FUDGE-5G

FUlly DisinteGrated private nEtworks for 5G verticals

Deliverable D2.2

FUDGE-5G Unified Service Based Architecture Platform

Version 1.0

Work Package 2

Editor	Zoran Despotovic
Status	PU
Delivery date	June 2022

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957242

Versioning and Contribution History

#	Description	Contributors
0.1	ToC, first version	HWDU
0.2	General resource architecture description	HWDU
0.3	Resource scheduling architecture description	HDWU
0.4	Scheduling evaluation	HWDU
0.6	Service routing detail	HWDU
0.7	Architecture requirements, microservices considerations	IDE, ATH
0.8	Internal and external review	UPV
0.9	Improvements addressing the reviews	ALL
1.0	Final version - submitted	HWDU

List of Authors

Authors	Partner
Zoran Despotovic, Artur Hecker, Dirk Trossen, Karima Saif Khandaker	HWDU
Sebastian Robitzsch	IDE
Nicola di Pietro, Daniele Munaretto	ATH
José Costa-Requena	CMC
Thanos Xirofotos	UBI
Kashif Mahmood	TNOR
Filippo Rebecchi	THA
Hergys Rexha	AAU
Pousali Charkaborty, Marius Corici	FHG
Peter Sanders	O2M

List of Reviewers

Reviewers	Partner
Josep Ribes Rodríguez-Moldes	UPV
Bessem Sayadi	Nokia Bell-Labs

Acronyms

5GC	5G Core
API	Application Programming Interface
CB	Cell Broadcast

NF	Network Function
NPN	Non-Public Network
NRF	Network Resolution Function
PLMN	Public Land Mobile Network
RS	Resource Scheduler
SBA	Service-based Architecture
SCC	Service Chain Controller
SCP	Service Communication Proxy
UPF	User Plane Function
VAO	Vertical Application Orchestrator
WP	Work Package
USBA	Unified Service Based Architecture
MP	Management Plane
CP	Control Plane
UP	User Plane
RE	Resource Element
RCA	Resource Control Agent
SF	Service Function
SR	Service Router
R2R	Resource-to-Resource
ROSA	Routing on Service Addresses
CDN	Content Delivery Network
SAR	Service Address Router
SFC	Service Function Chain

FQDN	Fully Qualified Domain Name
CNF	Cloud-native Network Function

Executive Summary

This deliverable presents a possible realisation of the enhanced unified service based architecture. The key contribution of the deliverable is an attempt to bring in unification in all architectural aspects of the 5G and beyond mobile networks. The central part of this unification is a “general resource” with a set of well-defined properties, connects in a robust and scalable manner with other resources in the network, exposes its properties and enables hosting higher level services in a dynamic manner. This dynamic hosting of services is a key feature, which this deliverable is focusing on. It is based on proper scheduling of service requests to the resources, hosting them and having enough capacity to satisfy these requests in an SLA compliant manner.

1. Introduction

Work Package (WP) 2 is chartered with the implementation of FUDGE-5G components and interfaces defined in the architecture WP, WP1. WP2 is organised across the five tasks:

- T2.1: Unified Service Based Architecture Platform.
- T2.2: LAN in 5G Environments.
- T2.3: Cloud Native Service Orchestration.
- T2.4: Disintegration of Network Functions as Micro-Services.
- T2.5: Platform Continuous Integration in a Sandbox Environment.

Unlike other tasks, T2.1 (Unified Service Based Architecture Platform) takes a more futuristic view and implements a component, resource scheduling namely, that is not meant to be deployed as part of the FUDGE-5G system in the addressed use cases. Instead, this component is expected to play a critical role in the enhanced SBA architecture, which is more appropriate for beyond 5G network, or even 6G.

On the one hand, resource scheduling is supposed to bring in efficiency in the operation of mobile networks. This is a first and natural conclusion that comes with an attempt to dimension the network not for the peak load of every service type that has to be supported, but to multiplex these various services onto available resources and drive their execution such that their service level agreements remain satisfied. On the other hand, as this deliverable describes, service scheduling is a crucial step towards turning the mobile networks into service execution platforms, a long desired goal of mobile network operators.

This deliverable takes the standpoint that besides service scheduling, network unification is the other necessary ingredient. Unification is in this deliverable meant as a rather broad term, it applies to all resources in the network. Precisely, no resource (type) (e.g. wireless link, i.e. wireless access nodes) is given a special treatment, and thus no architectural decision is made that favours a resource or type. Instead, all resources are equal, they connect in a robust resource mesh, disseminate their properties to a set of network control points and expose appropriate APIs that enable, among others, service deployment and execution.

Not that we used the word *scheduling* above, as opposed to *orchestration*. We are not focusing on orchestration in this deliverable. Orchestration is essentially a network management concept, whereas we want to focus here on network operation at a finer granularity level, on its runtime (of both control plane and data plane). To illustrate, orchestration should be equivalent to a decision to build a highway between two regions or cities, along with gas stations and other accompanying infrastructure. Scheduling would then pertain to decisions for every single car, which lane(s) and gas station(s) to use at any moment such that an appropriate metric is optimal.

2. Architecture Requirements

2.1. FUDGE-5G Architecture Requirements

This section describes the FUDGE-5G requirements towards a system architecture that supports unified methods and procedures of a Service-Based Architecture (SBA).

2.1.1. Components to be considered

As introduced in D1.2 [10] and illustrated in Figure 1, FUDGE-5G introduces a dedicated platform layer between the infrastructure and services. The functionalities of the platform layer are routing, orchestration, telemetry and slicing, and is collectively referred to as the Service-Based Architecture (SBA) platform. As services operating “on top” of the platform utilise the platform functionalities through well defined, open and programmable APIs (dotted lines in Figure 1), the proposition for unification can be directly derived from the platform layer.

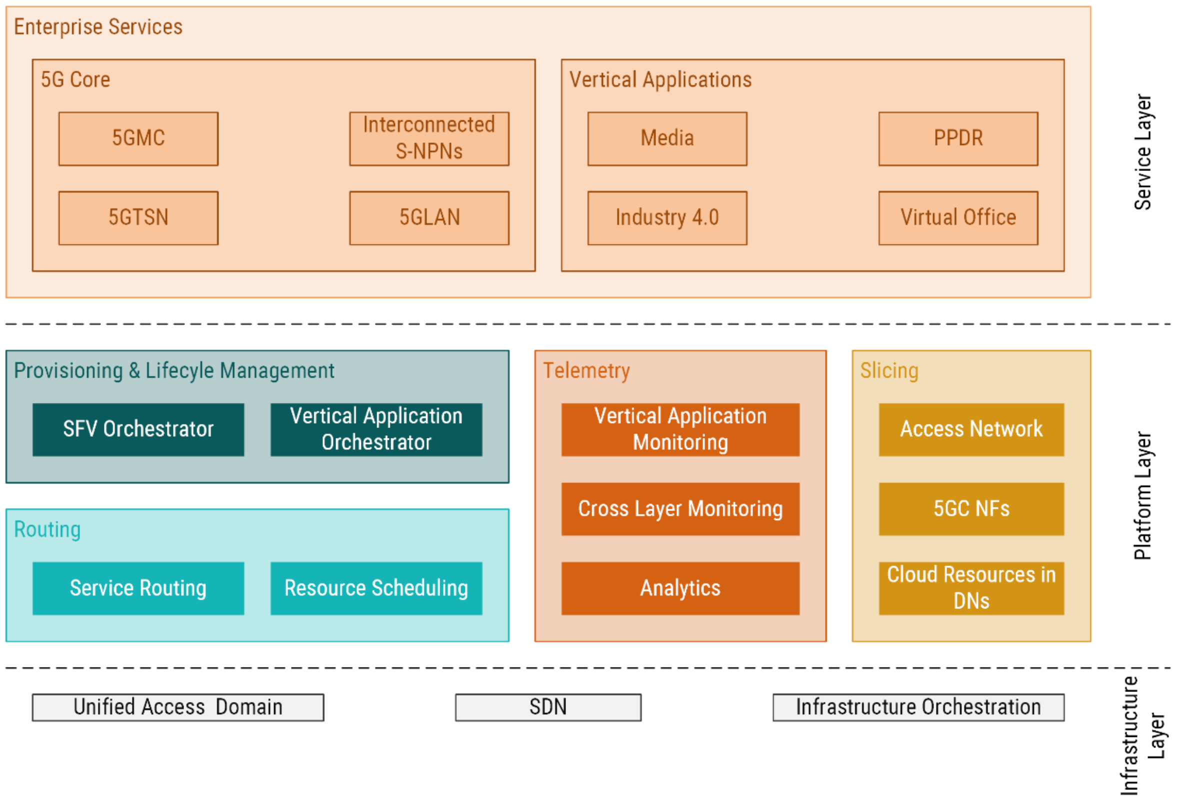


Figure 1: FUDGE-5G System Overview [10]

As such, the components illustrated in the system overview of D1.2 are the components under consideration for this deliverable, namely:

- Routing
- Provisioning & Lifecycle Management

- Telemetry and
- Slicing

While orchestration of is not of concern for the system architecture of a mobile telecommunication system (i.e. standardising this within 3GPP SA groups), as well as slicing capabilities of a 5G Core Network, the considered components in this Deliverable are Service Routing with input from Telemetry. The rationale to focus on those two components is as follows: in a public cloud, that comes as an Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) offering, the tasks of routing and orchestration is offered as part of the IaaS or PaaS package. Through descriptors and web APIs resources are configured accordingly and the services and functions deployed in these environments do not have to worry about how packets are routed and how the virtual instances get deployed and lifecycle managed. As monitoring is a key capability for lifecycle management decisions, the capabilities of collecting data points from deployed services and functions. Finally, when interpreting slicing capabilities as a task of resource isolation and QoS enforcement, there is numerous areas which are either outside of 3GPP's scope or not defined (yet). And as 3GPP's 5G System specification (23.501) already defines the Service Communication Proxy (SCP) as well as the Network Data and Analytics Function (NWDAF), the focus for a unified SBA platform is put on the components Routing and Telemetry in this deliverable.

2.1.2. Exposed platform component interfaces

As illustrated in D1.2 and describe beforehand, the FUDGE-5G platform layer exposes programmable and open APIs towards the service layer for all platform components (dotted lines in Figure 1). To support programmable Routing and Telemetry, the interfaces to do so are defined in D2.5 (to be published in November 2022) and described in their functionality in more detail in D1.3 (the updated architectural deliverable due in July 2022). In essence, the Routing component offers an open and programmable interface for services to programmatically change the routing behaviour of the platform, in particular for stateless application protocols, e.g. HTTP. The interface also allows to define any desired affinity between two service instances, where an affinity is defined by the routing layer to keep the exchange of messages between a specific set of two service instances. Also, the routing policies can be set such as shortest path routing or constraint-based shortest path routing.

The Telemetry component also offers an open API for data ingest in a time series fashion. While agnostic to the actual data points themselves, the Telemetry components requires an implicitly known information model across all services to form a unified representation of data points and the ability to run analytics over. More information on that can be found in D1.3.

2.2. Architectural Advances towards Unified Service-Based Architecture

This section presents the rationale behind the architectural advances towards a unified Service-Based Architecture and the work conducted in FUDGE-5G to achieve that.

2.2.1. Unification Scope and Architectural Extensions

Unification shall be understood as an attempt to streamline the methods and procedures within a 5G system without allowing any permutation. The ultimate purpose for the presented unification herein is to evolve SBA to enable greater flexibility for operators in deploying and operating a mobile telecommunication network with software components from different vendors. In order to define the scope for an architectural system unification, a reference system architecture is required as the foundation. 3GPP's System Architecture Working Group 2 (SA2) [11] serves as the reference architecture. This architecture is illustrated in Figure 2 and illustrates 5G Core Network Functions (NFs) in dark green. All NFs with a non-numeric interface name offer a Service-Based Interface (SBI) which is characterised by a (by 3GPP) well defined HTTP-based API with JSON-encoded payload.

