utilization may increase as the number of active session slots increase. On the other hand, better resource sharing can also result in a decrease of per-session slot utilization.

- Number of deadline misses. The script also triggers the Vampire-specific application property interface for obtaining the number of deadline misses since the last measurement. If this number surpasses some threshold (accumulated or per period), the session slot availability is adjusted accordingly.
- Available CPU utilization. The amount of available CPU resources can be measured and/or estimated in a passive or active mode. In the active mode, we run a small side-application next to the main application service. We run this application using the SCHED_IDLE scheduling policy (chrt -i 0) to avoid interference. Note that in stable environments, this background idle service could be activated in a sampled fashion, and perhaps could be combined with other mechanisms to estimate the actual remaining CPU resources (e.g. by also incorporating the average rendering delay).

In this paragraph, we evaluate an initial implementation of this script. Note that this script is written to be useful for a wide range of services, though it does require some interaction with the application to be fully functional and effective. Note also that this is only an initial version; more robustness, as well as capabilities such as learning from previous or other executions, could be included as well.

The initial state of this demo scenario is shown below, and consists of our usual EPG composite service graph, with the main difference that in this case, we statically enabled the DynSlots scripts in the manifest (hence the different registered service name).

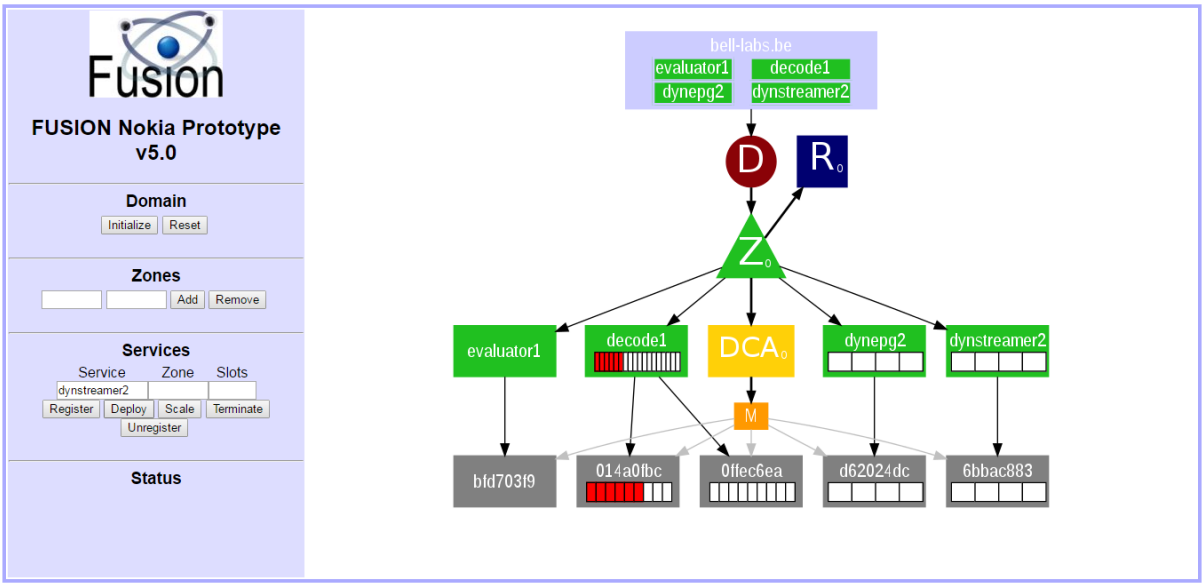


Figure 77. Initial State for evaluating our DynSlots script.

In the background, while there are 0 slots in use for the dynepg2 service, it measures the idle background CPU usage, using some weighted average for stabilizing the results while still keeping them up-to-date. For this service component, the idle CPU utilization is roughly 1-2% CPU on our test platform.

Next, we connect one client to the EPG composite service. As a result, the DynSlots script now also starts collecting statistics on the per-slot CPU utilization (subtracting the 0-slot idle CPU utilization), which is about 13%. Combined with the measurements of the remaining available CPU resources, the DynSlots scripts estimates about 6 slots can be supported in total (1+5), and reconfigures the EPG service component via the Vampire properties API to allow for up to 6 session slots. This subsequently is also reported to the FUSION zone manager, as is shown below.

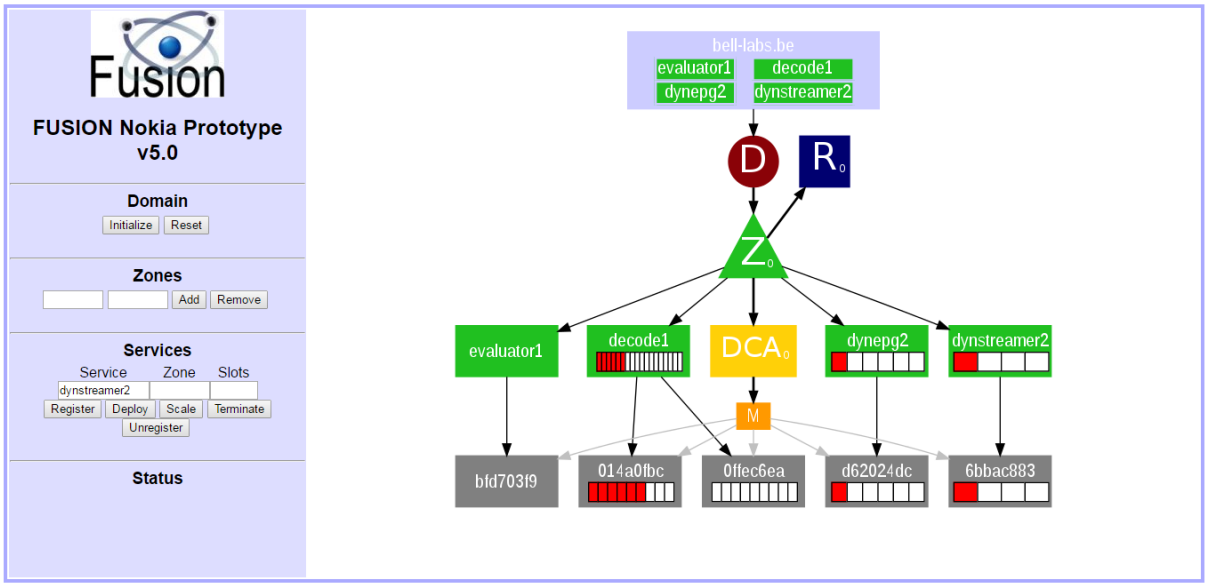


Figure 78. Intermediate state after a client has connected: the DynSlots scripts updated the total session slots of the dynepg2 service instance from 4 to 6 slots based on the measurements.

Adding two more clients does not change the estimated number of session slots. Next, we introduce some noisy neighbour behaviour by starting some background application on the host. This causes the effective available CPU resources to drop, which is detected by the DynSlots script: by gradually increasing the load, the announced number of session slots drops from 6 to 5 to 4 and finally 3 when we try to saturate the host, as depicted below. In case of such high background load, the application could even decide to gracefully shut down one or more active sessions to maintain QoE, and remember for future deployments that this environment provides relatively unreliable performance.

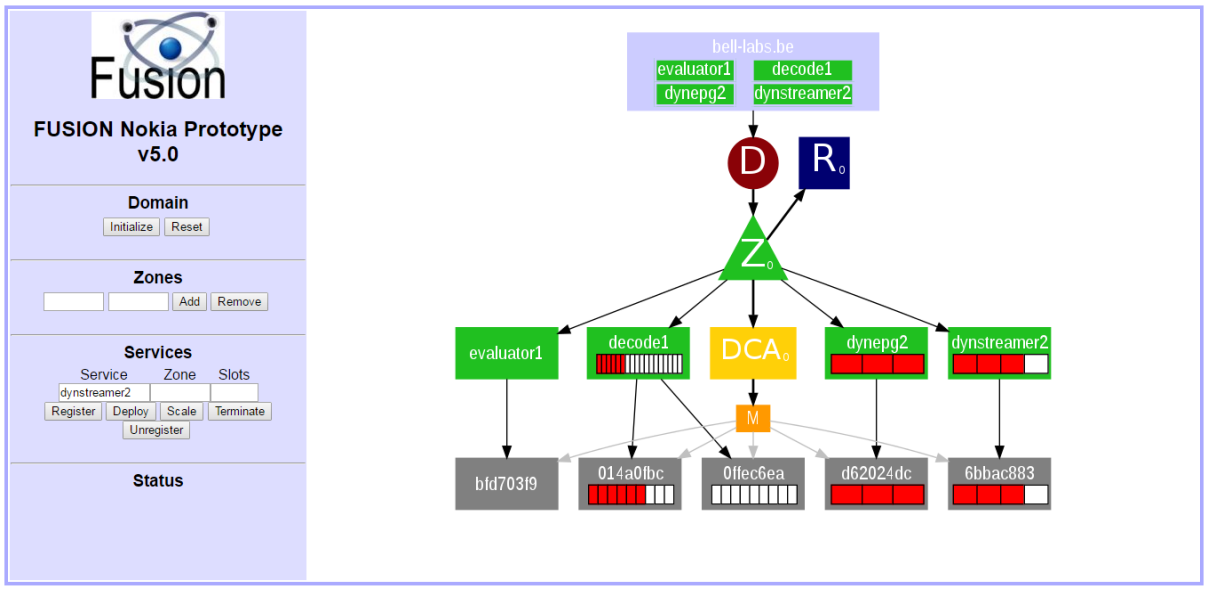


Figure 79. Updated state after three clients have connected and a high background load is introduced: the DynSlots script reduces session slots availability to from 6 to finally 3.

When we remove the background load, the DynSlots script aggressively recovers, announcing 6 session slots again. Obviously, how aggressive or conservative the DynScript should reduce or increase the session slot availability depends on the application behaviour and requirements, as well as the noisy neighbour pattern. In general, one could expect the DynSlots script to quickly reduce session slot availability but only to increase session slot availability after a longer period of stability, the duration of which may depend on the average session duration.

5.13 Resolution policies optimizing costs versus QoS

This scenario is based on the setups described in 5.1 and 5.10.

5.13.1 Testing plan

Based on the deployment described in section 5.1, we request services based on two different resolution policies. We assume the following cost profile:

Session slots	Virtual Wall zone	Spinor zone
Simulation service	Cheap	Expensive
Rendering service	Cheap	Equally cheap

Table 4 Cost profile of services deployed on the virtual wall and Spinor zones

Furthermore, in this specific scenario the user is located in the Spinor network, and therefore is very close to the Spinor zone, while the Virtual Wall zone is farther.

A user close to the Spinor zone requests a dual-user scenario with two different resolution policies:

- Prefer better QoS over lower costs. This should allocate both the rendering service and the simulation services from the closer Spinor zone.
- Prefer lower costs over better QoS. This should allocate the rendering service from the closer Spinor zone, but the simulation services from the cheaper Virtual Wall zone.

5.13.2 Testing and measuring procedure

The measurements are conducted in the following way, based on the deployment described in section 5.1. This approach is used not only for this scenario, but for all following scenarios involving the Shark 3D based services.

1. The users are requesting a dual-user dashboard scenario from the lobby software.
2. This triggers the lobby software requesting two rendering services and one simulation service from FUSION. It then configures the session slot instances for the specific scenario. For details about the mechanism see sections 3.6 and 3.8.
3. Each user is connecting each one client to the two rendering service slots.
4. On one thin client we control multiple elements in the dashboard and game scene. The elements are, depending on the specific situation, a physically simulated vehicle, a character and the 3D camera.
5. Additionally, we replay a pre-recorded set of actions directly on the renderer service.
6. The input commands of these actions are transmitted to the rendering service belonging to this user, where they are evaluated.
7. The renderer service renders new output and transmits it to the thin client.
8. The renderer service also transmits the actions in the scene to the simulation service, which evaluates them, too.
9. The simulation service updates the actions to the other rendering service.
10. The other rendering service renders new output and transmits it to the other thin client.

5.13.3 Emulating users and automatic measurements

Using manual control inputs means that the input is more or less different for each experiment run, resulting in different behaviour of the simulation and rendering, and causing different network traffic, and therefore affecting the QoS. Furthermore, human estimations of the QoS are less precise than measurements.

This is the reason why we decided to not only use human input and subjective human evaluation, but focus mainly on automatically evaluating the QoS by emulating users via automatically playing pre-recorded actions and using automated measurements for evaluating the QoS. Using pre-recorded actions ensures that the actions happening in the scene are always exactly the same and the results can be measured precisely and in a reproducible way. This makes it possible to compare the results for different runs with different configurations directly.

The pre-recorded actions emulating users aren't simply simulated user input like "moving forward", because non-deterministic behaviour of the simulation may result in different effects in the scene. Instead the pre-recorded actions contain the desired action results like movements on a particular path, which are converted each simulation tick into input commands like "moving forward" based on the current simulation state, for example the current vehicle position. The Shark 3D software already contained such a recording and replaying mechanism, which we used and modified as needed for the FUSION measurements.