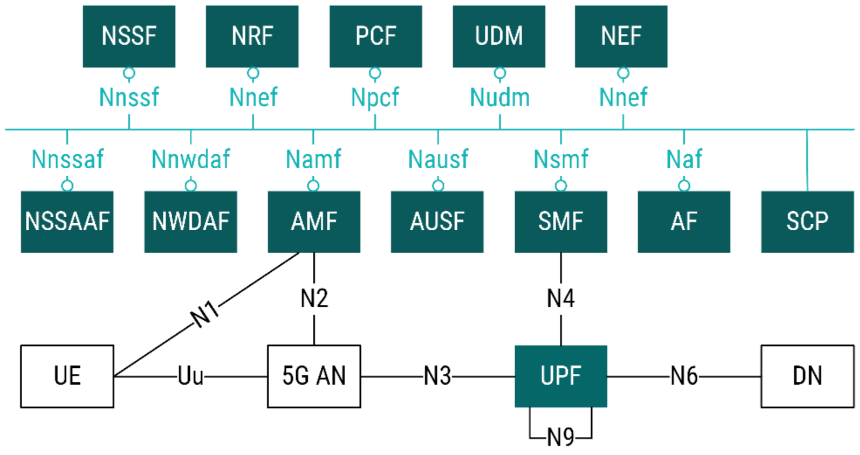


Figure 2: 3GPP's 5G System Architecture in Release 17 [12]

The scope of unification in relation to routing and telemetry concerns the 5G system components Service Communication Proxy (SCP) and Network Data and Analytics Function (NWDAF). As will be described in further detail in D1.3, the SCP as well as the data collection capability of the NWDAF are positioned as part of the FUDGE-5G platform and made mandatory for all services (5G Cores) to be used. The resulting unified system architecture is illustrated in Figure 3 and has the following changes towards a unified Service-Based Architecture:

- The SCP is not a Network Function any longer that can be addressed explicitly
- The NWDAF is split into a Network Monitoring Function (NWMF) and Network Analytics Function (NWAF)
- The discovery of NF instances or NF sets is removed from the NRF [12], allowing the SCP to schedule all HTTP transactions based on available resources. However, in order to obtain the Fully Qualified Domain Name (FQDN) used in a deployment that identifies a Network Function, the Who Am I Function (WAIF) has been introduced and fully decoupled from the NRF, allowing it to be placed inside the platform.
- Allowing the SCP to route N2 and N4 traffic between ANs and AMF as well as UPF and SMF, respectively. Furthermore, it is enforced that ANs and UPFs must address their peers (AMF and SMF) through FQDNs instead of IP addresses. Thus, FUDGE-5G can

ensure cloud-native procedures for the orchestration of Enterprise Services. For 5GCs this results in no post-deployment operations and/or configurations of AN, AMF, SMF and UPF when it comes to finding the addressing identifier of the peers they aim to reach.

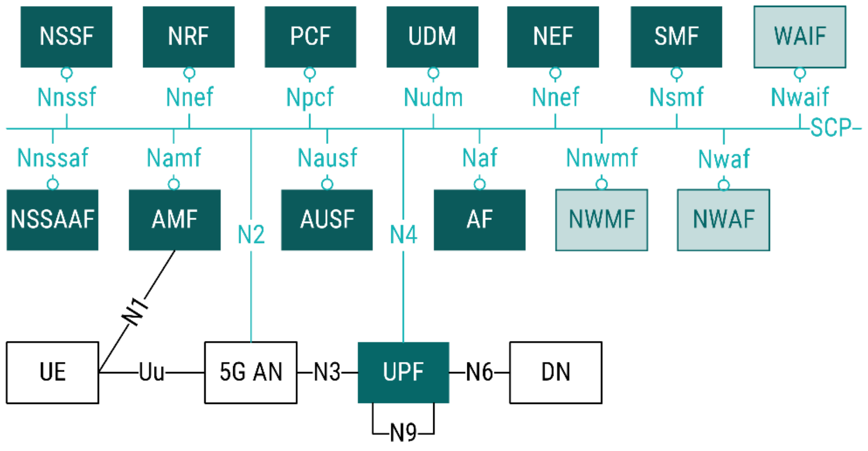


Figure 3: Proposed Beyond Release 17 System Architecture

When mapping the proposed system architecture from Figure 3 to the FUDGE-5G system from Figure 1, the NFs SCP, WAIF and NWMF are part of the platform layer and come with every deployment of the SBA platform (Figure 4).

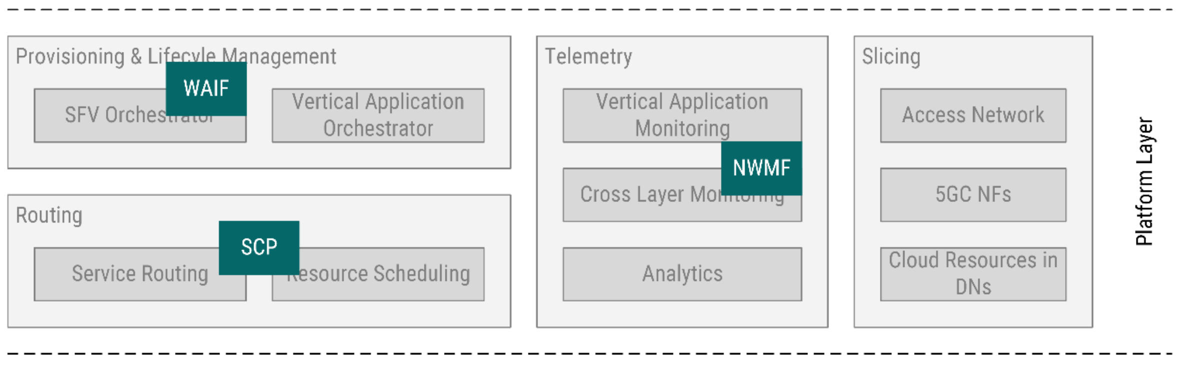


Figure 4: Applied System Architecture to FUDGE-5G Platform

2.2.2. Scaling considerations

SBA is built upon two core requirements: 1) the ability to vendor-multiplex the deployment of a 5G system and 2) the ability to scale the 5G system with the demand across several locations (cloud regions). While physical capacities have finite upper limits, the deployment of a mobile telecommunication network with all its software components is still subject to network planning with regards to dimensioning the various components, including the 5G Core. While this task and the resulting effort can be justified by the deployment of a national Public Network, it becomes inevitably harder to justify in a Private Network setting due to the significant different requirements for each Private Network environment and the ability to potentially deploy Private Networks for a much shorter timeframe. Furthermore, with traffic demands expected to further increase in an exponential fashion and the desire to optimise operations for a greener operation of a mobile network, optimisations around energy consumption and energy efficiency do require a flexible, scalable and programmable system.

FUDGE-5G sees SBA and the unification work presented in this deliverable as a key contribution towards a more flexible approach beyond PaaS and data centre-centric technologies, such as Kubernetes or OpenShift, to deploy and operate a mobile telecommunication system. In particular the ability to offer location- and resource-aware orchestration and routing capabilities, fine-tuned towards the telco domain, is at the core of the unification efforts and considered as the evolution of SBA.

2.2.3. The impact of microservices

As outlined by NGMN's project "Future Networks Cloud-Native Platforms" [13], the evolution from Physical Network Functions (PNFs) to Cloud-Native Network Functions (CNFs) is mainly driven by the cloudification of Network Functions in the way they are deployed into an infrastructure and how the Network Functions are implemented in software. For CNFs, the adoption of the 12-factor app methodology is seen as the set of design patterns for software engineering to transform a monolithic software into a set of microservices. The ultimate outcome of a microservice-based software realisation is a higher number of executables that have a small code base (lines of code) and can ideally operate in a stateless fashion. Ultimately, this allows to scale the number of microservices of the same type with the demand of incoming requests, allowing to keep the entire service the set of microservices implement always up and running.

In the perspective of deploying 5G core networks as services within the proposed framework, FUDGE-5G is specifically addressing the problem of re-designing 5G core networks with a microservice-based approach. This entails the decomposition of 5G core network functions into actual microservices, an activity addressed within T2.4 and reported in D2.4. From the architectural point of view, such a decomposition is not trivial, because it requires the identification of dependencies among each functionality offered by a 5G core network function, and the separation and conglomeration of such functionalities into independent "clusters" that are eventually programmable as actual microservices. In particular, as further outlined in D2.4, there is not a one-to-one mapping between the services produced by each 5G core network function as defined by 3GPP (TS23.501) and a coherent decomposition of such a function into microservices.

As microservices exchange information among each other to offer the intended service, the routing of messages among them as well as the decision which instance to choose from a set of available microservices of the same type must be addressed. This is what the work in this deliverable addresses as a proposition of the Service Communication Proxy.

2.2.4. User plane considerations

Extending FUDGE-5G's architectual advances of the Control Plane to the 5G User Plane requires rather disruptive changes to 3GPP's architecture, in particular on the signalling procedures on the Control Plane. As any service routing capability wihtin the SCP will not be of concern to 3GPP's standardisation efforts, it is a less intrusive effort to introduce these changes. 3GPP's User Plane however is fully controlled by the 5GC and any on-path routing approach demands radical changes to all architectural components that play a role on the routing decision for User Plane packets. Therefore, the 5GC must offer the ability to support the required signalling on the Control Plane to achieve that (N1 for the UE and N4 for the UPF) as well as making decisions about routing decisions in a distributed fashion on the "on-path" components, if needed.

For instance, the support of Name-Based Routing on the User Plane (described in D1.2 [10]) has been designed to require almost no changes on the 5G Control Plane. Instead additional methods and procedures on the UE, UPF and SMF have been proposed to operate on top of established 5GLAN-based PDU sessions. However, when integrating any service routing solution on the 5G User Plane with proper support of the entire 5G system, the current protocol stack of the User Plane requires either heavy signalling between UE, UPF and gNB with the 5GC or a radical change of the protocol stack. As illustrated in Figure 5, the N3 interface between the gNB and UPF uses the GTP-U protocol to map PDU flows to UEs and to control their QoS requirements. Any changes to the PDU flow between a gNb and UPF requires an update of the GTP-U information on the gNB and the UPF via the AMF and SMF, respectively.

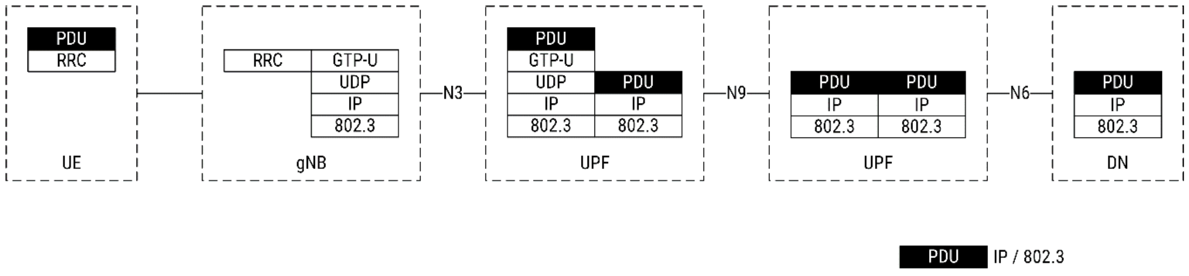


Figure 5: Protocol Stack of 5G User Plane

Furthermore, the inter-UPF interface (N9) and UPF-DN interface (N6) also have their own IP routing layer with potential VLAN/VXLAN or L2/L3 VPN configurations for added slicing and/or security considerations by the operator. If local break-out points are configured by the operator it must be ensured that the 5GC can freely move PDU flows to different UPFs and DNs without changes to the underlying transport network.

The last architectural consideration for the 5G User Plane is Device-to-Device scenarios. While UEs can communicate to each other via the UPF as their anchor point (or multiple UPFs interconnected via N9), 3GPP is also working on scenarios where a UE uses another UE to reach the gNB (called sidelink). If service routing capabilities are brought to the User Plane, these scenarios pose further challenges to allow any unified service routing capability, as presented in this deliverable.

3. Developed Unified Service Based Architecture (USBA)

3.1. USBA High level description

The main purpose of the Unified Service Based Architecture (USBA) is to create a dynamic operational resource control environment to efficiently run mobile network services, such as per 3GPP 5G specifications and its typical implementations and beyond 5G.

Before we start, it is worthwhile to define a number of terms, as they might slightly differ from what the reader is used to. The first notion is the notion of System, notably the system under consideration. While we generally consider mobile telecommunication systems, in this text, we mainly refer to an important constituting part of the whole, notably to the mobile core network system (the so-called 5G Core, 5GC), providing different services to the RAN, to UEs but also to the mobile network operator. We notably concentrate on the "Service Based architecture", which, in Rel15, is only concerned with the Control Plane of 5GC. What is important to observe is that by changing the notion of the system under consideration, we also change the semantics of both services and of constituting modules.

Second, for the core network realization, since 3GPP Rel15 (TS23.501), the notion of "Network Function" (NF) has become common. Technically, this term is not very precise, as such NFs are actually not functions, but implementation modules of the 5GC, together realizing different services consumed by its various users as explained above, i.e., by the other parts and strata of the overall mobile telecommunication system (e.g. by the RAN) or by the subscriber UEs or by mobile network operator (management services). All NFs together constitute the overall Rel15 Core (and notably its user, control and management planes) and realize different parts of the overall required functionality, i.e. of the overall expected service. Herein, some NFs directly map to expected services, e.g. UDM implements major parts of the subscriber management services. Others, like AMF, implement only a part of the overall mobility services, with the more important part being implemented in the RAN's gNB. Note that NFs themselves can be of different realizations: while some NFs can be monolithic, some others can actually be distributed systems per se: e.g., while an SMF could be a monolithic entity tying a part of session control at one particular realization point, a UDM NF could be realized as a distributed data base system (DBMS). In particular, for monolithic NFs it is quite usual and useful to raise the question of the number of running instances, i.e. of the number of active entities of this particular type. In particular with the increase of popularity of virtualization in the realization, where many NFs are implemented as containers or virtual machines, it has become quite simple and common to add and to remove NF instances to /from the running network.

The name USBA leans itself to 5G Rel15 SBA (TS23.501). However, while SBA is an architecture strictly limited to network functions and service interfaces of the control plane of the core network of the operator, with only very rudimentary resource mappings (e.g., as per NRF: NF type to URL list), the proposed USBA:

- Is not limited to either core or control plane, but, instead, explicitly regards every mobile network function as a service element to be run on some resource. (The exact meaning of the term resource, as we use it in this deliverable, will be explained shortly, see Section 3.1.2). With this, USBA can support different elements from the mobile network, including MP, CP, UP, RAN (as, for example, O-RAN), and even elements rather pertaining to the implementation of the mobile network (e.g. as per 3GPP CT4 or proprietary), such as security gateways (SeGW) and proxies (e.g. SCP).
- Includes explicit considerations for both network functions (i.e., service elements) and the resources that these functions run upon. Herein, USBA handles both resources and service elements on top of those resources with the same dynamics, i.e. it includes considerations for both compute and networking resources, and it allows both network functions and the underlying resources to be dynamically added and removed from the network. This accounts not only for unstable, virtual and mobile resources but also for the mutual dependency of resources and services that run on top of these resources (as discussed later).
- Supports runtime service scheduling, i.e., beyond pure directory mappings, USBA is explicitly capable of diverting the incoming service requests to respectively suitable network service instances in runtime and within service request constraints. Here, “suitable” means that the selected service instance with high probability:
    - is reachable for the request originator with the current resource and network function situation,
    - is running and available,
    - yields the best known service quality.
- Optionally can support/include network verification means, conflict resolution means and network garbage collection.

USBA is able to account for the heterogeneity (different capacities in terms of storage, networking compute; different capabilities; different purposes; different needs; different natures of implementation), dynamics (mobility provoked through physical movement like e.g. in Cell on Wheel (COW), changes in configurations, stability over time e.g. provoked by activation-creation or deactivation-removal) of both resources and different functions running on top of those.

The heterogeneity of the considered resource pool is mainly due to the needs of the very different types of functions in the operational mobile network (e.g. packet processing functions, flow payload processing functions, session tracking functions, directories, data bases, security functions including dedicated per-packet processing and stateful tracking). While the dynamics of the resources is typically provoked through external influence (e.g. by third party providers such as energy providers, physical resource owners, by errors, failures and overloads), USBA accounts for the mutual dependency of service elements and the underlying resources. While it is straightforward to see the dependency of the running service element on the hosting resource, it is often overseen that a service element can alter the reachability, visibility, capability or capacity of either the hosted resource or many other resources by creating, diverting and blocking service loads. Consider e.g. a SeGW element blocking all uplink traffic of a particular type or traffic to particular address ranges; this would effectively change capacities or reachability of some resources.

While USBA as such clearly distinguishes functions and resources, note that this does not preclude the usage of function-dedicated resources in the implementation, typically in situations, where it is more suitable (where the whole resource is the service function, e.g., sensors and actuators) or required for better performance (e.g., particular accelerators, like in dedicated packet processors, NPUs for ML processing, etc).

3.1.1. Used technologies

USBA accounts for a modern ICT landscape, typically consisting of some mix of the following resources:

- Physical resources such as spectrum, cables, network connections, fiber, switches, routers, NAS boxes, packet processors, compute accelerators spread over some territory in form of interconnected data centers, cabinets, base station poles, etc.
- Virtual resources, obtained through some form of host virtualization technologies on physical resources or groups thereof, such as virtual machines (VMs), containers, virtual switches, virtual routers, virtual channels, lines or connections.

Generally, USBA expects resources to be basically controllable by some technical means. In practice, USBA considers that it is authorized to (has credentials) and can operationally access (there is either existing physical or USBA-controlled logical connectivity to) some control API of each respective resource. This also explains what is meant with controllable network links, which, per nature, are passive.

More precisely, USBA only considers resources that can be controlled and simply disregards resources, which are not visible or not active from its point of view, i.e., which it cannot at least monitor. All these resources are considered in the resource pool, and the purpose of USBA is to assure that these remain controllable at any moment in time in spite of resource churn and provided service configuration and load changes.

Herein, the actual implementation of the resource control API is out of scope of USBA and could be substantiated by e.g.:

- Direct, resource-local control means, with resource itself offering an API that enables control of various its aspects, e.g. through the operating system means or by explicit hardware integration.
- Indirect, remote control means, which might use some form of connection and protocol to control the respective resource aspects. Such protocols can include but are not limited to various management protocols (e.g., SNMP, NETCONF, WBEM). In management slang, in this case, the controlled resource is the Managed Object at the respective Manager. This second way might be required sometimes but is actually suboptimal, as it might incur additional limitations and latency.

Note that the virtualization technology is explicitly accounted for by USBA as an important enabler. Indeed, one of the purposes of USBA is to support the dynamics of the resources precisely because USBA acknowledges that with modern technologies (e.g., IaaS) resources could be themselves provided as a service (by a third party, i.e. public cloud, or by the same provider, i.e., private cloud). Note that modern virtualization environments export explicit (runtime) control APIs for all the obtained virtual leases/resources (e.g. to VPNs, to VMs, containers, etc. – cmp. e.g. OpenStack, VMWare offerings, etc). This is something that USBA is specifically designed to be able to seamlessly integrate and make use of.

At the same time, USBA does not rule out physical resources. The opposite is the case: modern "programmable" network gear offers explicit and standard control APIs (e.g. OpenFlow [17], I2RS [18], P4 [2]). Depending on the setup (authorizations and credentials, system integration), USBA can either consider the respective controller (e.g. the OpenFlow controller) as the API to all of the controlled switches, or it could be used to provide the control connections from the Controller to the Switches, both of which become USBA Network Functions in this view. Other network-controllable physical resources can be used as well.

As USBA generally supports resource dynamics, it can support mobile (moving) resources and churn at the resource layer. To do this, USBA generally uses many principles of self-* systems. It is per itself conceived as a distributed system, tries to avoid single point of failures, requires only minimal initial configuration and, with only this minimal configuration (essentially representing the security association) features self-bootstrapping and self-maintenance of its components over the lifecycle.

USBA integrates the different control means in a uniform fashion seeking to get a uniform API to various resource types in a systematic manner. Note that in doing so, USBA-uses a holistic system approach: its own means and resource consumption are always explicitly considered as part of the operations on the very same resource pool, i.e. of the same system. Accordingly, all USBA control channels are "in-band", i.e. provisioned through and by the means of the controllable resources from the very same controlled resource pool. Thus, all compute operations of USBA itself are instantiated as services on top of the same resource pool. Finally, USBA overheads (costs in terms of energy, additional latency, loads on links or on compute nodes) are always considered in the overall service optimization.

Consequently, for the service to be provided, USBA cannot support service elements, the hosting resource of which is not visible or appears as passive to USBA (a cable, a non-managed bridge, etc.). By insisting on having explicit handles on all resources involved into the considered service delivery, USBA is distinct from pure overlay solutions, which concentrate solely on the obtained virtual elements. USBA notably supports "underlay-aware" overlay solutions and assumes that it can act on both the resource and the service layers. However, USBA is a real extension to the overlay approach: if all considered resources are virtual, then USBA can be essentially equivalent to a "blind overlay", even though it would still distinguish resources from the services.

3.1.2. USBA and its components

As shown in Figure 6, USBA distinguishes the following layers and components:

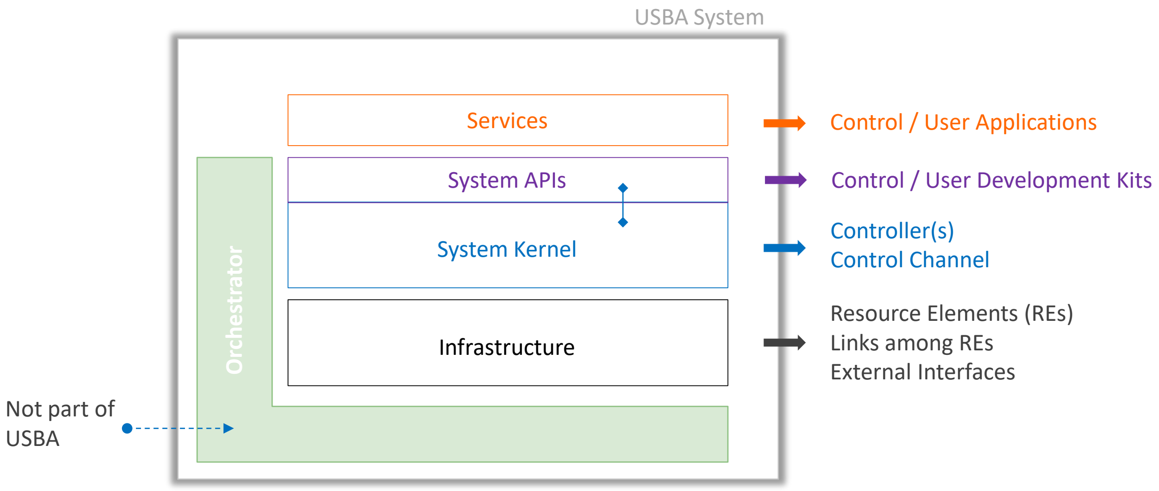


Figure 6: USBA Layers

- **Infrastructure layer:** in USBA, the infrastructure layer consists of "Resource Elements" (RE) (1) (pairwise) interconnected by Links (2). Besides, there are External Interfaces (3), connecting some REs to non-USBA elements (i.e., outside of the system). This is shown in Figure 7.
    - All REs in USBA are network nodes.
    - A Link in USBA is some form of connection between REs.
    - External Interfaces are some form of connectivity of REs with or to elements outside of the considered USBA system instance.

  The infrastructure layer of USBA (and notably the RE / Link representations) is not an actual part of the system, but rather a resource / hardware abstraction that USBA uses to unify all different resources.

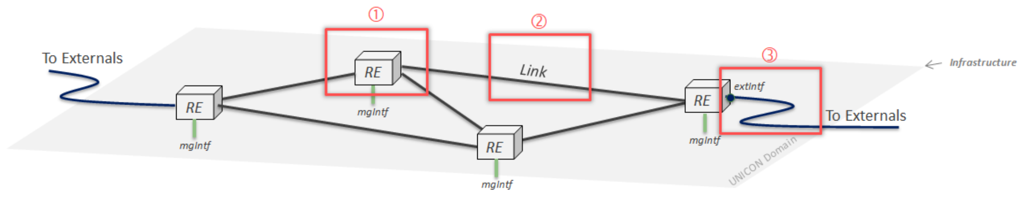


Figure 7: Infrastructure Layer of USBA

- **System Kernel layer:** the main part of USBA is a distributed resource control system, consisting of "Resource Control Agents" (RCA), control channels between some of them and a set of distributed protocols. Herein, both RCAs and Control Channels are always using resources of the Infrastructure layer (REs and Links). In the simplest view, each RCA runs on an RE, and Control Channels are paths between RCAs consisting of at least one Link and an arbitrary number of REs. The System Kernel of USBA is self-organizing, which means that the distributed algorithms are mostly running without any human intervention.
    - An RCA is the main component of USBA and has two "faces". On the one hand, an RCA is a handle to a considered resource element and its

configuration. Hence, RCA can map incoming requests to the local resources. On the other hand, an RCA is a peer in the USBA system, involved in the self-organization of the system and contributing to the basic USBA services in a distributed manner. Hence, it maintains some local awareness (neighbours, etc.) and uses this knowledge in the system-wide distributed protocols.

  - The required distributed protocols include a resource control plane routing and a distributed scheduling. In addition, RCAs can span a distributed object store.

- **System API layer:** this layer provides a uniform system API to the individual resources and also to the basic USBA system capabilities to any consuming application/service. It notably provides different families of API calls, which will be further described below.
- **Services layer:** the USBA service layer distinguishes resource control services and resource usage services. The former are resource control applications, which can alter the way or formulate rules on how to best use the resources, e.g. for particular hosted usage services. These "usage services" are essentially microservices, i.e., typically service function components of the mobile system.

3.1.3. USBA and its capabilities/services

The Resource Usage services of the USBA Service Layer essentially constitute the user applications. In the particular FUDGE-5G case, these are service functions and service elements of the mobile system generating requests to and loads at the resources. In the simplest case, a mobile system service function (e.g. a core network NF) could be directly considered a resource usage service of USBA, executed in one USBA RE as a process. However, other deployment/integration scenarios are supported: the main difference from the resource usage services in USBA and the actual mobile system functions (e.g. 5G NFs) is that, in addition to atomic modules, mobile system functions in USBA can also be implemented as compositions of several basic resource usage services or as distributed interconnections of many resource usage service instances.

USBA provides basic support for these Usage Services, both explicitly and implicitly. Explicit support requires the use of the particular USBA calls. Implicit support can be implemented by formulating per app policies, constraints and optimizations (e.g. in form of traffic selectors, request treatment rules, service budgets, etc.). These rules can be formulated "manually" by the system administration (as a configuration). In USBA, they can be autonomously upheld by dedicated resource control services for different "usage scenarios".