The pre-recorded actions are executed directly on the rendering service, not on the thin client, since that system requires live access to the state of the scene, for example the current vehicle position, while the thin client does not have that information but only the rendered video stream.

5.13.4 Quantities to be measured

Internally we log all sent and arrival times of each single data packet in a brute force way. Note that we don't measure the send and arrival times on network level, but on Shark 3D application level. Additionally, we use various other logging information offered by the Shark 3D software out of the box. This logging makes it possible to run a more extensive analysis afterwards.

The focus of the specific evaluation used here based on the Shark 3D based services is the delay experienced by a second user in a multi-user scenario. This covers use-cases like multi-player gaming, collaborative 3D applications like distributed animation productions, interactive product presentations or shared media experiences with multi-user dashboard. Therefore, we evaluate the additional delay the second user experiences relative to the first user depending on different resolution results of FUSION. Specifically, the measured response times include the following elements:

- Network transfer between the rendering service and the simulation service. This value strongly depends on the locations of the service instances and therefore is significant to evaluate the FUSION results.
- Processing time on the simulation server. This time depends on the specific service application. For example, a complex game may require more time. Nevertheless, this time is usually also negligible.
- Network transfer between the simulation service and the other rendering service. This value is also significant to evaluate FUSION results.

Note that we don't need to include the final rendering time and the network delay for transmitting the rendered video stream to the thin clients, since this is the same on both rendering services and therefore no additional delay for the second user. Furthermore, as discussed in section 5.1, the involved test beds don't provide multiple locations of nodes having NVENC enabled GPU hardware, which is required by the rendering services, so that measuring such values is not useful anyway.

5.13.5 Involved components

- Orchestrator
- Resolver
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.13.6 Results

5.13.6.1 Policy preferring better QoS over cost minimization

We first chose the policy for optimizing QoS over costs minimization. FUSION resolves to a simulation service which is in the same zone as the rendering services.

The following diagram are the total delay between the two renderer services, which are synchronized via the simulation services, for the period where the recording was executed.

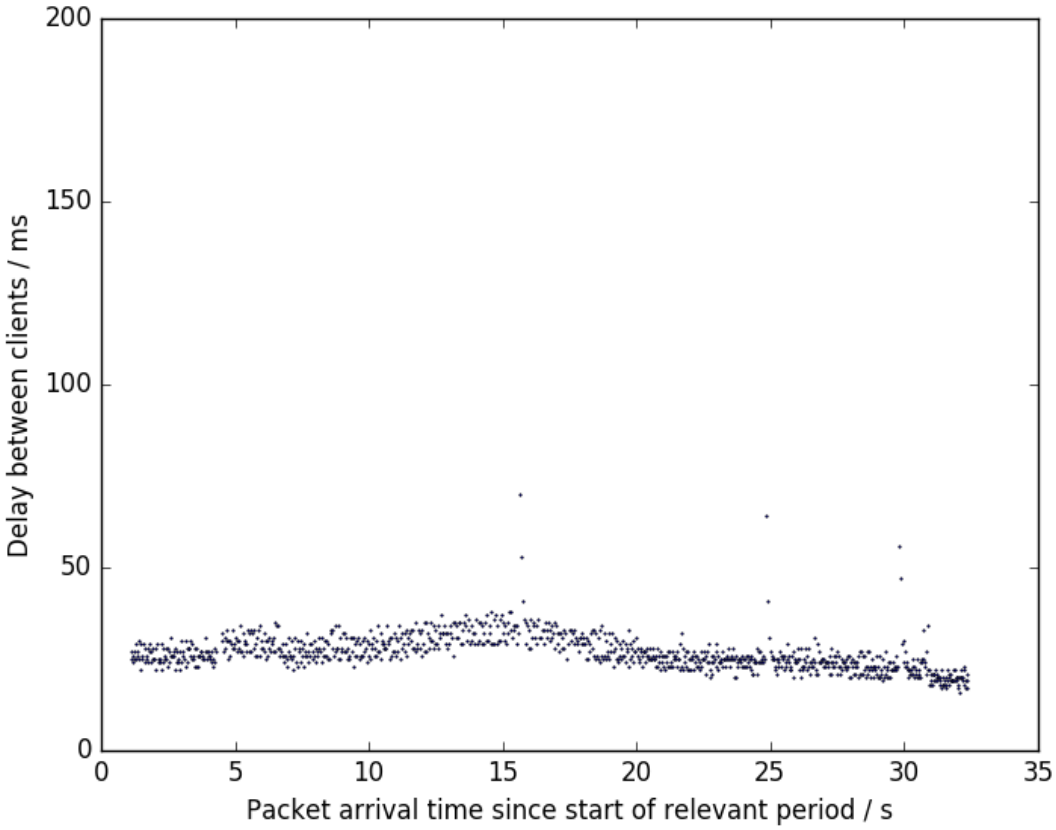


Figure 80 Delays between clients with policy optimizing for quality

The delay includes the network delay from transferring the input from one rendering service to the simulation service plus the simulation time of the simulation service plus the network delay from the simulation service to the other rendering service.

By disabling Nagle's algorithm, we avoid unnecessary additional delays caused by that algorithm, improving quality at the costs of slightly more network traffic, which is the standard approach for time-sensitive interactive applications like games.

The diagram shows that different sections of the pre-recorded sequence have different delays. For example, the delay has two maximum values around time 5s and around time 15s, and is decreasing at the end around 30s and afterwards. This is caused by different situations in the recording, causing different amounts of data which has to be synchronized via network depending on moving the vehicle, the avatar and the camera. The diagram shows that this change in the delay caused by different actions happening in the scene is significant, since it has roughly the same order of magnitude as the jitter of the delay itself. Depending on the situation the delay in different situations may be much larger, for example when the number of object movements which have to be transferred, varies greatly.

The following diagram shows the same delay, but excluding the simulation time on the server. Therefore, it contains the delay caused by the network, plus the time until the application layer processes the network packets, but not including the simulation time.

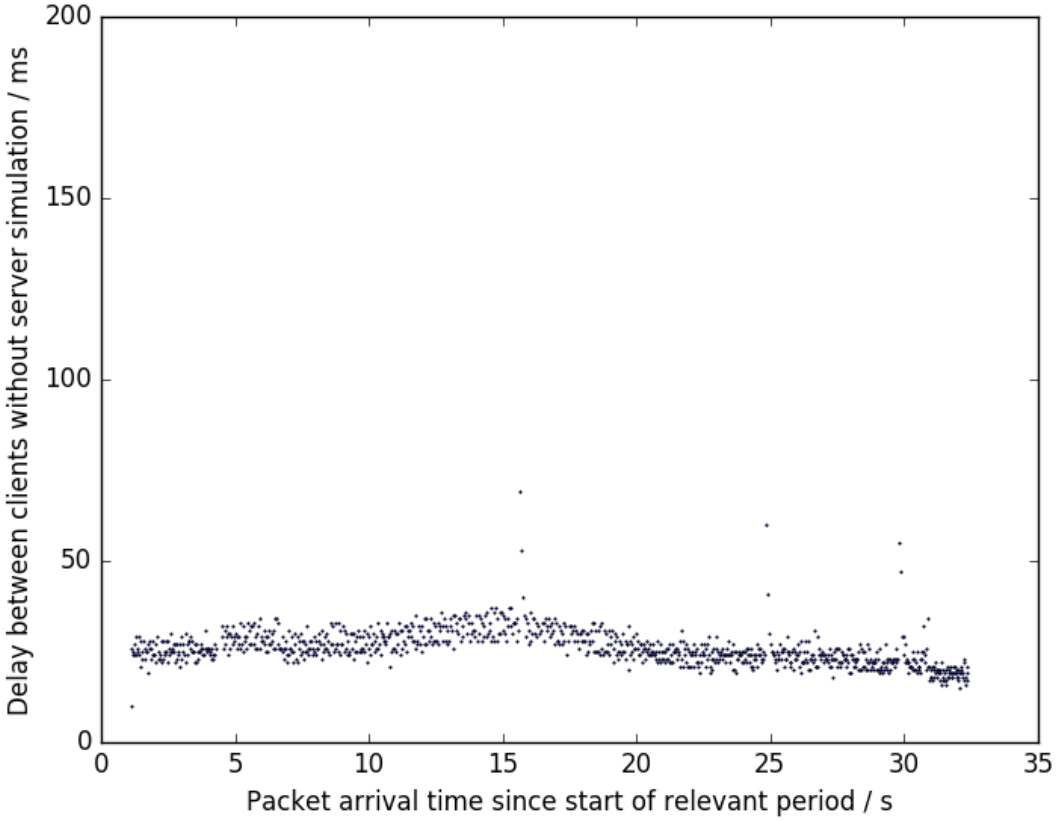


Figure 81 Network-only delays between clients with policy optimizing for quality

This diagram shows that the actual simulation time is negligible. Therefore, we use the full time in the following diagrams, which is closer to the QoE of the users.

Both diagrams also show some few outliners. Since the same outliners are also present in the measurements excluding the simulation work, they are caused by individually delayed packets. On the other hand, the application is running on a fairly steady framerate during replaying the recordings, hence we conclude that the reason for these individually delayed packets is not on the application layer. Furthermore, multiple of these packets having different delays are arriving approximately at the same time lead to the conclusion that the resubmission of a single lost packet causes the delay of such a group of packets. Since the Spinor network handles all company traffic and therefore the Spinor services have to share the network with all other corporate network traffic, such unpredictable packet losses are possible.

5.13.6.2 Policy preferring lower costs over quality

Then we change the policy to preferring lower costs over quality. In the specific setup the nodes running the simulation service are quite expensive. Therefore, FUSION resolves to the simulation server on the virtual wall instead of the Spinor zone, which is farther. The following diagram shows the measurement results. The pattern of the different situations during the recording is the same as for the other policy, but significantly higher delays.

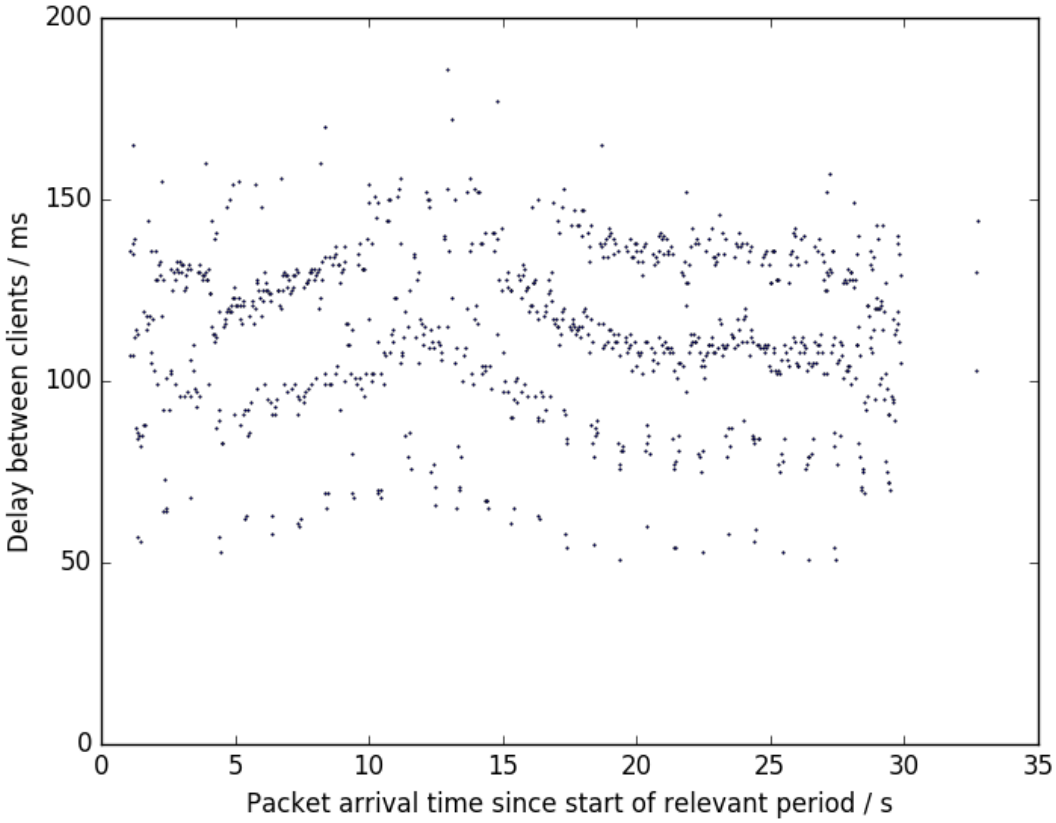


Figure 82 Delay between clients with policy optimizing for low costs

The additional delay is caused by the distance between the Spinor network (located in Munich) and the Virtual Wall testbed (located in Ghent).

5.14 Dynamic resolution adaption

5.14.1 Testing plan

This scenario is based on the setups described in 5.1, 5.10 and 5.13, but enhanced by an additional dynamic aspect based on monitoring. A monitoring component continuously collects data about the network quality. The results are reported to an ALTO server. The resolver accesses the ALTO data for adapting the resolution tables.