These resource control services or apps can be programmed and deployed on the system along with the "unsuspecting" user apps. The resource control services hence are part of the autonomous system understanding and can be used to enrich or alter the USBA system so as to provide additional system-wide or dedicated capabilities, extending or customizing the basic capabilities of the USBA for particular service functions.

To simplify the development of the service level apps, at its System API layer, USBA offers four basic API families:

- SF to SF communication calls
- Job scheduling calls
- Flow scheduling calls
- Storage placement calls

Using these calls, it is possible to provision both SFs, their shared data and communications between different SFs in a coordinated manner and to keep track of all allocated / executed items throughout the runtime. By defining scheduling policies, system-wide prioritization and optimization can be achieved directly by the existing USBA means. Additional Resource Control Services could introduce further coordination, e.g. network garbage collection or resource conflict resolution means.

USBA autonomically spans a distributed system over all the resources in the resource pool. To achieve this, USBA uses highly scalable distributed mechanisms, notably for routing and load balancing. Therefore, USBA per se yields a flat, non-hierarchical system, with vertical hooks into the considered resources. Since these vertical hooks can reach REs at any level of realization, USBA achieves unification of resource control: it represents all controllable resources albeit at different levels and in different authoritative domains and of different natures, as one flat system with a common API, readily available for control and usage.

Note however that USBA as architecture is not limited to flat systems, because USBA readily supports recursion: if required (e.g., if the authorizations to directly control resources in some pool cannot be obtained), a whole distributed resource pool managed by an USBA instance 1 could be represented as, e.g., a single controllable RE in an USBA Instance 2, using, e.g., a particular resource control application in that same instance 1. Hence, USBA is capable of building hierarchical systems as well. However, USBA is not limited to hierarchies as a means for scalability improvement.

3.1.4. USBA and derived resource requirements

The main abstractions for resource description used in USBA are REs and Links.

USBA Link is a network link or a logical connection, a tunnel or an inner-host virtual link between two REs. The active control of any Link requires an RCA, which can be exploiting any available network control point. Since a link is attached to a RE, ideally, a Link is controlled by RCAs of the connected REs.

RE represents any network resource with some forwarding, computing or storage capabilities. In practice, a RE could be a switch, a router, a leased line, a VPN connection, a server, a rack, a VM, a container, a network device, a NAS, a whole DC or all resources of a global company. USBA poses no requirements as to the size, addressing, position, form, stability, connectivity or resource posture of a RE. It could be distributed or atomic, virtual or physical, on a service layer or deep in the infrastructure. The only requirement is that any RE be network controllable (directly or indirectly). Hence, any given RE can be quite limited or very powerful in each individual dimension, but it must be “active” to be visible and controllable. Since resource control is holistically considered part of the overall system, although a given RE could refuse any (further) allocations on any of these dimensions, none of these dimensions can have a zero capacity for an unused RE.

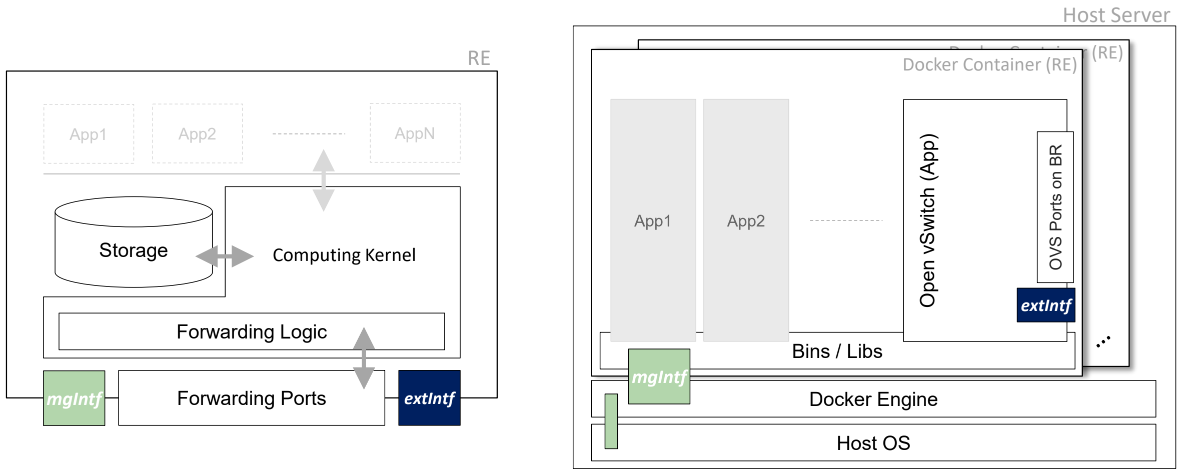
Ideally REs would onboard native USBA RCAs, because otherwise the corresponding RCAs need to be deployed elsewhere and might need to use adapters and wrappers; the usage of RCAs as an active agent is a powerful means to achieve unification, still wrappers and adapters can incur additional limitations and delays. Deployment of RCAs is usually not a problem for virtual elements.

3.2. Formal USBA presentation

3.2.1. General resource stub

Figure 8 depicts a general resource element stub. The left side shows its logical architecture, and the right side a possible implementation architecture based on the Docker virtualization engine. The high level logical architecture resembles the contemporary computer architectures. It has network access hardware such as network interfaces, forwarding logic (which can be implemented either as a hardware unit, e.g. flow tables, or a software unit, e.g. as part of software switch module). It contains a storage unit (e.g. external hard disks or a storage-area network) and a general purpose compute unit, which together provide a general computing context for a set of possibly diverse applications, running on top.

It is critical to understand that this is indeed a logical architecture, physical realizations of which can actually range from an OpenVSwitch, over a single PC to a whole data centre. All of those should be in our opinion viewed from the same angle, as resources which can do a (set of) specific task(s). That is the unified view we are insisting on throughout this deliverable.



Logical Architecture *Implementation Architecture*

Figure 8: General resource element stub

Common to all resource elements is that they have an owner and they are managed by that owner. Again, be it a single router, a server, virtual machine, or a whole data centre, there is an owner who can configure the resource. The owner can, for example, connect it to another resource he or she owns.

The critical part of resource element is what we call Resource Control Agent (RCA), cf. Figure 9. As indicated in the figure, it spans the forwarding logic and the computing kernel of the resource elements, but also interacts with the applications by providing appropriate API calls, as we will see shortly.

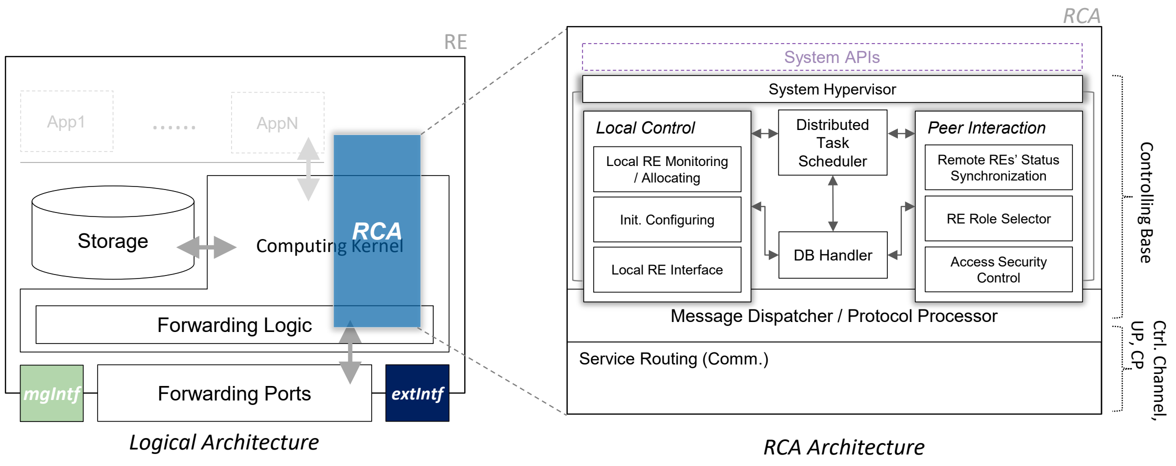


Figure 9: Resource Control Agent (RCA)

Essentially, RCAs in all REs collaborate and create a distributed system that represents what we call System Kernel. The right side of Figure 9 gives the RCA architecture. We'll now describe the main modules in the RCA architecture.

The bottom part in the RCA architecture stack is named Service Routing. That is a protocol (in fact a protocol suite, as will be seen shortly in Section 3.2.4) that enables service invocation protocols among the REs to realize the peer interaction patterns shown in Figure 9. One might be tempted to think of typical OSI layer 3 protocols here plus name resolution systems like the DNS, but they are in our opinion highly inappropriate for this type of task. The missing features are automation, robustness to failures, scalability, etc. In other words, we need way more scalable, autonomous (configuration-free) and robust protocols to enable any-to-any communication among the REs. As indicated on the right side of the figure, this connectivity may be used as control channel (in a controller – controlee deployment) and manual configurations of this (typically out-of-band) control channel, as is the case in current control models, is highly inappropriate. The result of this control plane setup is a network as shown in Figure 10, in which every RE can communicate with every other RE. This network we call "slice 0", a self-organizing and always available slice, which makes the basis of our system kernel.

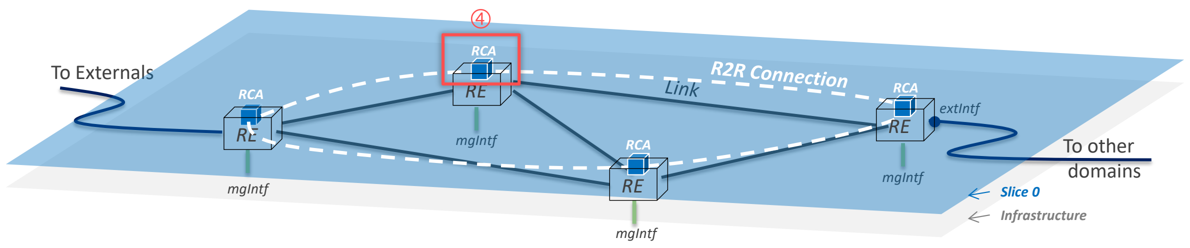


Figure 10: Resource-to-Resource Network

Building on the established control plane connectivity, the Peer Interaction module within the RCA integrates a couple of basic functions that serve the purpose of communicating with other RCAs, those residing in other resource elements in the network, to make various important, system level decisions. For example, an RE might be a controller, while other REs are just “normal” REs. That decision is made (or information disseminated) by these modules communicating with each other. It is not necessary that a single RE is a controller, the model presented readily embraces the setups in which a given RE takes over a specific role in a distributed controller, even mixing the controller and controlee roles.

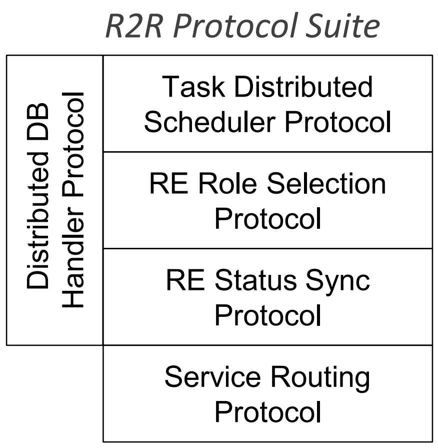

R2R Protocol Suite
Distributed DB Handler Protocol
Task Distributed Scheduler Protocol
RE Role Selection Protocol
RE Status Sync Protocol
Service Routing Protocol

Figure 11: R2R Protocol Suite

From another angle, resource elements can be said to run an R2R protocol suite, which is depicted as a protocol stack in Figure 11.

The Local Control module of RCA controls the local resources – forwarding, storage and computing. It monitors them, performs various configurations and resource allocations, etc.

The Distributed Task Scheduler module, taking the central part in the RCA architecture from Figure 9, is of particular importance for us in this project. This component has been designed, developed and tested in this project, within a particular RCA context, that of Service Router namely. The scheduler is responsible for shrinking the set of locations where a specific task can be executed to a set of location where it can be executed within particular constraints (e.g. time). Note that the scheduler is just an RCA module, such that the schedulers in all RCA taken together constitute a distributed scheduler, i.e. cooperate to make appropriate scheduling decisions.

3.2.2. Databases and repositories

RCA architecture, shown in Figure 9, depicts also a DB Handler. Normally, the storage units presented in REs (cf. Figure 8) are just raw storage. The DB handlers within RCAs operate at higher semantic levels and are supposed to manage these storage units such that they together form a distributed database which provide access to all information the applications and various RCA modules happen to need.

For example, if RCAs have flat IDs and control plane routing is based on these IDs (routing on flat labels), then a side result of this control plane connectivity is a distributed hash table (DHT), a distributed storage system which enables finding data based on their IDs, such as for example their hashed values. If that is the case, local DB handlers are supposed to resolve DB queries by comparing the data ID in the query and the ID of local RCA and then make decision whether to access the local storage to fetch the data (in case of a match) or to forward the query to a next hop otherwise, i.e. forward the query to the hop whose ID is closer to the queried ID.