The scenario consists of the following steps:

- Configure the resolver with the best quality policy.
- Request a dashboard service using standard network conditions. This should select the closest service instance on the Spinor zone.

- Add an artificial network delay of 800 ms to the simulation service instances (but not the rendering service instances) in the Spinor zone. This emulates some congestion in that part of the zone. This does not affect the connection of clients near the Spinor zone to the virtual wall. As result, also for clients close to the Spinor zone, the service instance in the virtual wall will have better quality now compared to the instance in the Spinor zone which has the artificial delay.
- Request a dashboard service again, now having bad response times.
- The measurement components update the new network delays to ALTO, used by the resolver to calculate new resolution tables. While this can be done periodically automatically, in this case we triggered updating the resolution tables by hand to have control over when that update happens.
- Request a dashboard service again. The request for the simulation service is now resolved to an instance on the virtual wall testbed.

5.14.2 Involved components

- Orchestrator
- Resolver
- Monitor
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.14.3 Results

The first diagram is the default configuration, where FUSION resolves to the simulation service in the Spinor zone, which is closest. Comparing this diagram with the diagram for the same situation above in section 5.13 exhibits the same pattern over time, showing that the recordings make the measurements reproducible.

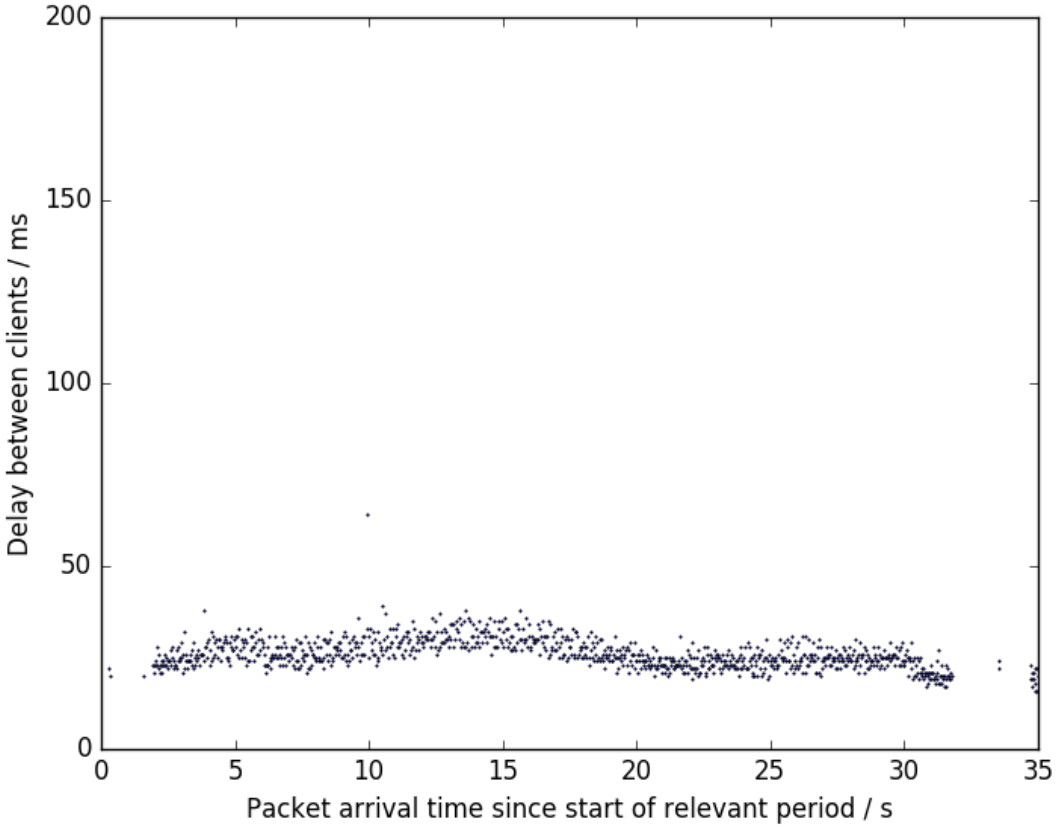


Figure 83 Delay between clients under good network conditions

After adding the network delay to the Spinor zone, the services have the following quality:

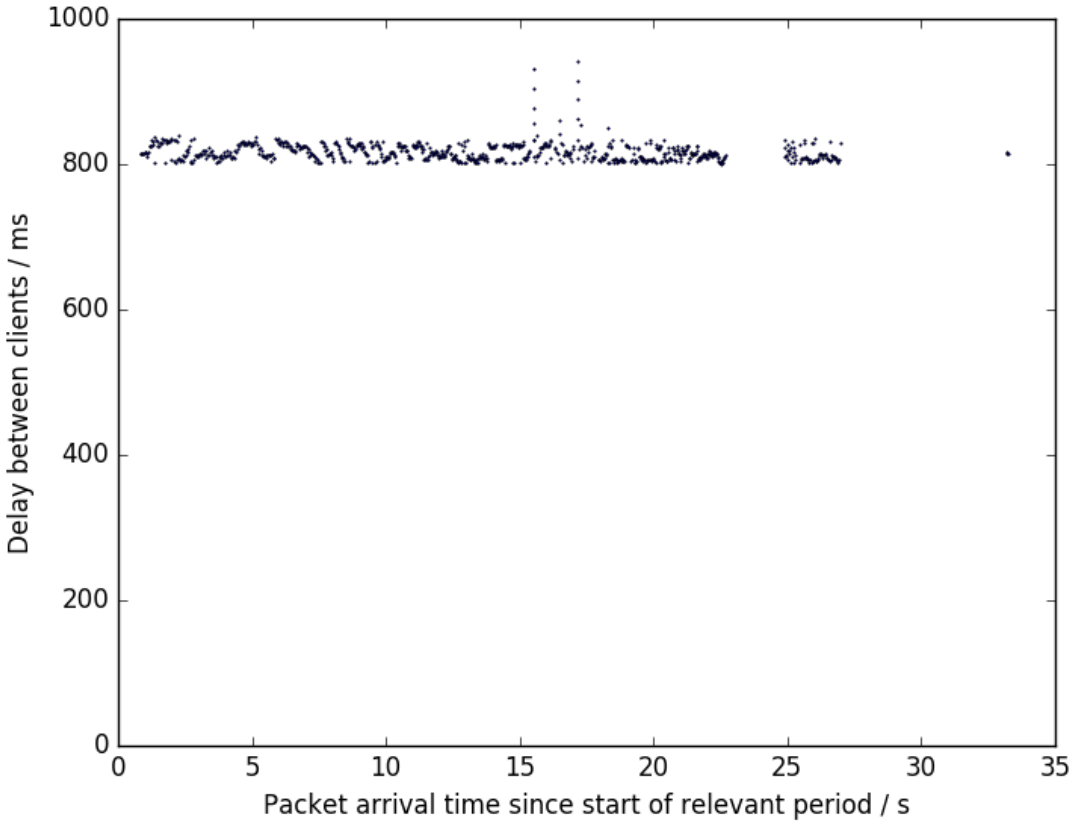


Figure 84 Delay between clients under bad networking conditions

Note the different scale of the y-axis. The additional delay of 800 ms is clearly visible.

After FUSION has detected the bad network quality, and adapts its routing tables, requesting the same service returns a simulation service instance on the virtual wall testbed now, having the following response times:

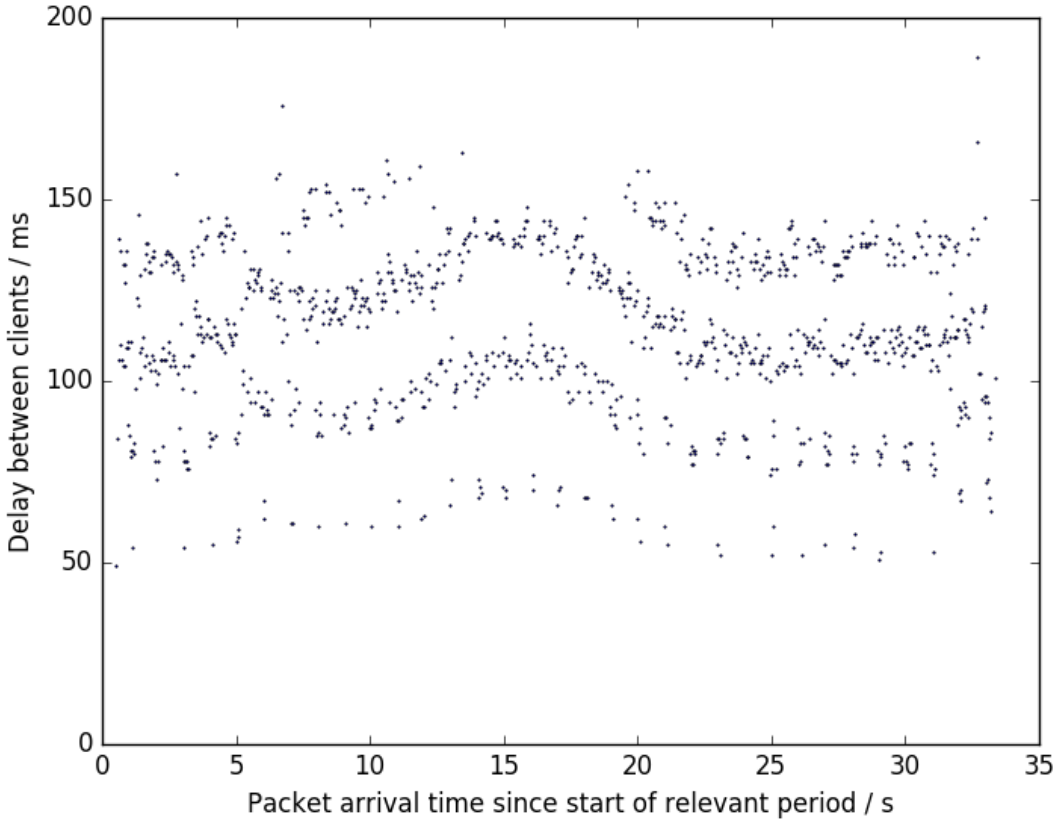


Figure 85 Delay between clients after resolution has been adapted

Due to the larger distance from the renderers, this is still worse than the original situation, but this is the best possible now due to the bad network performance on the closer Spinor testbed.

5.15 Resolution diversification for different user groups

This scenario is based on the setups described in 5.1, 5.10, 5.13, 5.14, but enhanced by using a second user group at a different location affecting the resolution.

5.15.1 Testing plan

A resolution request for a server takes into account where the user is located. There are two modes: Per default the resolver uses the IP the request is coming from. Alternatively, the request can contain an “on behalf” IP, which is then used by the resolver. This is necessary for example if a lobby software is sending the request of FUSION, so that FUSION should optimize the resolution not depending on the location of the lobby software instance (which may be a web server), but the location of the user since that is from where for example the thin client will connect to the service.

In our lobby software this is implemented by the “Client IP” parameter, see the screenshot of the lobby software in section 3.6.

5.15.2 Involved components

- Orchestrator
- Resolver
- Monitor
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.15.3 Results

The scenarios in sections 5.13 and 5.14 used one user group, resolving the rendering service always to session slots in the Spinor zone. Now we request services on behalf of a different user group having the client IP 192.168.178.26 (see close to the top left of the following screenshot). The test scenario is configured in that way that for this user group the best resolution is different. Then FUSION returns service slots from the virtual wall zone both for the simulation and rendering services:

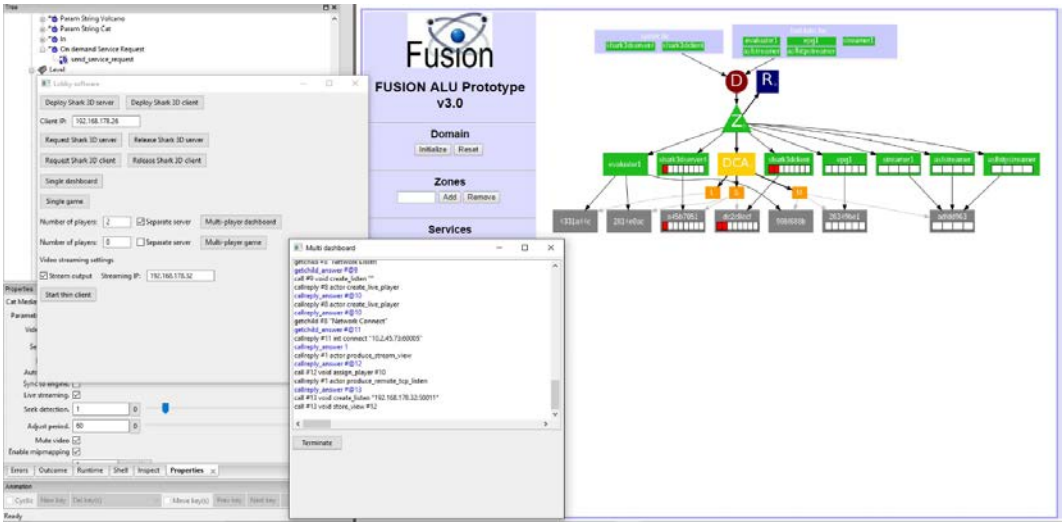


Figure 86 Session slot usage on Virtual Wall by a different user group

5.16 On-demand service deployment scenario

5.16.1 Testing plan

In this scenario, we evaluate the on-demand service deployment features of our FUSION prototype. In case the resolver receives a service resolution request for which it cannot find any appropriate session slot, we foresee in FUSION that the resolver can trigger either a registered domain orchestrator, or possibly even a zone manager directly. This on-demand scenario is especially important for efficiently handling long-tail services (for which predeploying instances everywhere is not feasible) or for handing flash crowds, with unexpected large demands.

Triggering a zone manager directly can significantly reduce the total startup delay compared with having to involve the domain orchestrator, especially in case the extended session slot version including the queuing time is used (see Deliverable D3.3), in which case the resolver has knowledge on the average delay in each zone before a new session slot can be made available.

5.16.2 Involved components

- Orchestrator

- Greedy Resolver
- Simple evaluator service
- Simple test service
- Fat 2D EPG service
- Thin client

5.16.3 Results

The initial state is depicted in the figure below. The *test1* and *epg4* services are only registered, and no autoscaling has been enabled in the manifest for this test.

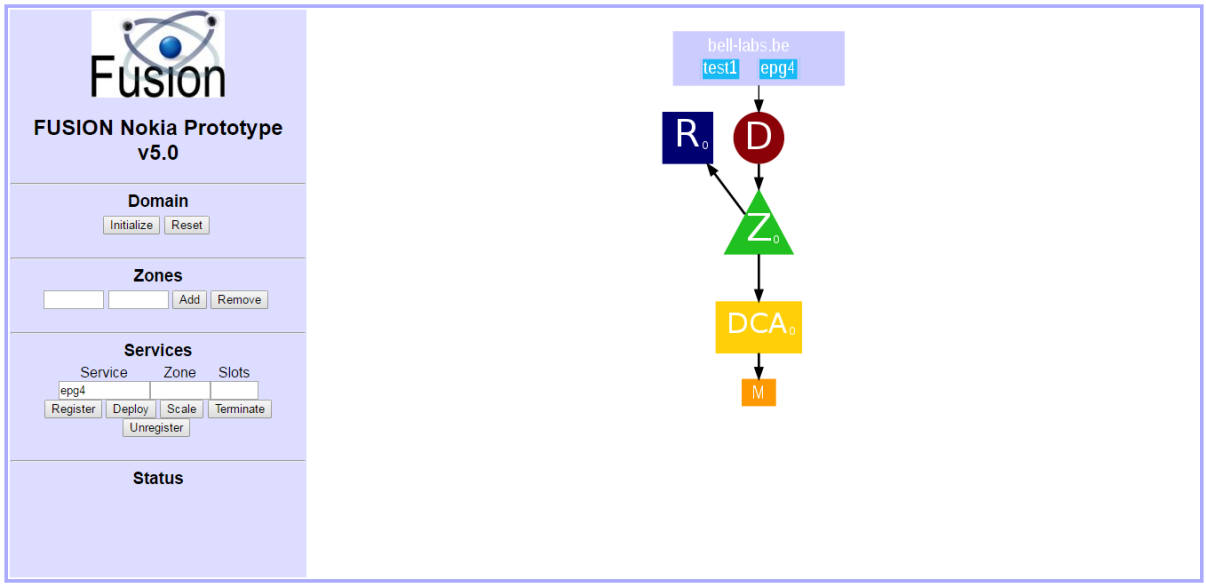


Figure 87. Initial setup for the on-demand test scenario

Next, we make a service resolution request to the FUSION resolver for service test1.bell-labs.be. As the resolver has no information (so definitely also no available slots) about this service, it triggers the registered FUSION domain orchestrator for deploying a new instance in a nearby execution zone of the client. This will trigger all domain-level and zone-level placement and deployment functions, possibly including evaluator services. At the end, a new instance is deployed, and session slots are announced. The resolver detects the new session slots (directly or indirectly), and returns the endpoint to the client.

For the test service, this entire process takes about 1.5s in total. Note that this includes the deployment time of the service as well as the time before session slots are announced by the service and propagated to the resolver. In this setup, the service container images are assumed to already be completely preprovisioned on the target execution environment. Any additional preprovisioning delay (of some or all container image layers) would have to be included if this is not the case.

Next, we fill all available session slots of this test service, and make an additional resolution request. This time, the resolver will directly trigger the execution zone that is reporting session slots for this service (but temporarily has none available). The selection of the zone in general could be done based on various criteria, including the reported queuing delay, ping latency, etc.

As this zone-level deployment trigger avoids domain-level placement, evaluator services, etc. and as such reduces the on-demand deployment time. For our test service, this on-demand deployment now takes only 0.5 seconds. As this service is already deployed in this zone, it can be expected that the provisioning delay will typically be minimal. The final state is shown below.

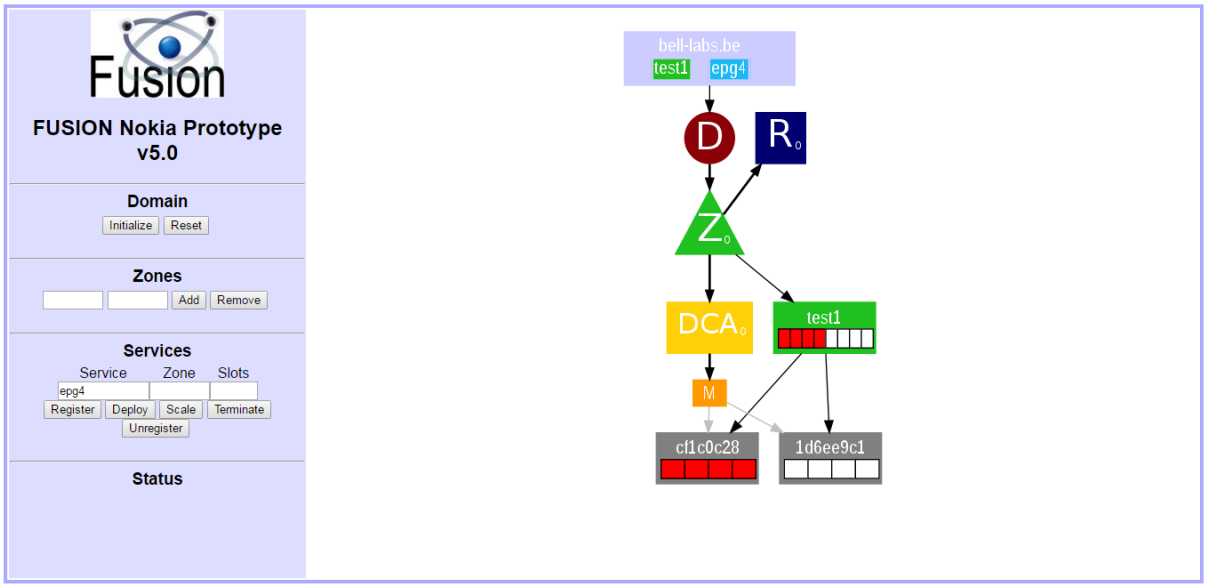


Figure 88. Setup after the resolver triggered a domain-level and zone-level on-demand deployment of the test1 service.

Next, we register and deploy the simple evaluator service, and make a FUSION service resolution request for service epg4.bell-labs.be. This again will cause the resolver to trigger the domain orchestrator, which in its turn will also trigger the evaluator1.bell-labs.be evaluator service, and finally the deployment of the service as a container and the announcement of session slots. For this service, the end-to-end latency is about 2.15 seconds, again assuming the (fat) service was completely preprovisioned on the target execution node. Obviously, in the on-demand deployment scenario, the placement algorithm should ideally take into account the on-demand provisioning latency.

Next, we again consume all session slots by connecting test clients, and subsequently make another service resolution request. This will now trigger again the resolver to directly request a selected zone manager to deploy a new instance, bypassing some of the domain-level deployment functions. For this service, the on-demand deployment now takes about 1.2 seconds. This is more than the 0.5 seconds of the test service, as the EPG service needs more time to initialize and configure itself before it can announce session slots. Services that can be deployed on-demand could however accelerate the announcement of session slot availability to minimize this delay. The zone manager itself could also proactively announce a conservative amount of session slots (e.g. 1 session slot) to the resolver while waiting for the effective number from the service. We did not implement such accelerations though. The final state is depicted below.

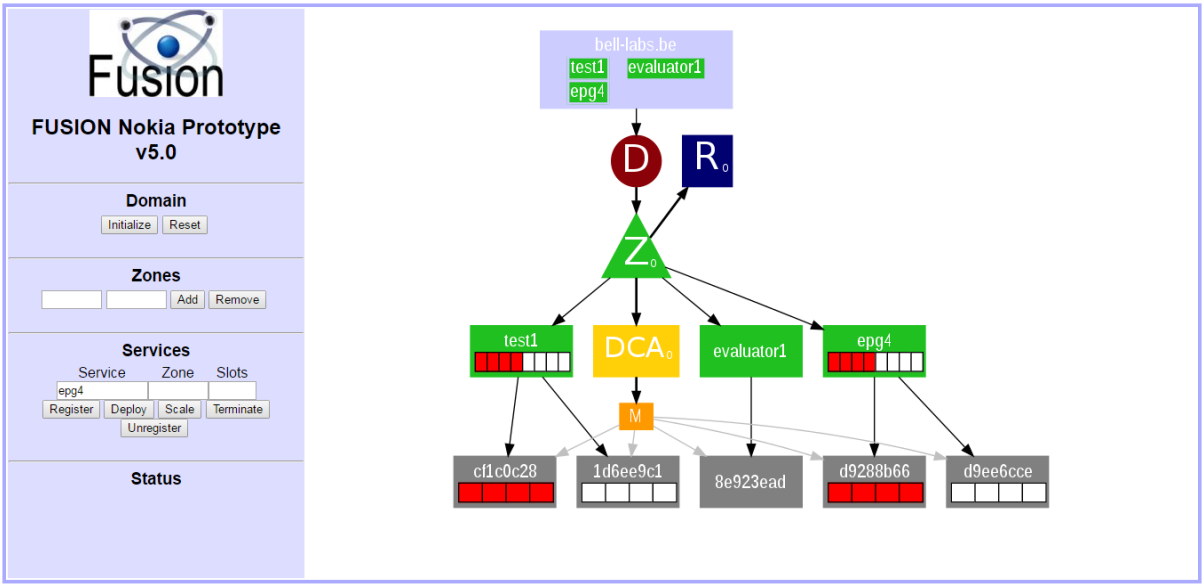


Figure 89. Final setup after the resolver triggered a domain-level and zone-level on-demand deployment of the epg4 service.

5.17 Dashboard end-user test

As described in D6.3, Spinor hosted an interest group meeting, where we presented the FUSION architecture and provided a live demo involving the Virtual Wall and Spinor testbeds and showing the thin client game and dashboard use-cases in the same configuration as used for the evaluation described in chapter 5 and mainly sections 5.1, 5.4, 5.5, 5.6, 5.10 and 5.13, based on the prototypes described in sections 3.3, 3.4, 3.6 and 3.8 and the integration described in chapter 4. The meeting included discussion about various technical details and different use-cases like training and support applications.

Based on this meeting we had off-line follow-up talks and individual meetings. As one result, we cooperated with Dexperio GmbH, Munich, for conducting additional tests, using the thin client at their premises.

5.17.1 Testing plan

The company Dexperio GmbH tested the dashboard service running in the Spinor network from a computer in the Dexperio office.

Both the service and the end-user computer were connected via the M-Net provider, optimized mainly for down-stream. So the video down-streamed on Dexperio side had to be up-streamed on Spinor side via a connection not optimized for it.

5.17.2 Results

The ping between the two computers were jittering between around 27 and 40 ms, showing a significant delay and a quite high jitter.

We then let Dexperio connect from their location to the rendering services and provided them questions about the subjective quality of the thin-client based dashboard service:

Question	Feedback
What was the subjective quality of the video streaming of the virtual scene with the static	All very good

scene, static camera but moving vehicle, and a moving camera?	
What was the subjective reaction time when controlling the camera and moving the vehicle?	Quite large lag
What was the subjective quality of the live webcam video integration?	Good

In summary, the feedback about the video streaming quality is very good. But the network lag was clearly visible. As the ping times discussed above show, the network connection quality was not the best, which shows up in the quite large lag. The reason is manly that both the testbed and the thin client were connected via standard SME internet connections which have a bad upstream. The quality was in particular significant worse than when having the thin client deployed in the same network as the services. This supports the need for deploying service instances as close to the user as possible.

5.18 Heterogeneous cloud platform

5.18.1 Testing plan

In this scenario, we validate the prototype in enabling heterogeneous clouds through different software runtime environments as well as support different hardware platforms. We already evaluated the notion of heterogeneity from an application service perspective via the evaluator services scenarios (see Sections 5.2, 5.3 and 5.4). In this section, we validate our FUSION prototype and testbeds from a cloud management perspective.