Certain (complex) queries will require aggregation or fusion of data from multiple REs, i.e. involvement of multiple RCAs. In the said DHT example, a range query (all IDs in a certain ID range) may span multiple RCAs. Coordination among them is needed in that case. For instance, the RCA closest to the lower bound in the specified range may take over the coordination and recursively compose the final reply to the querying application.

3.2.3. Interface descriptor

Figure 12 presents RCA internal architecture, emphasizing this time the system level API calls, which are available to the applications (e.g., network functions, micro-services, etc.) There are four families of APIs. These are:

- Job scheduling calls
- Flow scheduling calls
- App to App communication calls
- Storage placement calls

The two lines connecting the applications and the local control module, respectively, the remote RCA, are supposed to emphasize the distributed nature of the APIs and the entire system. Some API calls can possibly be satisfied by the local resource, while in reality majority of them will require help of remote resources, i.e. their RCAs. For example, executing a complex data query (e.g. querying for a range of data IDs) will require coordination of the local resource and a set of remote ones. Scheduling a job will typically involve multiple resources, installing a flow is its nature a task that spans multiple resource elements, etc. This local-remote coordination, i.e. distributed operation is illustrated also in Figure 12.

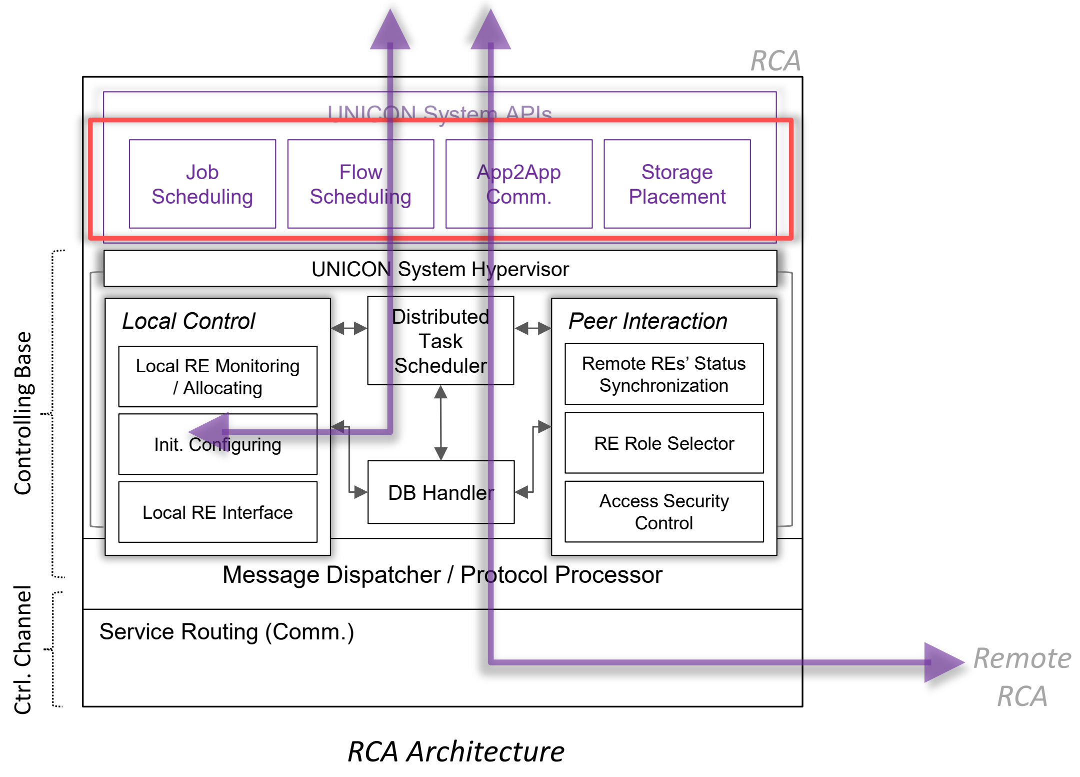


Figure 12: RCA Architecture and API Calls

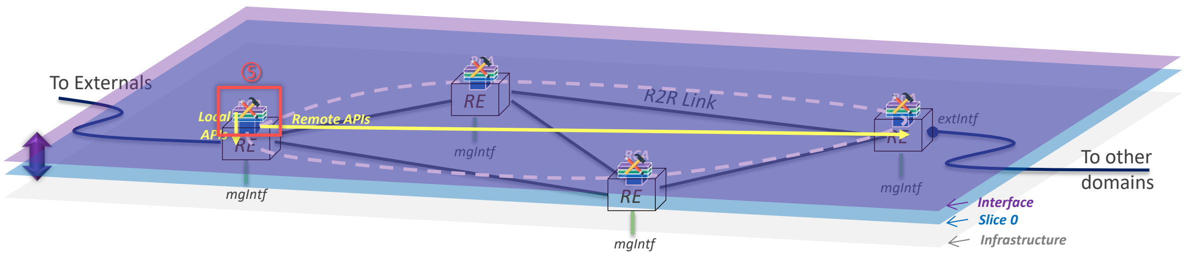


Figure 13: Distributed SDK Platform

In addition, Figure 13 illustrates that the system operates as a distributed SDK platform, which is transparent to the upper layers (i.e. the distribution of the platform) and provides them high-level atomic operations, including both control level and service level ones. All, or at least most of the functional modules of RCA are used for realization of this SDK platform.

To illustrate the provided APIs, it is best we walk through a number of concrete examples. To start, `create_job(name, job_descriptor, RE_ID)` creates a job with a given name at a specified RE, where the job is fully described (including the code to run, or its location) by the provide descriptor. This call is normally not directly called by applications. Assume that an application is a network function, which is about to make a call to another network function (how this call exactly looks like is not important at this moment). This call is trapped by the system kernel, which extracts the parameters of the call and, in collaboration with the scheduler, creates a call to a remote resource where the requested NF will be run. A similar scenario would happen in case of communication of micro-services within one network function, etc. Once the `create_job` returns, the parameter of the original call are forwarded to the destination RE_ID for execution. The entire process of creation of a new job should be transparent to the applications above the system kernel.

`create_SFC(name, job_descriptors[], RE_IDs[])` is a call to create a service function chain consisting of an array of job descriptors and RE IDs where these jobs should run. Normally, it is a sequence of `create_job` calls (it can be, in fact, executed in parallel) followed by setting up the state in the concerned resource elements that enables forwarding of data through the chain.

Jobs and SFCs are terminated by appropriate `terminate` calls with the parameters that match job IDs, which are normally returned by the job (or SFC) creation calls.

To better understand the APIs related to the job creation and termination, one can think of similar calls in Apache Spark[1] or Databricks[2]. The main difference to what we present here lies in the realization of scheduling (e.g. we deal with a general network topology, as opposed to data centre, normally assumed "controlled" as opposed to "managed" environment, etc.) rather than the offered APIs.

Storage handling APIs should at the minimum provide calls to put and get data items. For example `put(data_item), replication=1)` permits storing a data item in the distributed data storage. This basic calls can be further extended with various filters. For example, one can specify a desired replication factor (at how many different nodes the item will be stored) or in appropriate way limit the set of nodes where the item can be stored (e.g. distance from the caller, capabilities of the storage node, certain policies, etc.).

Similarly, there is a `get(data_item_ID)` call which returns either the location, a reader, or similar, of the data item with the given ID. As in case of a put, various extensions and refinements of this basic data retrieval operation are possible. In particular, it is critical to support more complex queries, such as range queries, or filter the queried data based on more complex set of attributes rather than just an ID.

The final piece in our unified architecture are services, which run on our resource elements in the application space. We distinguish between network control services and network data services. This is shown in Figure 14 (cApp and dApp, respectively).

[1] https://spark.apache.org/

[2] https://databricks.com/

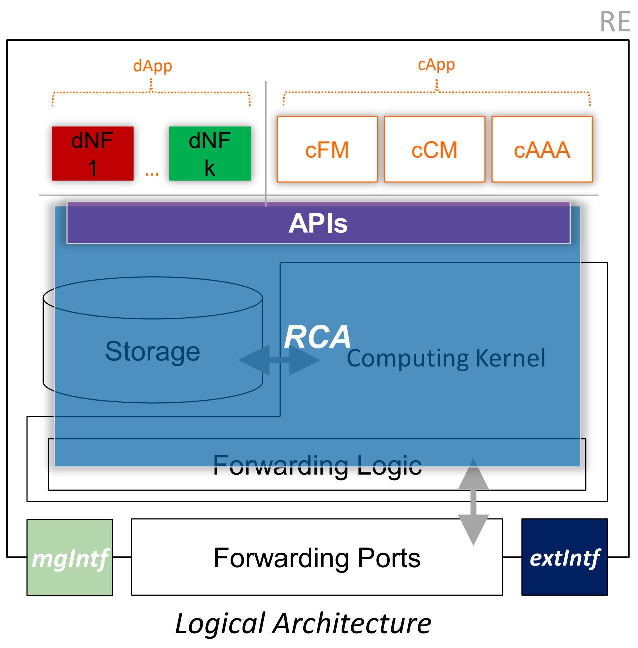


Figure 14: RCA and Network Services

3.2.4. Service routing

As can be seen in Figure 12, service routing forms a key part of the USBA in that it flexibly distributes service invocation among distributed network locations. In the following, we outline key aspect for this service routing that shape the design for efficient realizations of it.

3.2.4.1. Key aspects of flexible service routing in USBA

We can observe from the evolution of network functions to micro-services, as also discussed in the previous sections, that there are a number of key aspects to consider when designing solutions for flexible service routing:

- **Services may be distributed**: While the point-of-presence (PoP) of the current Internet is prevalent in provisioning services, following the cloud model, many of the envisioned beyond 5G use cases result in a significantly larger distribution of services in different network locations. Edge sites with limited capabilities will be increasingly utilized to provide services to clients.
- **Service workload may fluctuate**: The distribution of services across often smaller edge sites also leads to a higher fluctuation of workload, where sites may become overloaded quickly and ‘closer’ sites (in terms of network topology) may not be the right choice in a routing decision from a client to a suitable service instance.
- **Service selection is highly service-specific**: Selecting one of possible many service instances is not just a matter of “one size fits all” criteria. The possible diversity of deployment aspects, such as hardware differences in edge sites, software capabilities etc. leads to often highly service-specific aspects for choosing one instance over another.

- **Service selection must adhere to transaction semantics:** When thinking of traffic steering, there is no relation between the typical ‘unit’ of decision making, namely a packet, and the transaction semantic that drives a relation between a client and a service instance. The latter is captured in a service transaction that is entirely dependent on the application semantic when it comes to length (in time) and number of packets created during the transaction. When performing traffic steering decisions, this affinity to a selected service instance is crucial to maintain to avoid unnecessary costs imposed on the service infrastructure for maintaining state information across possibly distributed network locations.
- **Services may be highly dynamic in their selection criteria:** When assigning requests to specific service instance, we can observe a possibly high dynamicity when doing so. This may either stem from mobility, e.g., of client moving, or from workloads changing, as discussed above. Latency here is crucial, where the budget for transaction latencies can often not accommodate much latencies stemming from communication. This begs the question how suitable current models of traffic steering through indirection points, such as the DNS or load balancers or similar, are; an aspect we discuss below in Section 3.2.4.4.

3.2.4.2. Design goals

Before presenting our realization of service routing capability in USBA, let us outline the main design goals:

- **Enable service-specific traffic steering**: Constraints for routing traffic should not be limited to network metrics like delay or bandwidth. Instead, services should be able to define multiple service-specific constraints, either realized through a *multi-optimality routing* solution or through a more direct, request-level and possibly compute-aware *request scheduling* method for selecting one of possibly several service endpoints [14][15][16]. The expected benefit is to reduce service completion time, while reducing messaging overhead.
- **On-path traffic steering to reduce service latencies**: Existing solutions, most notably the DNS resolution process with subsequent IP routing, relies on indirection points for the traffic steering decision. This adds significant latency to the initial request but also when wanting to re-assign a client to another service instance, which requires repeating those indirection steps more frequently. Instead, the goal for service routing is to perform those ‘resolution steps’, i.e. the assignment of a service to a specific service instance, on-path from the client to the selected service instance.
- **Support short-lived and long-running interactions**: The duration of service interactions with a selected service endpoint depends on the specific service. Any solution must support short-lived as well as long-running interactions with any (dynamically) selected service endpoint.
- **Bound routing tables**: Insights from previous work in information-centric networking (ICN) have shown the strain on routing tables for similar routing approaches. Any service routing solution must improve on this, reflected in its domain-local service routing capability, supported by its IP-based interconnection to other service routing enabled domains.