In a full autonomous heterogeneous cloud platform, the lower-level data centre layers could learn about the behaviour and QoE when applications are being deployed together on the same physical infrastructure, and minimize interference (e.g. through providing resource guarantees as well as avoiding not-easily controller interference patterns) as well as maximize QoE. In both cases, receiving addition QoE metrics from the application service instance beyond session slots is key.

5.18.2 Involved components

- Orchestrator
- Greedy Resolver
- Simple evaluator service
- Video Decoder service with application metrics logging enabled
- 2D EPG service with application metrics logging enabled
- Video Streamer service with application metrics logging enabled
- Application metrics logging server
- Thin client
- Metrics UI Dashboard

5.18.3 Results

In this scenario, we will evaluate how the video decode and EPG rendering service components behave on three different software/hardware environments on a heavily loaded system. The first two runtime environments share the same physical *Xeonv2* typical cloud node described in Section 5.2.3.1, whereas the third environment is deployed onto the *Avoton* micro-server cloud environment, which is tuned

for energy efficiency compared to high performance. For the first runtime environment, we deploy the services as regular best-effort services, whereas in the latter two environments, we enable the resource guarantees features studied in Deliverable D3.2.

The final state is depicted in the figure below, where we removed the evaluator services as well as the control panel on the left to increase clarity.

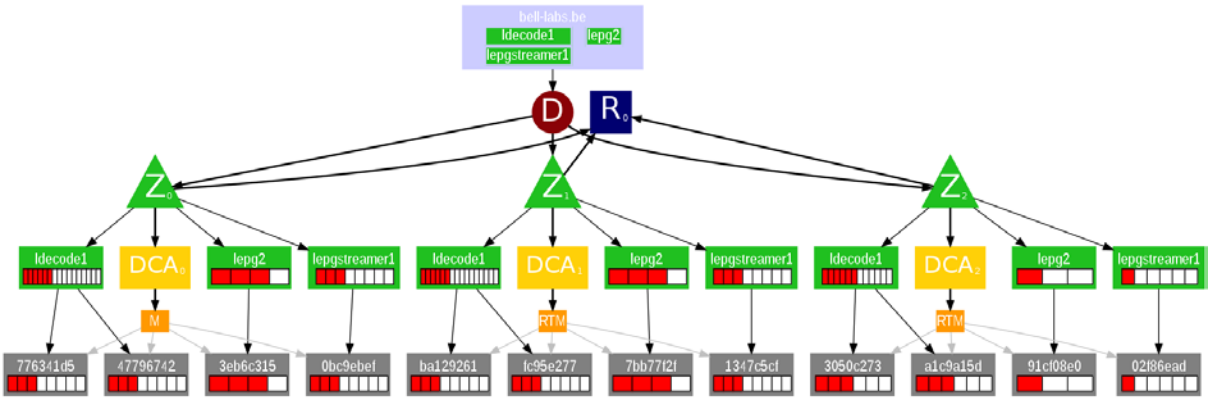


Figure 90 Heterogeneous cloud platform scenario state.

In this setup, we basically have three zones/DCAs, each with a single specific DCA environment type. In each zone, we deployed two *decoder* service instances, along with one *epg* and *streamer* service instances. Three clients each are connected to the EPG composite services deployed in the first two zones, and a single client is connected to the third zone.

Each of the service components have been configured to send essential application-specific QoE metrics once every second to a common logging service. Using a monitoring dashboard, we can visualize the actual runtime behaviour of each application instances. In the following figure, we first demonstrate the application QoE metrics for one decode service instance session from each zone. Note that we did not enable the DynSlots feature and used a simple evaluator service for this experiment.

As can be observed, the first decode session is showing signs of saturation and bad QoE, as the average target frame rate is not achieved anymore, with large average frame decode latencies, large jitter as well as an increasing number of deadline misses. This is caused by the interference of background interference. In the second case in the middle, although exactly the same application is deployed onto the same physical environment, but treated as a real-time application. Clearly, the results are much better and stable, Even though the load on the system is the same. In the final third environment, we used a micro-server environment, but applying also the same real-time guarantees features. The results are similar to the previous case, though with slightly worse predictability due to the limitations of the hardware environment. Note that this type of CPU is 5 times more energy efficient that the xeonv2, though less powerful. Depending on the application, this can result also in a significantly cost efficiency gain.

Figure 91 Application QoE metrics for one *decode* service instance session deployed in each zone runtime environment.

Apart from the decode service instance, we also studied the behaviour of the epg rendering service component. A screenshot of the live monitoring dashboard is presented below. A similar conclusion can be made as before, with a key difference that the EPG rendering service instance is a bit more heavyweight on the microserver, and as such in its current software implementation is probably less suited to be deployed cost-efficiently on this microserver, specifically when comparing session slot capacity. On the other hand, the real-time guarantees do still provide a significant advantage to improve density for the real-time application services.

A heterogeneous cloud platform on the one hand, as well as combined with proper evaluator services, it should be possible to be able to automatically manage and optimize deployment of a wide variety of unknown applications onto a wide variety of heterogeneous software and hardware platforms. More work is needed to further explore and refine this interaction between applications and heterogeneous cloud platforms.

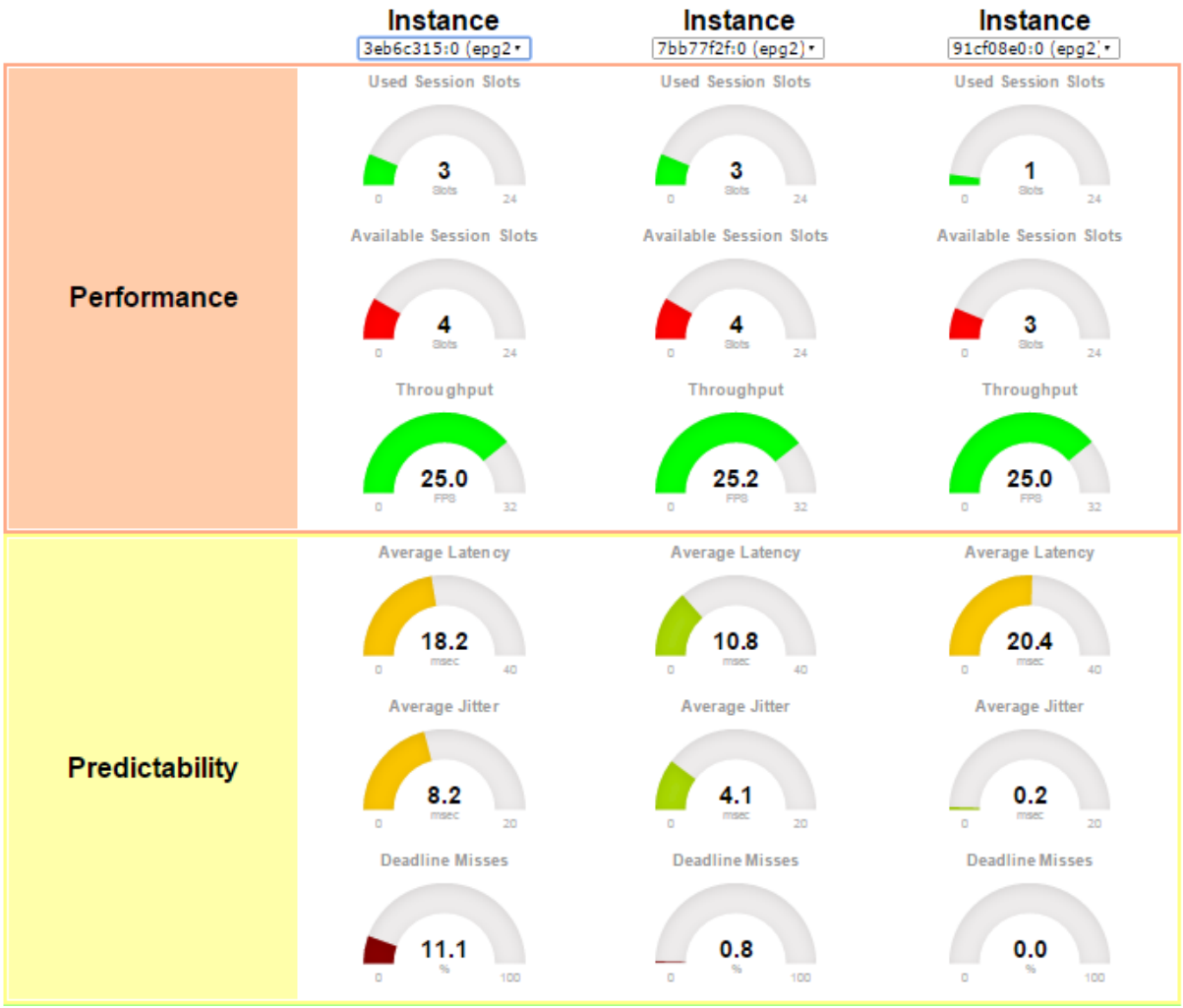


Figure 92 Application QoE metrics for one EPG service instance session deployed in each zone runtime environment.

5.19 Dynamic service graphs

This scenario is based on the setups described in 5.1, 5.10, 5.13 and 5.14, but enhanced by a service itself (not a user or dashboard) dynamically requesting another service and connecting to it, therefore dynamically changing the service graph of a composite service.

5.19.1 Testing plan

In this specific scenario a Shark 3D rendering service for a dashboard requests a EPG service to integrate the video stream from the EPG service into the dashboard, which composes a final video stream delivered to the thin client.

We executed this scenario with two different resolver prototypes: The Nokia and the Orange resolvers.

5.19.2 Involved components

- Orchestrator
- Resolver
- Monitor
- EPG service

- Streamer service
- Shark 3D rendering service
- Shark 3D simulation service
- Lobby software

5.19.3 Results

5.19.3.1 Single user session slot usage

As can be seen in the following screenshot, the zone manager of the Virtual Wall zone has allocated not only a session slot for the Shark 3D rendering service (as requested by the lobby software), but also for a EPG and a streamer component (as requested by the rendering service).

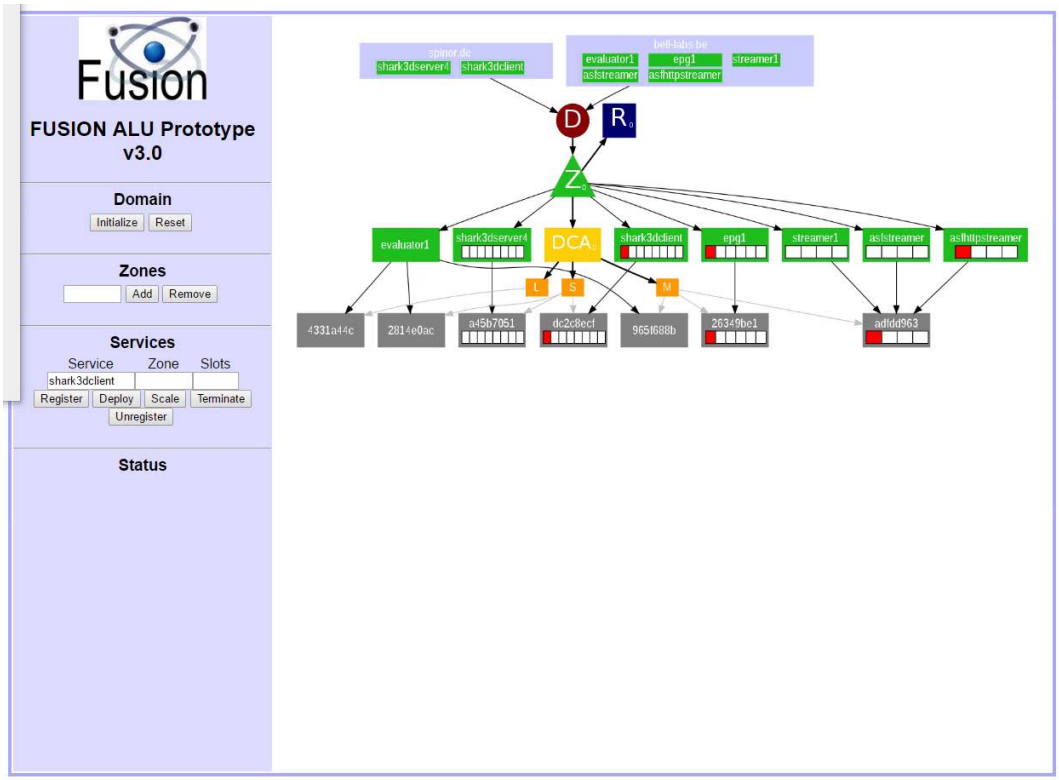


Figure 93 Rendering service using a EPG service slot

The following is a screenshot of the video from the EPG service streamed into the dashboard scene.

Figure 94 Screenshot of a rendering service integrating the output of a EPG service

5.19.3.2 Dual user session slot usage

The next screenshot is a sample of a dual-user dashboard scenario where the renderer services of the two dashboard users are synchronized by a dashboard simulation service, and each user having a different video integrated in their dashboard coming from two EPG services, each also needing a streamer service.