- **Preserve communication confidentiality**: Due to the often sensitive nature of service interactions, *routing on service addresses* (ROSA, to be explained in Section 3.3.1) must support communication confidentiality, while also allowing for single packet flows through an optimized security handshake.
- **Access beyond single domain**: Although we see USBA being initially deployed in a single domain, access to services in remote USBA domains and the Internet should still be supported.
- **Coexistence with IP routing**: Service routing does not replace IP routing, but actively uses its extension capability in the form of extension headers, while also foreseeing services still being exposed via existing DNS-based methods.

3.2.4.3. Expected benefits of service routing

The benefits of service routing are as follows:

- Indirection through the DNS is not used anymore, leading to faster initial request completion and faster indirection.
- On-path traffic steering can realize scheduling decisions that improve latencies and system evaluations compared to off-path solutions such as the DNS, as evaluated in Section 4.1.
- Service addresses can be described as so-called structure binary names, enabling secure validation of service announcements and therefore providing a secure alternative to the administrative domain name assignment of the DNS. Further, secure delegation is supported when chaining keys used to secure service announcements. This leads to appealing security capabilities at this low level of the system.
- Service relations are inherently multicast by nature, i.e. service selections may not just select a single service instance but a group of instances, including randomized groups for diffusion purposes. This may be an appealing capability for services such as Distributed Ledger Technologies that rely on unicast-based diffusion solutions today, leading to significant overhead created by them.

3.2.4.4. What is wrong with off-path solutions?

Standard DNS resolutions in the public Internet experience latencies of 15 to 45ms per resolution (for well-known domain names, i.e. those that can be resolved by the first hop DNS server), while additional load balancing in solutions such as Global Server Load Balancing, used in most CDN systems, require often up to 100ms latency for an assignment of a suitable service instance. These latencies are caused not just by the additional signalling (which involves crosses the wireless access link from the client to the network ingress at least four times, often more) but also the application layer processing of most of the required functionality, such as the DNS database lookup for resolution or the HTTP-level indirection for load balancing. Going through these processes frequently is prohibitive since it would inevitably impact service completion time but also increase complexity of the overall system.

On-path resolution, as suggested by service routing, instead performs the necessary mapping from service addresses to IP locator on-path, i.e., when the service request traverses from the client over the ingress to the selected service instance. When implemented at, e.g., Layer 3 or Layer 3.5 (as IP extension header), the processing is significantly faster, even in SW-based solutions with Linux fast path operations, while data operations are limited to routing table lookups with additional selection algorithms in case of more than one service instances being available. The evaluation in Section 4.1 shows the impact of changing from off- to on-path in terms of latency and system utilization.

3.3. USBA Realization – Scheduling Aspects

3.3.1. Service routing as a Routing on Service Addresses (ROSA) shim layer

In the following, we outline key aspects of realizing service routing through an approach we call *routing on service addresses* (ROSA), where service information is directly used on-path, as outlined before, to steer traffic to the appropriate service instance in one of possibly many network locations. In the following, we focus our presentation on the general system overview, followed by the new message types introduced, and the forwarding operations performed on-path, concluded with client changes that are required to support the novel service routing capability. We will here focus on scheduling-based traffic steering methods, presented in Section 3.3.2, while ROSA also supports a fully routed traffic steering, which is left out from this deliverable.

3.3.1.1. System overview

Figure 15 illustrates a ROSA-enabled limited domain [3], interconnected to other ROSA-supporting domains via the public Internet through the *Service Address Gateway* (SAG).

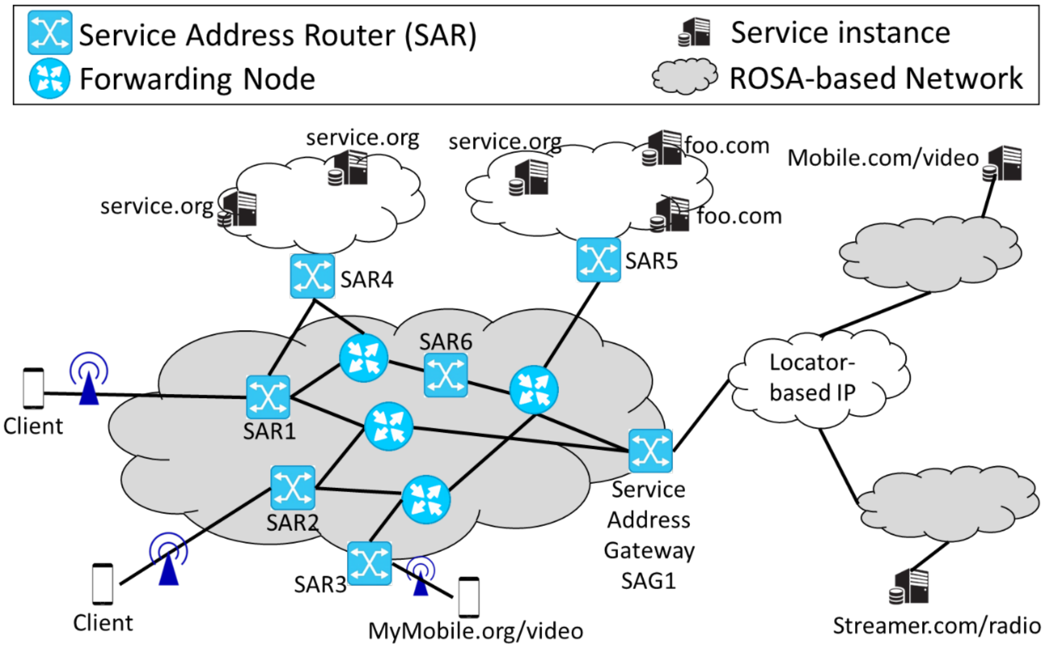


Figure 15: Service Routing System Overview

ROSA endpoints start with discovering their ingress *Service Address Router* (SAR), e.g., through extensions to DHCP or similar approaches. Furthermore, we also foresee that an endpoint may discover different ingress SARs for different categories of services, each SAR being part of, e.g., a category-specific ROSA overlay, which in turn may be governed by different routing policies and differ in deployment (size and capacity). Within the original SBA framework, the SAR is closest in functionality to the SCP while performing an on-path discovery of the suitable endpoint on-path rather than through the off-path NRF function in SBA.

Services are realized by *service instances*, possibly at different network locations. Those instances expose their availability to serve requests through announcing the *service address* of their service to their ingress SAR.

To invoke a service, a ROSA client sends an initial request, addressed to a service, to its ingress SAR, which in turn steers the request (possibly via other SARs) to one of possibly many service instances; see Section 3.3.1.3 for SAR-local forwarding operations and end-to-end message exchange and Section 3.3.1.4 for the needed changes to ROSA clients.

We refer to initial requests as *service requests*, which are routed via the ROSA network. If an overall *service transaction* creates ephemeral state, the client may send additional requests to the service instance chosen in the service request; we refer to those as *affinity requests*. With this, routing service requests can be positioned as *in-band service discovery*, resulting in subsequent routing of direct client-service instance traffic. In order to support transactions across different service instances, e.g., within a single DC, a *sessionID* may be used, as suggested in [7]. Unlike [7], discovery does not include mapping abstract service classes onto specific service addresses, avoiding semantic knowledge to exist in the ROSA shim layer for doing so. The next two sections detail message types and exchanges to realize this behaviour.

3.3.1.2. Message types

Apart from affinity requests, which utilize standard IPv6 packet exchange between the client and the service instance selected through the initial service request, ROSA introduces three new message types, using an IPv6 extension header (EH) [4] for extra information required for ROSA functionalities. The messages Figure 16 highlight only the entries needed for the specific purpose of the message, omitting other IPv6 packet header information for simplicity. An initial prototype, developed by Huawei, uses a TLV format for the extension header with Concise Binary Object Representation (CBOR) [6] being studied as an alternative. The EH entries shown are populated at the client and service instance, while read at traversing SARs.

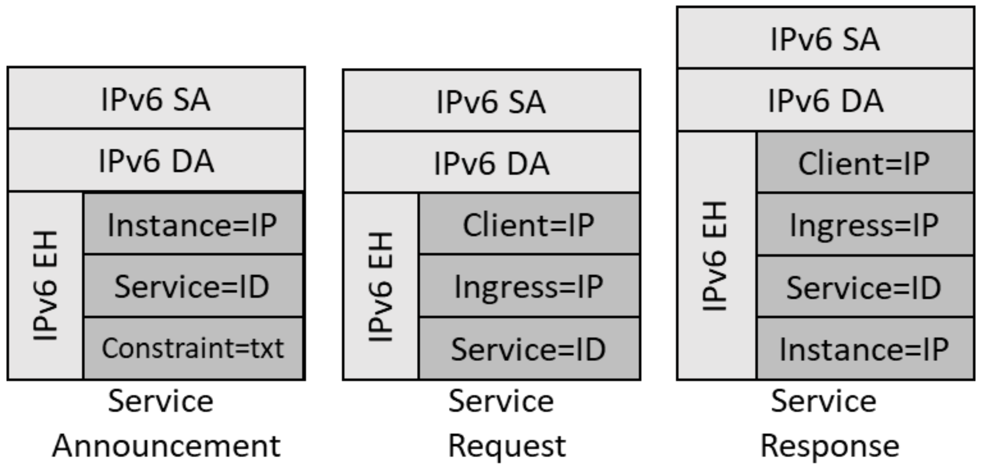


Figure 16: Service Routing Message Types

A service address may be encoded through a hierarchical naming scheme, e.g., using the naming conventions in [5]. Here, service addresses consist of *components*, allowing to map existing hierarchies of services addresses in the Internet onto those over which to forward packets, illustrated in the forwarding information base (FIB) of Figure 17 as purely illustrative URLs. Components are treated as binary objects, while the hierarchical structure allows for grouping addresses along common prefixes, reducing routing table size and forwarding lookup times. Despite the variable length of service addresses, their explicit structure (in the form of components) allows for efficient hash-based lookup during forwarding operations, unlike IP addresses which require either log(n) radix tree search software or expensive TCAM hardware solutions.

With the *service announcement* message, a service instance signals its ability to serve requests for a specific service address. Section 3.3.2 outlines the use of this message in scheduling-based traffic steering methods.

The *service request* is originally sent by a client to its ingress SAR, which in turn uses the service address provided in the extension header to forward the request, while the selected service instance provides its own IP locator as an additional extension header entry in the service response. The next section describes the SAR-local forwarding operations and the end-to-end message exchange that uses the extension header information for traversing the ROSA network.

3.3.1.3. SAR forwarding engine

The SAR operations are typical for an EH-based IPv6 forwarding node: an incoming service request or response is delivered to the SAR forwarding engine, *parsing* the EH to obtain relevant information for the forwarding decision, followed by a *lookup* on previously announced service addresses, and ending with the forwarding *action*.

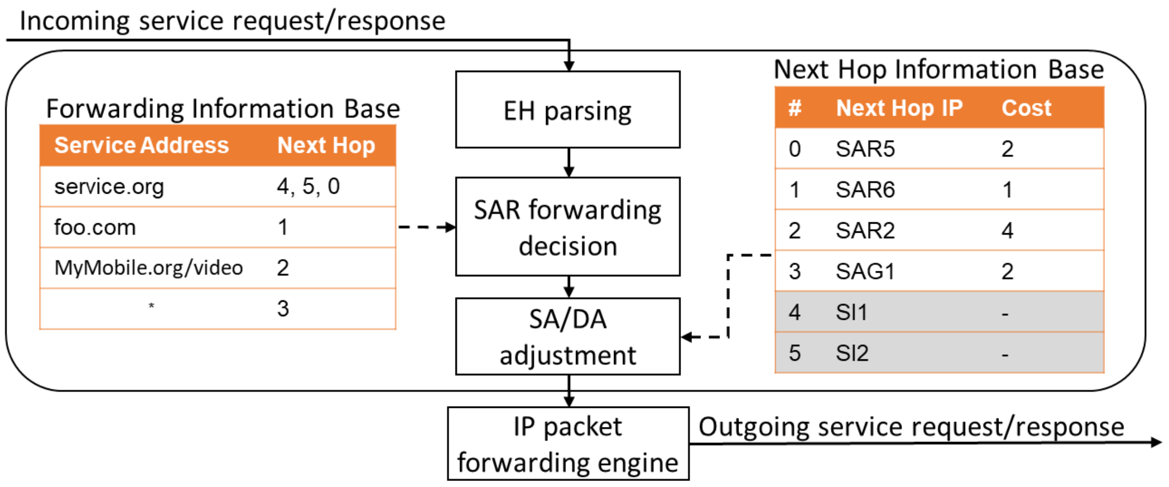


Figure 17: SAR Forwarding Engine

Figure 17 shows a schematic overview of the forwarding engine with the *forwarding information base* (FIB) and the *next hop information base* (NHIB) as main data structures. The NHIB is managed through a routing protocol, with entries leading to announced services.

The FIB is dynamically populated by service announcements, with the FIB including only one entry into the NHIB when using routing-based methods (rows 0 to 3 in Figure 17). Scheduling-based solutions (see Section 3.3.2), however, may yield several dynamically created entries into the NHIB (items 0, 4 and 5 in Figure 17) as well as additional information needed for the scheduling decision; those dynamic NHIB entries directly identify service instances locations (or their egress as in item 0) and only exist at ingress SARs towards ROSA clients.