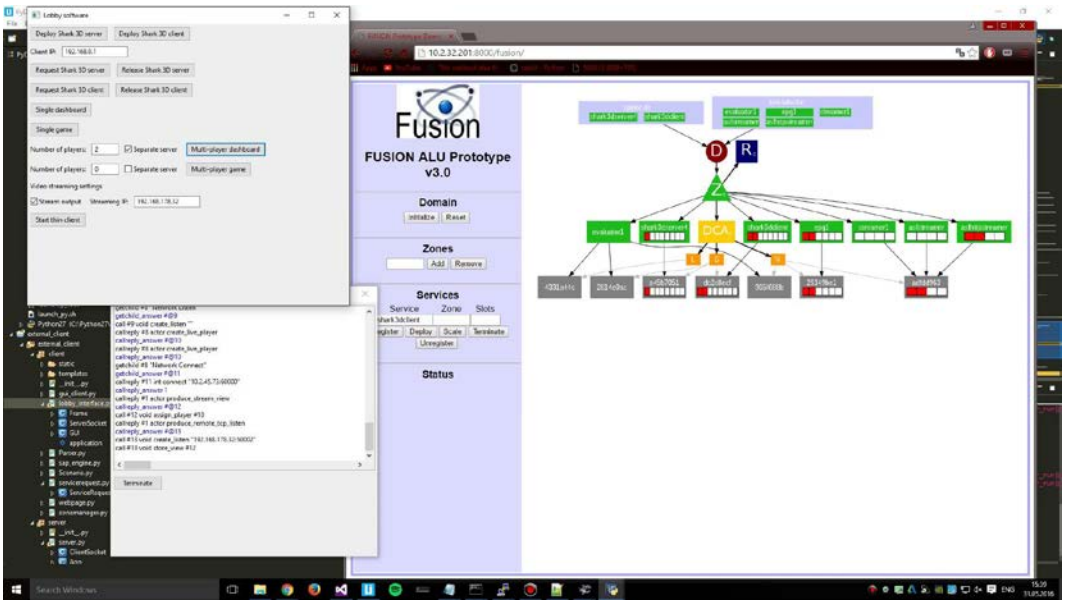


Figure 95 Dual-user dashboard using two EPG session slots

The dashboard contains different videos and different perspectives (camera positions) for each user, but the avatar position is synchronized:

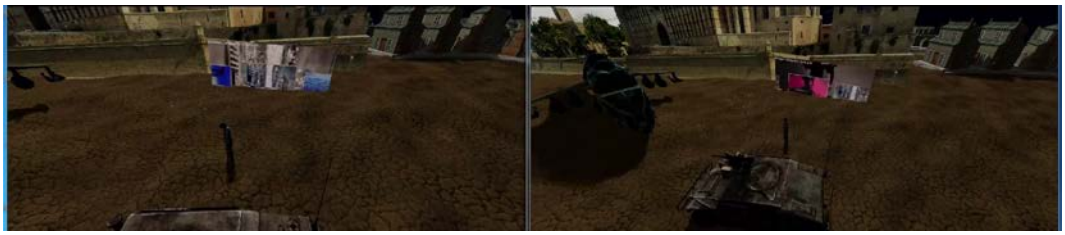


Figure 96 Two users of the dual-user dashboard scenario having individual EPG content

5.19.3.3 Performance over time

The following diagram shows the delay between the two clients over time.

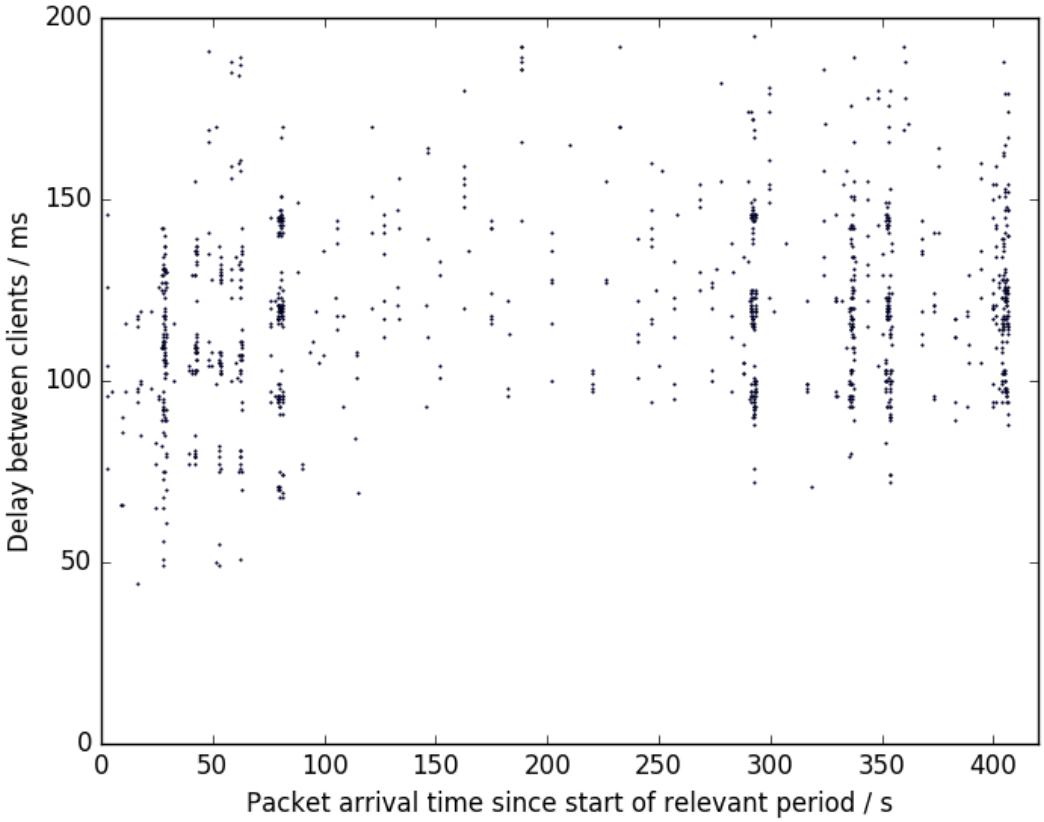


Figure 97 The delay between two clients over time

The deployment is similar to the low-cost scenario described in section 5.13.6.2, having a similar delay as the measurements described in that section, which is quite high compared to the high-quality scenario. At around one minute the EPG services are connected, which increase the load and therefore also the average response times and therefore in turn the delay between the clients slightly.

5.20 Application roundtrip latency versus framerate

5.20.1 Testing plan

In this test scenario, we study the effective total end-to-end roundtrip latency of a FUSION service such as the EPG on our testbeds. To measure the end-to-end roundtrip latency, we added a module in the thin client that sends input events at a particular rate at random moments in time. These are processed in the EPG rendering service by a special debug component that toggles the color of a 16x16 square in the upper left corner of the rendered frame. This rendered frame is encoded and streamed back to the thin client. When the thin client detects the toggled square, it measures how much time has passed since it sent the corresponding input event. This effectively emulates the time between a user pressing

some key and seeing the response, apart from the delay for capturing that input event (e.g., a key is pressed) and the visualization delay on some local screen.

5.20.2 Involved components

- Orchestrator
- Greedy Resolver
- 2D EPG Rendering Service
- Streamer Service
- Simple Latency Test Service
- Thin client

5.20.3 Results

The roundtrip latency measured in this experiment consists of three main components, as depicted in the diagram below: the two-way network latency, the processing latency and the framedelay latency.

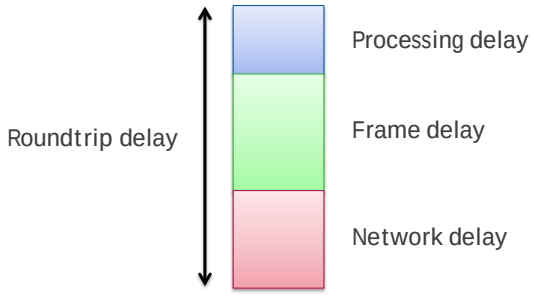


Figure 98. Breakdown of total roundtrip latency measured by our setup

The total network latency is the sum of the upstream network latency for sending a key event to the FUSION cloud service, and the downstream network latency of streaming the resulting video to the end user. The processing latency is the sum of rendering and encoding a frame in the FUSION service, as well as decoding the frame in the thin client. The framedelay latency is the latency between receiving the input event in the FUSION service and rendering the new frame. When using a fixed rendering frame rate (of e.g. 25 FPS), the average framedelay will be half the total delay between the rendering of two frames (i.e., 20 ms in case of 25 FPS). This is because on average, the input events will be received halfway through the waiting period in between rendering two frames. Note that with more dynamic rendering frame rates, it may be possible to reduce the average frame delay.

The latency tests have been done by deploying the prototype on the following four testbeds, and running the thin client in our local testbed in Antwerp:

Testbed	Location	Ping time from Antwerp
Local	Antwerp	~0 ms
vWall node	Ghent	~8 ms
Spinor	Munich	~45 ms
Jfed GENI node	Princeton NJ	~90 ms

Table 5 Four testbed locations

Each full testbed setup looks roughly as follows, though we will only present the results for the EPG+streamer service; the results with the other services are very similar and differ slightly w.r.t. the processing delay. Note that we ran each test for 1 hour.

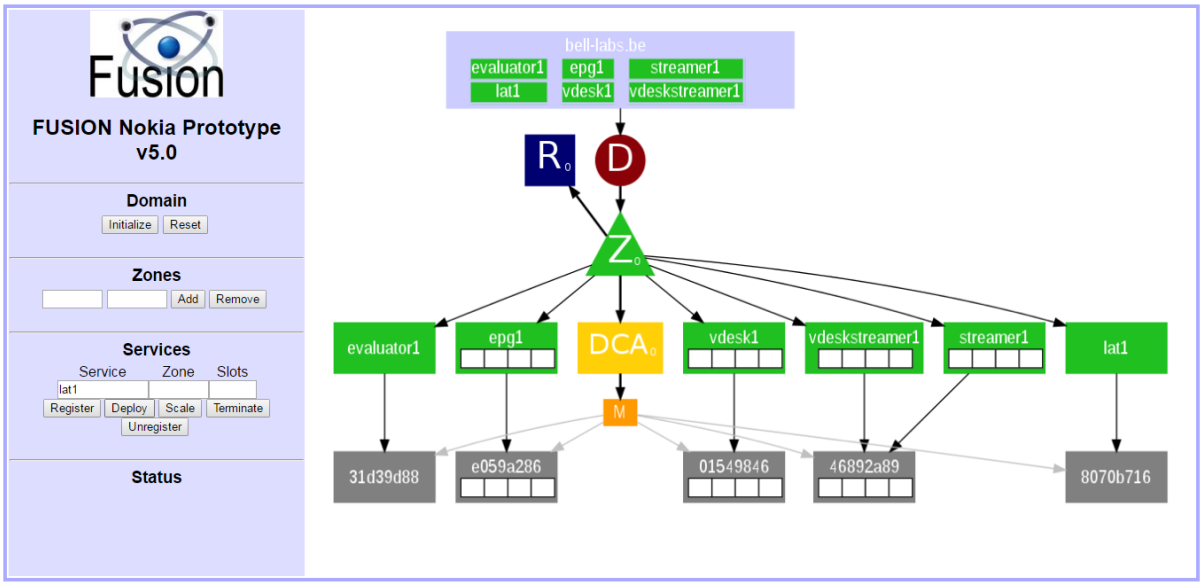


Figure 99. Full testbed setup involving all service components used in our experiments

In the figures below, we first present the average total roundtrip results as well as the standard deviation for the EPG+streamer service running at different rendering and streaming frame rates at the various locations. To reduce bandwidth related bottlenecks in this test, we chose the streaming bit rate very small (i.e., 1Mb/s).

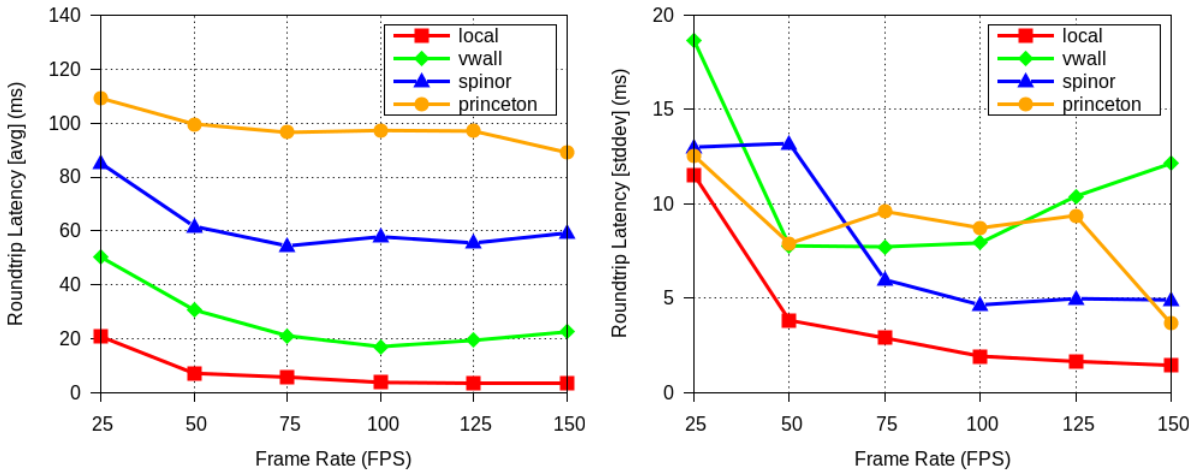


Figure 100. Average roundtrip latency (left graph) and its standard deviation (right graph) for the EPG+streamer composite service running at different frame rates, deployed at different locations.