For a service request, a longest-match service address lookup (using the *Service* EH entry) is performed, leading to next hop (NH) information for the IPv6 destination address to forward to (the final destination address at the last hop SAR will be the instance serving the service request).

Forwarding the response utilizes the *Client* and *Ingress* EH fields, where the latter is used by the service instance's ingress SAR to forward the response to the client ingress SAR, while the former is used to eventually deliver the response to the client by the client's ingress SAR, ensuring proper firewall traversal of the response back to the client. Our Huawei prototype shows that the operations in Figure 17 can be performed using eBPF [8] extensions to Linux SW routers.

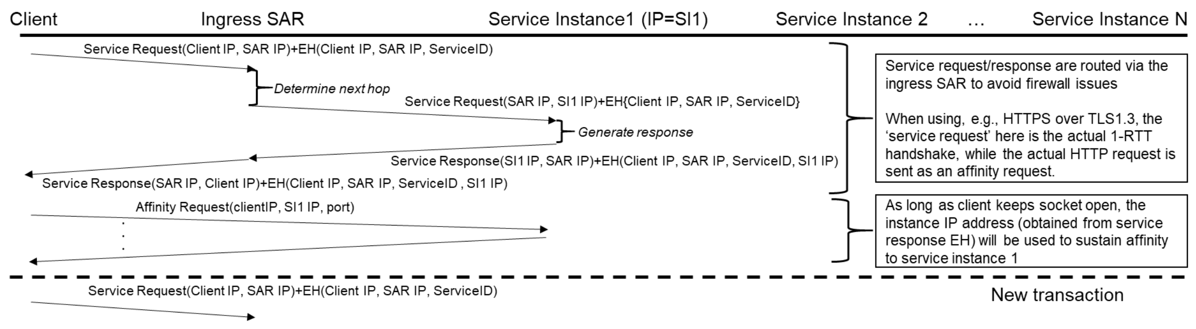

Client
Ingress SAR
Service Instance1 (IP=SI1)
Service Instance 2
…
Service Instance N
Service Request(Client IP, SAR IP)+EH(Client IP, SAR IP, ServiceID)
Determine next hop
Service Request(SAR IP, SI1 IP)+EH{Client IP, SAR IP, ServiceID}
Generate response
Service Response(SI1 IP, SAR IP)+EH(Client IP, SAR IP, ServiceID, SI1 IP)
Service Response(SAR IP, Client IP)+EH(Client IP, SAR IP, ServiceID , SI1 IP)
Affinity Request(clientIP, SI1 IP, port)
Service request/response are routed via the ingress SAR to avoid firewall issues
When using, e.g., HTTPS over TLS1.3, the 'service request' here is the actual 1-RTT handshake, while the actual HTTP request is sent as an affinity request.
As long as client keeps socket open, the instance IP address (obtained from service response EH) will be used to sustain affinity to service instance 1
Service Request(Client IP, SAR IP)+EH(Client IP, SAR IP, ServiceID)
New transaction

Figure 18: Message exchange to route service request and subsequent affinity requests

Figure 18 shows the resulting end-to-end message exchange, using the aforementioned SAR-local forwarding decisions. We can recognize two key aspects. First, the SA/DA re-writing happens at the SARs of the ROSA shim layer, using the EH-provided information on service address, initial ingress SAR and client IP locators, as described above. Second, the selection of the service instance is signalled back to the client through the additional *Instance* EH field, which is used for directly sending subsequent (affinity) requests via the IPv6 network. As noted in the figure, when using transport layer security, the service request and response will be those related to the security handshake, thereby being rather small in size, while the likely larger HTTP transaction is sent in subsequent affinity requests.

3.3.1.4. Changes to clients to support ROSA

Within endpoints, the ROSA functionality is realized as a shim layer atop IPv6 and below transport protocols. For this, endpoints need the following adjustments to support ROSA:

(i) **Adapting network layer interface**: Introducing service addresses requires changes to the current socket interface for discovering the ingress SAR and issuing service requests as well as maintaining affinity to a particular service instance, i.e. mapping a service instance IP address to the initial service address. This could be achieved through providing a new address type (e.g., ADDR_SA) during socket creation, assigning the service address to the returned handle, while utilizing socket options to assign constraints to receiving sockets, utilized in the announcement of the service address.

(ii) **Transport protocol integration**: We see our design aligned with existing transport protocols, like TCP or QUIC, albeit with changes required to utilize the aforementioned new address type. For the application (protocol), the opening and closing of a transport connection would then signal the affinity to a specific instance, where the semantic of the `connection' changes from an IP locator to a service address associated to that specific service instance. With this, a new service transaction is started, akin to a fresh DNS resolution with IP-level exchange.

(iii) **Changes to application protocols**: The most notable change for application protocols, like HTTP, would be in bypassing the DNS for resolving service names, using instead the aforementioned different (service) socket type. These adaptions are, however, entirely internal to the protocol implementation. Given the ROSA deployment alongside existing IP protocols, those changes to clients can happen gradually or driven through (e.g., edge SW) platforms.

3.3.2. Runtime scheduler

Service request scheduling refers to the selection of the 'best' service instance, within the set of active service instances, to serve a service request at runtime of the overall system. Referring to the system model outlined in Section 3.3.1.1 and illustrated in Figure 15, this scheduling decision is performed only at the ingress SARs, which receive the service requests from the clients, while the remaining SARs illustrated act as forwarding nodes. The scheduling decision is interpreted as a service request routing problem at the data plane level.

The implemented and evaluated runtime scheduler is one that takes the computing capabilities of the service instances in the network into consideration for the scheduling decision. It is therefore referred to as the **C**ompute-**A**wa**r**e **D**istributed **S**cheduler, or CArDS. The objective of CArDS is to maximize the system's processing throughput by minimizing the (service) request completion time (as the sum of the delays at SARs and instances, together with network propagation delays) for individual service requests.

The forwarding is realized as a two stage process. First, the ingress SAR determines all outgoing interfaces along which an incoming service request could be sent. It then selects the appropriate interface to be used by implementing the scheduling decisions, which is elaborated in the following paragraphs. In essence, the SAR performs an on-path resolution of the service identifier provided in the service request to (a direction towards) a possible service instance; with this, the SAR has taken over the role of the DNS albeit utilizing the compute awareness in the scheduling decision to forward packets. A forwarding SAR then simply forwards the request to the next hop of a SAR, utilizing suitable encapsulation techniques.

We assume that the service instances for a given service are already deployed in the network. Furthermore, for each server in the distributed sites, we assume a total processing capacity. We furthermore assume that any service instance for service hosted on a server is assigned a compute unit, where the total compute units assigned to all instances hosted on said server do not exceed the total processing capacity. We see the assignment of those instance-specific compute units as part of the overall placement process, providing an input into our scheduling solution. With this, each compute unit represents a normalized processing rate that is the same across all server deployments, while the compute unit defines the share of compute resources that the assigned service instance will receive from its physical server resource.

Key to the compute-awareness of our solution is the mapping of compute units onto suitable routing constraints that can be taken as input during the ingress forwarding decision, i.e. the scheduling decision. This routing constraints are used for scheduling a packet at an ingress semantic router to one of the possible many service instances.

For this, we assume the integration of the compute metric assignment in placement methods and service orchestration operations. In order to turn the compute unit assignments into routing constraints, the service orchestration flattens and joins the service instance-specific compute units into a compute vector for a specific service identifier that represents a set of service instances. The needed information for each service identifier, containing each SI's locator together with the number of compute units allocated per instance, is expressed as lower and upper sub-interval bounds in the compute vector. The reasoning behind the use of the interval-based method in the compute vector is further explained in the scheduling mechanism in the following paragraphs.

The compute vector then needs distribution to the network ingress points to perform suitable scheduling operations together with the respective locator information for each service instance for the given service identifier. Key here is that this vector is seen as being rather stable since it is part of the overall service deployment and placement of service instances. Hence, any change will likely happen infrequently only, if at all during the service lifetime. As a consequence, extensions to existing routing protocols, to distribute the computing vector among all routers, will unlikely cause much additional overhead to the routing protocol performance. As an alternative, a service management system may directly signal the routing information to the ingress semantic routers only.

Once an ingress SAR receives a service request, after checking for a routing table entry for the service identifier provided in the request, the suitable next hop (or SI destination) is selected through a weighted round robin, with the weights being the compute unit for the service instance in the compute vector of the service identifier. The scheduling mechanism is as illustrated in Figure 19:

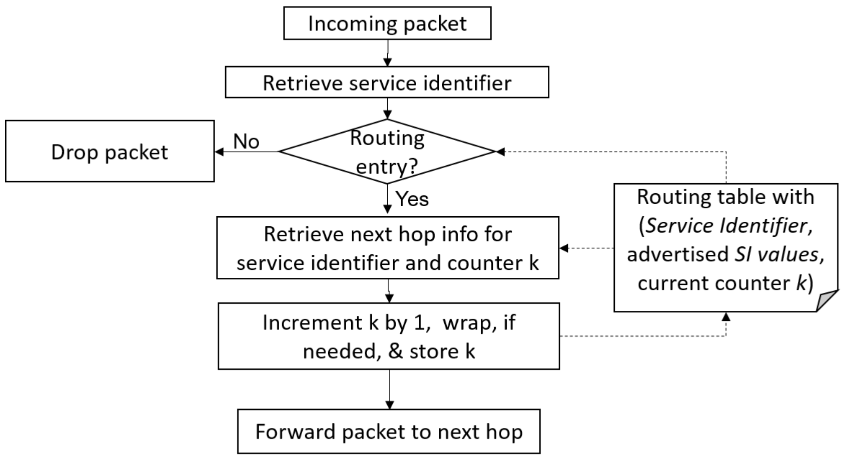

Incoming packet
Retrieve service identifier
Drop packet
No
Routing entry?
Yes
Retrieve next hop info for service identifier and counter k
Routing table with (*Service Identifier*, advertised *SI values*, current counter *k*)
Increment k by 1, wrap, if needed, & store k
Forward packet to next hop

Figure 19: Scheduling decision at ingress SAR

In order to avoid the need for implementing multiplications for the weights (i.e. compute units) at the scheduling decision at ingress SA, we assume that compute units are distributed as sub-intervals instead, with the total interval length being the sum of the compute units (each sub-interval equals one compute unit) of all the available service instances for the service identifier. This flattening of the weights into a vector allows for realizing the weighted round robin through a simpler counter, k, that cycles through that interval for any new service request that arrives at the semantic router. For every new increment of the counter, or wrap-around once the end of the complete interval vector is reached, the scheduling operations retrieve the next hop, i.e. service instance destination information, for the current counter and stores its new value in the routing table to be used for the next arriving request. Each semantic router chooses a random initial value for k, therefore increasing the randomness between individual semantic routers.

The needed scheduling operations are limited to a routing table lookup and a cycling of a counter over an interval (stored as part of the routing table). Technologies such as P4 [2] can be used for realizing such operations at line speed. Using structured binary names for the service identifier in our system allows for utilizing existing longest-prefix match operations to determine the suitable interval in our operations, while increment operations over such interval can be directly realized through P4 operations.

Additionally, crucial in our model is the support for instance affinity, accommodating the likely situation that within a longer transaction (consisting of several service requests), client-specific state is established at the service instance, such as in use cases like online gaming, AR/VR scenarios, or also client-specific transactions in a 5G control plane. As a result, any following request, i.e. the aforementioned affinity request, will need to be sent to the same service instance. While a service request is directed to the service identifier as a destination, the client will utilize the IP locator provided in the response (in addition to the service identifier) to the original service request when addressing any following affinity request to the same service instance. This approach positions the client as being the best point of determining what requests belong to a longer affinity.

However, the realization of this affinity requires support at the client. This could be realized through a dedicated socket type, alongside existing TCP, UDP, or (raw) IP sockets, managing the mapping of an initial service request to subsequent instance requests. For this, the socket implementation utilizes service endpoint information provided in the response to the initial service request, i.e. the usual tuple of source and destination IP addresses and ports, in order to form subsequent instance requests. Application libraries, such as for HTTP, would need to be adapted to use this new socket type rather than, e.g., a TCP socket, while applications based on HTTP would remain unchanged. Approaches relying on application protocol specific proxies (e.g., for HTTP) could also be used, rather than change clients directly.

4. Performance Evaluation

4.1. Runtime Scheduler

The performance evaluation for the compute-aware distributed scheduler (CArDS) [9], the runtime scheduler described in Section 3.3.2, was executed using a custom, event-based Python simulator. The main metric of the evaluation is the mean request completion time (RCT), which in this case is analogous to the service completion time. It refers to the round trip time from issuing a request at the client, processing at a service instance and being received back at the client.