A number of observations and conclusions can be drawn from these results. One key observation is that the rendering frame rate has a significant impact on the average roundtrip latency as well as the variation, especially for testbeds that are not too far from the thin client location. This obviously is due to the frame delay, which is about 20 ms at 25 FPS, and which also has a relatively large variation, depending on when the input event reached the rendering service w.r.t. the rendering of the next frame.

For example, running the service in a data centre about 50 km away at 75 FPS has a similar average roundtrip latency and with less variation as running the same service locally at only 25 FPS (even on

the same physical node). Similarly, running the service in a remote data centre about 1500 km away at about 75 FPS has a very similar average roundtrip latency (and variation) as running the service only 50 km away at 25 FPS. On the other hand, rendering at a higher frame rate will typically require additional runtime resources for rendering and encoding in real-time at this higher rates. However, as these remote services would be deployed in more centralized data centres, resources would be more cheap, resulting in a very interesting placement tradeoff.

The CPU utilization and system memory throughput of our EPG+streamer running at various frame rates is depicted below. Note that both the video decoding, rendering as well as encoding is done completely in CPU, not leveraging any hardware acceleration features. The decoder service, not being impacted by the rendering frame rate is obviously not impacted. The other two components however roughly scale linearly with the frame rate.

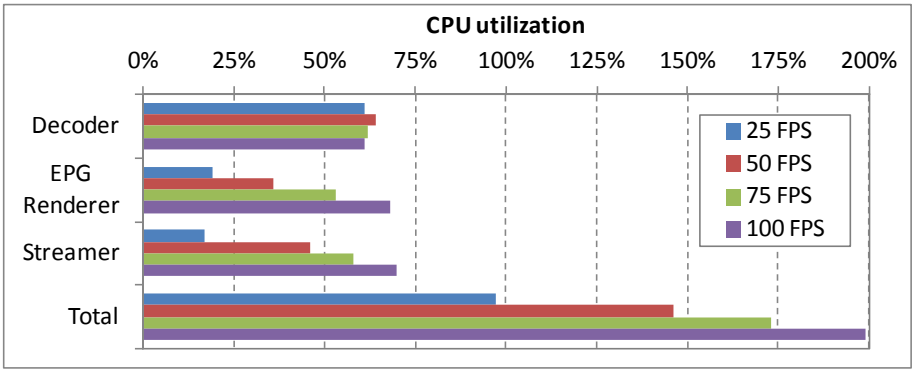


Figure 101. CPU utilization for the EPG service at various frame rates.

The overall system memory throughput also increases, but only slightly due to our internal memory pooling mechanism as well as caching.

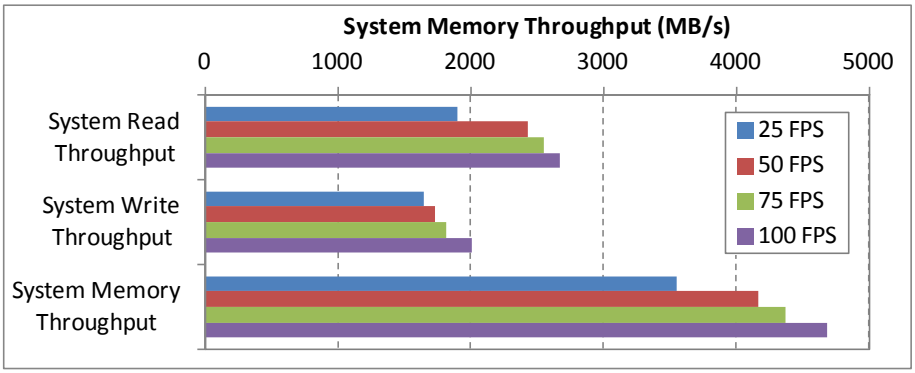
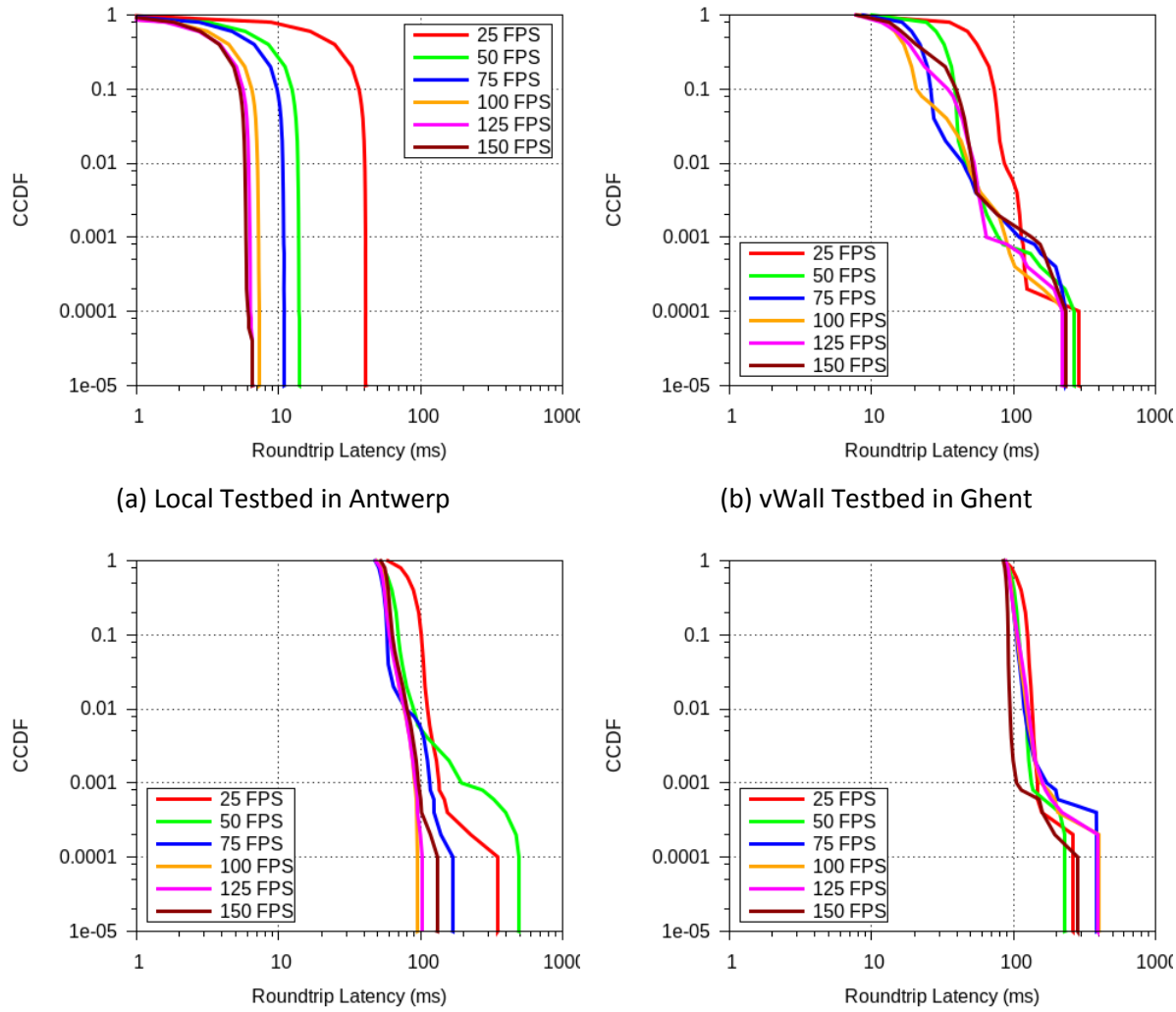


Figure 102. System Memory Throughput for the three deployment options

For completeness, it should be noted that application-level optimizations to reduce the frame delay when running at low average frame rates on the one hand, as well as rendering & encoding optimizations to reduce the overhead of rendering at higher frame rates, could be applied in both cases.

Finally, we also present the CCDF curves below, showing the tail latency behaviour when running these experiments from the four testbed locations for an hour. Obvously, the local testbed has almost no wide tail, whereas the others have worst-case tail latencies of well over 100 ms. The relatively bad vWall tail may be explained because of the VPN connection that was used to reach this testbed. Finally, as expected, the relative impact of the frame rate becomes less significant as the data centre is further away from the client.



(a) Local Testbed in Antwerp

(b) vWall Testbed in Ghent

(c) Spinor Testbed in Munich

(d) jFed GENI Testbed in Princeton NJ

Figure 103. CCDF plot of the roundtrip latencies of the EPG service deployed at various testbed locations for various frame rates.

6. CONCLUSION

6.1 Personalized services

In the area of personalized media applications, we conducted the following steps:

We implemented different prototypes of FUSION services covering the areas of gaming, augmented reality and media consumption. Some of the prototypes were created from scratch specifically as FUSION service. Other services were implemented on basis of existing commercial media software, which we extended into an authoring system for creating FUSION services.

6.2 Summary of integration and evaluation results

We integrated these service prototypes and accompanying software with the FUSION prototypes. Using this integration, we demonstrated and evaluated different FUSION features.

Main results are:

- The FUSION prototype passed all functional tests, including

  - different deployment results depending on different policies,
  - evaluator services,
  - session slot management,
  - resolution of services requested by a lobby software or by other services,
  - different resolution results depending on different user groups, and
  - static and dynamic service graphs.
- Various measurements showed different benefits of the FUSION prototype:
  - Different policies optimize for best quality versus lowest costs by resolving to suitable session slots in different zones.
  - Dynamic adaption to changing network situations (e.g. congestion) by changing the resolution behaviour accordingly to continue fulfilling the requested policy also in the changed environment.
  - Measurements over time, showing that actions (e.g. moving camera versus avatars or connecting to video streaming services within a dashboard) affect the measurement parameters a plausible way, but without significantly changing the quality.
- Feedback from human users confirmed a good quality of the prototype implementation of video streaming services, and supported the benefit of optimized service placement and resolution.

6.3 Future steps

From the perspective of implementing media services, we see multiple possibilities of simplifying the usage of service oriented networking:

- Additional integration of commercial software tools with service oriented networking software modules like developed in FUSION: Well-integrated tools are the key for exploiting the benefits for service oriented networking.
- Deeper investigation and implementation of security features: The prototypes described above neglect security aspects to a large degree in favour of verifying functional features.
- Implementing standard software components for easy creation of different kinds of lobby software supporting service oriented networking via FUSION: The ideal approach is an authoring system for lobby software as for media services as discussed in section 3.3.2. This makes it easier for application developers to take advantage of the various FUSION features, especially better quality thin-client based media application and scaling.

7. REFERENCES

[ABEA06] E. Altman, T. Boulogne, R. El-Azouzi, T. Jiménez, and L. Wynter, A survey on networking games in telecommunications, Comput. Oper. Res. 33 (2006), no. 2, 286–311.

[AFKS16] M. Aly, M. Franke, M. Kretz, F. Schamel and P. Simoens. “Service Oriented Interactive Media (SOIM) Engines Enabled by Optimized Resource Sharing”. In Service-Oriented System Engineering (SOSE), 10th Intl. Symposium, IEEE, 2016.

[CCLE14] W. Cai, M. Chen, and V. Leung, “Toward gaming as a service”, Internet Computing, IEEE, vol. 18(3), 2014

[DOCK16] Docker image layers visualization toolkit, https://github.com/justone/dockviz.

[FV09] F. Vandeputte, “Vampire Parallelization Toolchain”, IWT Vampire project, Deliverable D.B.4.3, 2009.

[Intel16] Intel Performance Counter Monitoring Toolkit, v2.11, https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[NVCOD] Nvidia, https://developer.nvidia.com/nvidia-video-codec-sdk

[NVGRID] F. Diard, “Cloud Gaming with Nvidia Grid Technologies”, GDC 2014

[NVGRID2] Q. Huo, C. Qiu, K. Mu, et al. “A Cloud Gaming System Based on NVIDIA GRID GPU”. In Distributed Computing and Applications to Business,Engineering and Science (DCABES), 2014 13th Intl. Symposium on, IEEE, 2014

[PLAYS] Extremetech - http://www.extremetech.com/gaming/175005-sonys-playstation-now-uses-custom-designed-hardware-with-eight-ps3s-on-a-single-motherboard

[SFL15] R. Shea, D. Fu, J. Liu, “Rhizome: utilizing the public cloud to provide 3D gaming infrastructure”. Proceedings of the 6th ACM Multimedia Systems Conference, ACM 2015.

[SHARK] Shark 3D – Spinor GmbH, http://www.spinor.com

[UNITY] Unity – Game Engine, https://unity3d.com

[UNREAL] Unreal Engine, https://www.unrealengine.com

[WXJ15] X. Wu, Y. Xia, N. Jing, et al. “CGSharing: Efficient content sharing in GPU-based cloud gaming”. In Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on, IEEE, 2015.

8. APPENDIX

8.1 Stateful 2D EPG rendering service component description

Below the full internal application structure of the stateful 2D EPG rendering service component. This application pipeline is created using our Vampire DSL language for composing real-time media pipelines. All other application service components created with our framework (i.e., the video decoder, streamer, cube, sphere, augmented reality, etc.) have a similar pipeline structure:

- A number of helper sub-pipeline definitions. In this example, there are two helper sub-pipelines, namely *Video* and *Photo*. The *Video* subpipeline will connect to a FUSION resolver to request a decoder service for decoding a particular external video file, do a format conversion if needed, and push the decoded video frames on a Vampire-internal shared-memory multicast. The *Photo* subpipeline opens a picture locally and does the same.
- The main EPG session rendering subpipeline, here called *EPGServer*. This subpipeline is spawned and configured dynamically for every connecting session. When the client disconnects, the subpipeline and all its runtime components are completely cleaned up and removed from the full application pipeline.

  This subpipeline description starts with the code that enables statefulness, followed by a number of rendering and monitoring/logging components, and ending with the interconnectively graph, described in a *dot*-like description format, as well as how components need to be mapped onto runtime threads.
- The main application pipeline (see *pipeline*). Apart from some runtime configuration and monitoring components, the core component is the *ServerSpawnWorker* component, which is the FUSION session slot specific component that handles the session slots and multi-configuration features. This component will automatically spawn and clean up the per-session subpipelines. This component is present in all our Vampire-based FUSION application service prototypes for creating and managing FUSION-enabled real-time media pipelines.

```
using source lib.fusion.fusionclient;

package Video(char *filename, t_fusion *fusion, char *decodeservice = "decode1.bell-labs.be")
{
        t_socket *inputsocket = fusion_client_connect(NULL, fusion, decodeservice, NULL);
        socket_write_vector(inputsocket, filename, strlen(filename));
        i: ClientReceiveY4MStream(NULL, 0, socket = inputsocket, mutex = TRUE, useshmpool =
TRUE);
        c: BGR2RGB(PLANAR);
                  m: ChannelSendMulticast(NULL, ignore = "_m");
                  i->c->m;
                  {i, c, m};
}

package Photo(char *filename)
{
                  i: JPGInput(filename, 0);
                  t: MaxThroughputPointer(24);
                  m: ChannelSendMulticast(NULL, ignore = "_m");
                  i->t->m;
                  {i, t, m};
}

package EPGServer(char *resolution = "540p", int serviceport = 5001, int fps = 25, int index =
0, int session = 0, int config = 0, t_bool browsing = TRUE, t_socket *clientsocket = NULL,
char *instanceid = "ID", t_bool debug = TRUE, char *logserver = NULL, int logport = 8050, int
statefulport = 25001)
```

```
{
        char *shmname = concat("/shm_%s_%d", instanceid, session);
        char *label = concat("%s - session #%d - config #%d", instanceid, session, config);

        t_namevalue *nv = namevalue_create(8, FALSE);

        t_socket *statefulsessionsocket = fusion_stateful_client_connect(clientsocket,
statefulport, nv);
        int statefulfps = fusion_get_intvalue(nv, "FPS", fps);
        char *statefulresolution = fusion_get_value(nv, "RESOLUTION", resolution);

                int height = atoi(statefulresolution);
                int width = 16*height/9;


        provider1: DefaultImageProvider(processallinputs = TRUE);
        provider2: SlideshowImageProvider("../../media/images/holidays");
        browser[2]: SimpleBrowser(width, height, statefulfps, eventqueue = index, epg=TRUE,
browser=browsing, running=TRUE, autobrowser=TRUE, processall = TRUE, throttle=FALSE);
        overlay: OverlayText(label, "../../fonts/FreeSansBold.ttf", 24, startx = 4, starty =
4+(debug-1)*24, display = debug);
        latency: OverlayToggleColor(16, KEY_SYSREQ, index, 2, debug);
                out: ServerSendY4MStream(serviceport, statefulfps, TRUE,
statefulsessionsocket, sendatonce = TRUE, mutex = TRUE, alwaysconvert = FALSE, @shmname);
                event: ReceiveRFBEvent(serviceport, eventqueue = index,
clientsocket=statefulsessionsocket);

        mon: MonitorPipelineLatency(concat("MONDATA%d", session), concat("s%d_browser",
index), concat("s%d_out", index), 0, statefulfps, skipframes=statefulfps, throughput = TRUE,
@logserver, @logport, lograte=statefulfps, ID=concat("epg2_%s_%d", instanceid, session));

                v1: ChannelReceiveMulticast(NULL, ignore = NULL);
                v2: ChannelReceiveMulticast(NULL, ignore = NULL);
                v3: ChannelReceiveMulticast(NULL, ignore = NULL);
                v4: ChannelReceiveMulticast(NULL, ignore = NULL);
                v5: ChannelReceiveMulticast(NULL, ignore = NULL);
                v6: ChannelReceiveMulticast(NULL, ignore = NULL);
        v1->provider1;
        v2->provider1;
        v3->provider1;
        v4->provider1;
        v5->provider1;
        v6->provider1;
        provider1->browser;
        provider2->browser;
        browser->overlay->latency->out;
        {browser, overlay, latency, out};
}

pipeline
{
                i: InterfaceServer(15001);
                r: RuntimeInterface();
                mgr: ChannelManager(NULL, waitpropertychannels = TRUE);

                char *resolution = "720";
                char *fps = "25";
                char *browse = "1";
                char *debug = "1";
        char *instanceid = concat("%d", runtime_get_system_id());
        char *logserver = "null";
        char *logport = "8050";

       s: ServerSpawnWorker(5001, totalsessions = 4, statefulstartport=25001,
              nstatefulports=16, cmdlist = {
```

```
                                    "RUNTIME_START_PACKAGE", "sIDX: EPGServer(clientsocket =
CLIENTSOCKET, index = IDX, session = SESSION, config = CONFIGURATION, serviceport =
SERVICEPORT, browsing = BROWSING, resolution = 'RESOLUTION', fps = FPS, instanceid =
'INSTANCEID', debug = DEBUG, logserver = 'LOGSERVER', logport = LOGPORT, statefulport =
STATEFULPORT)",
                                    "CONFIGURATION", "{v1->sIDX_v1, v2->sIDX_v2, v3->sIDX_v3,
v4->sIDX_v4, v5->sIDX_v5, v6->sIDX_v6}"
                                    }, paramlist = {
                                                    "RESOLUTION", resolution,
                                                    "FPS", fps,
                                                    "BROWSING", browse,
                                                    "INSTANCEID", instanceid,
                                                    "DEBUG", debug,
                                                    "LOGSERVER", logserver,
                                                    "LOGPORT", logport
                                    }, indextemplate = "IDX", logserver=logserver,
logport=logport, ID=instanceid);


        // perhaps I could also put this in a package and start it dynamically
        // now you need to set FUSION_RESOLVER_ENDPOINT as envvar
        t_fusion *fusion = fusion_create();
        char *decodeservice = fusion_get_env("DECODER", "decode1.bell-labs.be");
        //fusion_set_resolver(fusion, resolverip, resolverport);
        v1: Video("../../media/videos/720p/factory.mp4", fusion, decodeservice);
        v2: Video("../../media/videos/720p/elephant.mp4", fusion, decodeservice);
        v3: Video("../../media/videos/720p/ducks.mp4", fusion, decodeservice);
        v4: Video("../../media/videos/720p/bunny.mp4", fusion, decodeservice);
        v5: Video("../../media/videos/720p/life.mp4", fusion, decodeservice);
        v6: Video("../../media/videos/720p/pedestrian.mp4", fusion, decodeservice);
}
```

The *InterfaceServer* component allows an external tool to read and/or write particular application properties via some Vampire-specific internal API. This is e.g. used to add a new service configuration and read or update the used and available session slots. Combined with the *RuntimeInterface* component, we can also generate live views of the actual full internal application pipeline. Below two figures when 1 and 2 client(s)/session(s) are active, respectively. The red arrows represent the dynamic internal multicast connections.

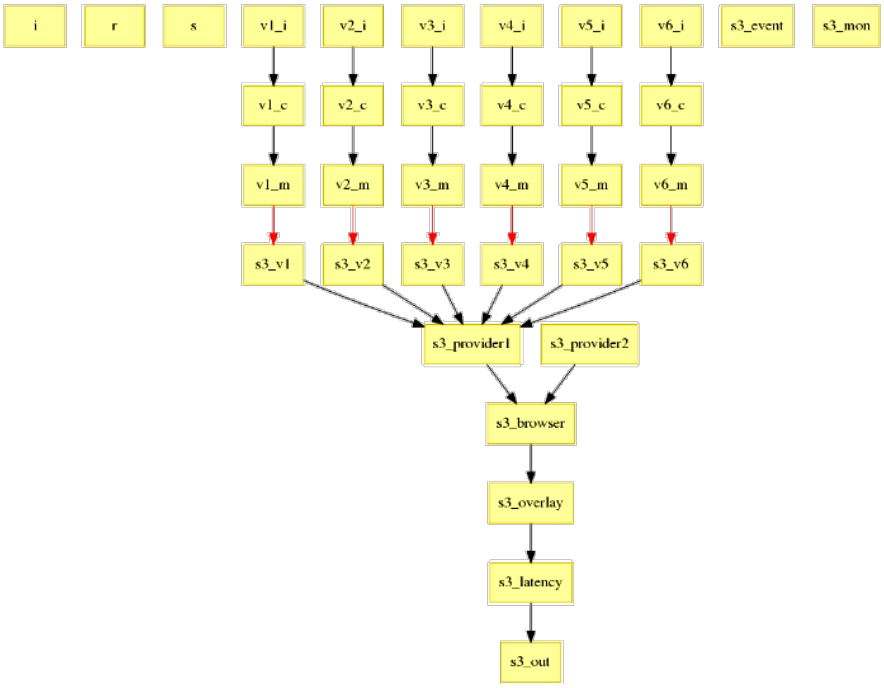


Figure 104. Actual EPG pipeline runtime state when 1 client is connected

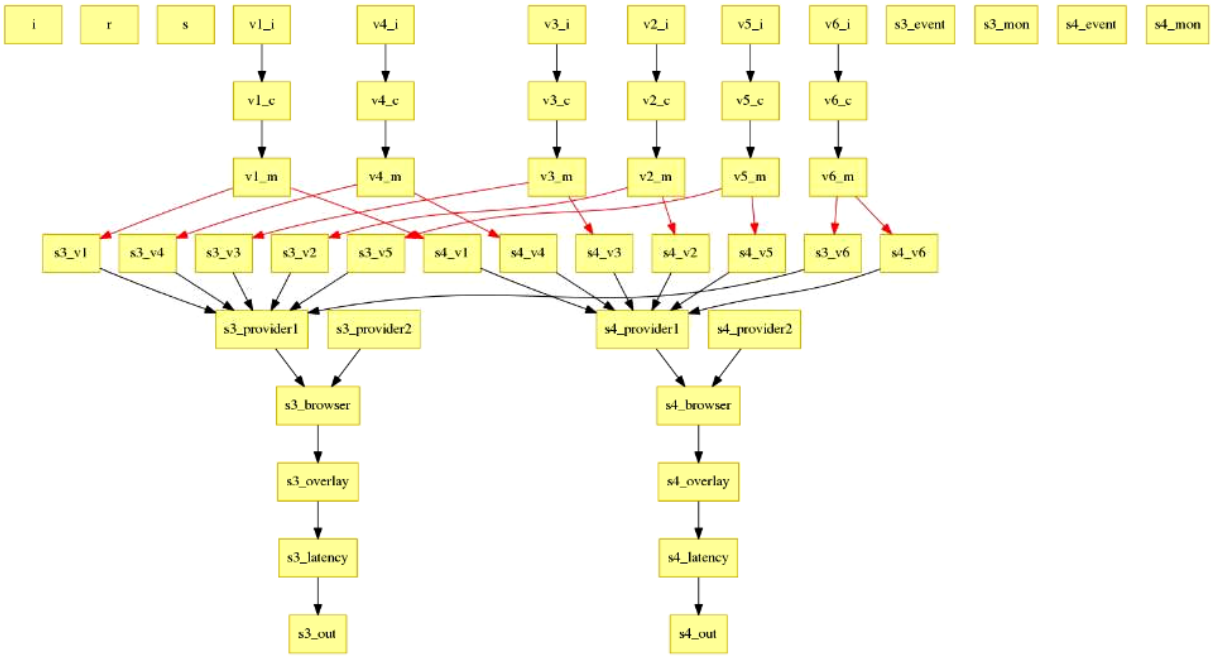


Figure 105. Actual EPG pipeline runtime state when 2 clients are connected