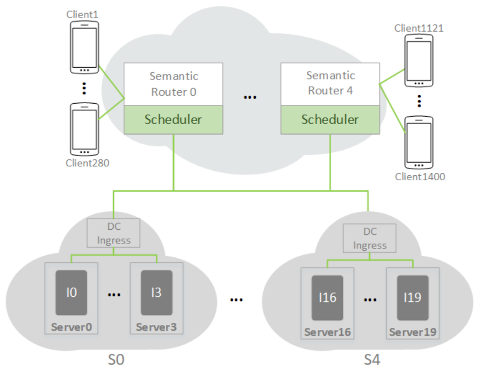


Figure 20 Evaluation network topology

Table 1 Evaluation configuration parameters

Scenario	Parameter	Configuration/Distribution, Value
All	Link Latencies	Inter-Site: exponential with average 3000 µs Intra-Site: exponential with average 700 µs
	Topology	5 Sites, 4 Servers per Site, 1 instance per Server, 5 Ingress Nodes
1,2	Simulation Duration	10s
	Network Traffic	Constant bit rate: inter arrival times selected uniformly from [2.5, 7.5] ms
	Server Processing Time	Uniform, [128, 192] µs
3	Simulation Duration	30 min
	Network Traffic	Variable bit rate: inter arrival times selected uniformly from [1.9, 2.1] s
	Server Processing Time	Uniform, [1.6, 2.4] ms

1,3	Compute Capacity Dist. Across Sites and Instances	S0[1,2,2,3], S1[1,2,2,4], S2[2,2,3,3], S3[2,2,3,4], S4[1,3,4,4]
2a	Compute Capacity Distribution (Uniform both across and within sites)	S0[2,2,3,3], S1[2,2,3,3], S2[2,2,3,3], S3[2,2,3,3], S4[2,2,3,3]
2b	Compute Capacity Distribution (Imbalance across sites, uniform within site)	S0[1,1,1,1], S1[1,1,1,1], S2[1,1,1,1], S3[1,1,1,1], S4[8,8,9,9]
2c	Compute Capacity Distribution (Uniform across sites, imbalance within site)	S0[1,1,1,7], S1[1,1,1,7], S2[1.1.1.7]. S3[1.1.1.17]. S4[1.1.1.7]

The setup and topology are summarized in Figure 20 and Table 1. The scheduler is realized in the ingress SARs within an ingress-based architecture. To simplify the evaluation, the simulations were executed with a single service function sent as single packet requests. Clients send their service requests to their corresponding ingress SAR, which selects the instance to schedule the request to, using the mechanism (with the routing table and counter, k) described in Section 3.3.2. The network load for each simulation is varied using the total number of clients sending service requests. The clients are distributed equally across the ingress SARs. The network load was varied between 20% and 110%, which are equal to 315 and 1710 clients respectively. A 100% workload is represented by 1550 clients, shown as a grey dotted line in the figures. The simulations were repeated to ensure a sufficiently small 95% confidence interval, shown as lighter regions on either side of the mean service completion time graphs in the evaluation figures. Note that cases, where they are not visible, imply that the interval was very small. Additionally, the minimum latency that appears to be zero is in the milliseconds range.

The first set of evaluations were performed to observe the impact of distributing the scheduling decision, i.e. from a centralized scheduler to a distributed one and further scaling up the extent of this distribution. The comparison made against the idealized, centralized scheduler, which is implemented as CArDS with a single central counter, was expected to perform the best by effectively reduce the potential of conflicting scheduling decisions, leading to contention at the instances. As it uses single central counter, it ensures that only one service request is scheduled to a single compute unit of an instance at a time. Based on our results, there was only a discrepancy between the performance of the centralized and distributed schedulers when the load was approaching the maximum capacity, where an increase in the average request completion times of around 11.3% was observed. In all other settings, there were negligible differences in the performance between the distributed scheduler and the idealized, centralized scheduler. In the second part of the evaluation, the number of ingress SARs were scaled to observe the impact of the scale of distribution on the performance of CArDS. For each network load, we checked if that load is better scheduled by exactly one, a small, moderate, etc. up to an extremely large set of schedulers. The previous finding from the comparison with the centralized, idealized scheduler applies here as well, i.e. only extreme loads cause deterioration of distributed scheduling. As the network load approaches capacity, this deterioration is seen to grow with the scheduling distribution scale, e.g. at 100% load compared to the centralized, idealized scheduler, a 29% increase in RCT is observed when the scheduling decision is distributed across a moderate number of ingress SARs, as opposed to a 11% increase for a small number of ingress SARs. In the worst case, for the highest load and an extremely large number of ingress SARs, the service completion times are around 50% higher than with the idealized, centralized scheduler. Considering the extremely large number of schedulers would be highly distributed in terms of network locations, thereby causing significant path stretch when utilizing centralized scheduling instead, an impact on overall latency needs weighing against the observed 50% increase in scheduling latency for distributed scheduling, since the latter allows for avoiding such path stretch latency.

Certain deployment scenarios may not want to expose the service instances directly to network-level routing but use DC-internal mechanisms instead. Our second evaluation scenario takes this account by comparing CArDS original mechanism of scheduling service requests directly to instances using their service identifiers, against a reconfigured CArDS that schedules service requests to a data center (or site) ingress, which directs the requests randomly to one of the instances within the DC. The DC ingress acts as a simple load balancer, unaware of the computing capabilities of the instances, distributing the requests to one of its instances uniformly at random. We found that this lack of compute awareness at the load balancers has a significant impact on the request completion times and this impact increases with an increased network load. With a network load as low as 30%, the mean RCT of scheduling to sites is almost double that of directly scheduling to instances, while at 80% load, this grows to more than 100 times higher. Although the sites receive requests proportional to their compute resources, the compute-unaware load balancers cannot distribute them to the instances according to their capabilities due to their random nature of distribution. Furthermore, the performance of using site-specific load balancers is largely dependent on the network topology, unlike scheduling to instances directly since the latter simply iterates over the compute units of all compute resources irrespective of their distribution across sites. The effect of the network topology on the performance of a scheduler is further observed in Scenario 2 evaluations, illustrated in figures Figure 21, Figure 22, and Figure 23.

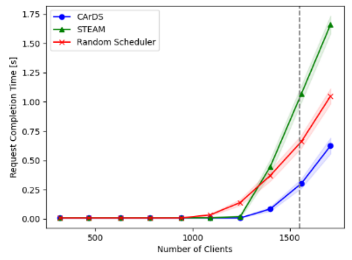


Figure 21 RCT of Scenario 2a

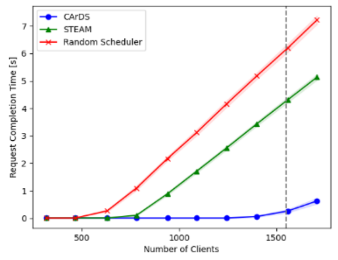


Figure 22 RCT of Scenario 2b

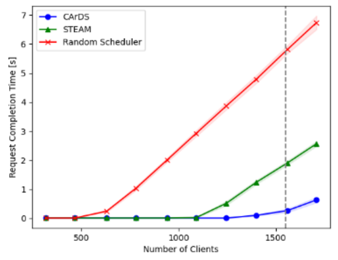


Figure 23 RCT of Scenario 2c

In Scenario 2, the focus was on evaluating the performance of CArDS in comparison with two other distributed, dynamic scheduling approaches, i.e. STEAM and a random scheduling mechanism, which behaves similarly to the previously used simple load balancer, distributing the service requests to the instances uniformly at random except that it is now positioned at the ingress SAR. The scheduling approaches were selected as viable alternatives to CArDS that are easier to implement and do not require additional load information or signalling. STEAM's approach to scheduling uses load estimation and local instance state information to perform batch scheduling at the sites. As its primary focus was for service function chaining applications, the admission control policy module at the site schedulers would be able to forward batches of requests to other sites when the site-local instances were unable to serve them. We disable this admission control part since forwarding to other instances is not supported in the ingress architecture utilized here but rather a capability of the specific service function chaining solution into which STEAM was originally embedded. Additionally, STEAM does not consider the concept of compute units similar to the random scheduler. Also, as the STEAM schedulers are positioned at sites, unlike the other two schedulers, they require the ingress nodes to forward the requests to them, so they can schedule these requests to a local instance. For STEAM's configuration, the ingress SARs simply forward the client service requests uniformly at random to the different sites with large batches of 50 requests being used. The network topology and traffic load is otherwise identical to the previous scenarios.

This scenario allows to observe the effect of factoring compute capabilities into the scheduling decision (as opposed to load estimations), as well as performing the scheduling at ingress nodes instead of sites. We further evaluate the impact of the compute unit distribution across sites as well as instances within sites, on the scheduling performance. For this, we fix the total amount of compute units across all instances to 50, while varying the allotment of those compute units across instances and sites for the different configurations, as specified in Table 1. While CArDS considers compute units irrespective of their distribution in a network, both STEAM and the Random Scheduler are compute-unaware. Although STEAM's scheduling mechanism allows it to avoid contention, it is limited to a site, thereby being unable to influence the requests beyond the site it is deployed at. As a result, the randomness of service request distribution across sites is expected to have some impact on the overall request completion times for STEAM. To reduce this impact, it requires the compute units to be uniformly distributed across sites, which is the case with both Scenarios 2a and 2c. The Random Scheduler, on the other hand, only considers the instances in the network, irrespective of their spread across sites, with no concept of compute units. Considering that it uses the uniform distribution for its random scheduling decision, it is expected to perform well in a network with a balanced distribution of compute units across instances, as in Scenario 2a, and badly when the compute unit distribution is skewed, i.e. with a large variance in the minimum and maximum compute capacities, as is the case with Scenarios 2b and 2c.

In Scenario 2a, where the distribution of compute units are uniform across the sites and the instances within the sites, all three schedulers are expected to perform well. However, as shown in Figure 21, CArDS brings benefits by significantly reducing request completion times in high load settings (i.e. number of clients larger than 1300). We also further observed, in the setting with 1245 clients, the point where the Random Scheduler performance starts to diverge from the rest, the tail is very heavy using Random Scheduler, slightly lighter when using STEAM, while there is no tail when using CArDS. This indicates that not only does CArDS improve on the average performance, but also significantly cuts the tail by distributing the resources fairly among the clients.

In Scenario 2b, where the distribution of compute units is uniform within a site but imbalanced across the sites, STEAM and Random Scheduler are performing very poorly as depicted in Figure 22, while CArDS's performance is unaffected by the imbalance, as it is compute-aware.

In Scenario 2c, with compute units distributed uniformly across sites but an imbalance within sites, we observe that STEAM is able to handle resulting contention within a site, performing similar to 2a, while Random Scheduler provides similar bad performance as in 2b. Again, CArDS outperforms both, providing a much lower request completion time at high loads.

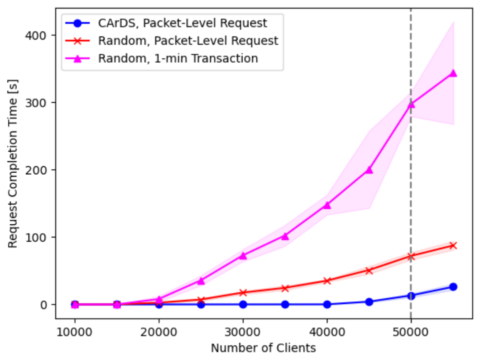


Figure 24 RCT for Scenario 3

To evaluate the performance of CArDS in a typical application that would benefit from improving completion times of individual service requests across more than a single site of service deployment, we considered a content retrieval use case for Scenario 3, which can often be found in localized service scenarios such as those outlined in our introduction. Content here may be video content, gaming assets (such as graphics or video snippets), or also application updates for current mobile applications or future edge applications provided locally within a single operator network.

In those scenarios, several replication sites may be used, while content can often be retrieved via stateless single requests with often larger responses being returned to the client. This is therefore an extreme scenario with scheduling being possibly performed for individual service requests. Evaluating CArDS' performance in such use cases allows for comparing against using long-lived approaches, typically used in application level solutions, such as CDNs, IETF Alto, etc.

For this, we change the traffic pattern to represent real-life video streaming traffic, while keeping the same network topology as specified in Table 1. Server processing was increased to 600 requests per second, which simulates retrieving roughly equally sized video chunks of 2 seconds length (a typical setting in over-the-top video platforms). As the remaining configuration aspects neither impact scheduling decision nor performance of the scheduler, they are not included in the scenario description. Transaction sizes were varied to represent packet-level requests and long-lived transactions of 1 minute. Clients join the system at different times, to simulate a more dynamic scenario compared to 1 and 2. Note that the ultimate number of clients in the system would be as shown in x-axis in Figure 24.

When transactions maintain longer affinities, as in application level solutions, it results in high contention and very high service completion times as shown in Figure 24. Bringing the scheduling decision down to packet-level allows for a significant improvement in service completion times. This translates into an improvement in overall utilization of the system in terms of maximum supported clients as follows: assuming a chunk length of 2 seconds, 1.5 seconds can be considered a request completion time that would result in an acceptable user experience (allotting 500ms for the remaining latencies) in terms of proper utilization of the retrieved content at the client, e.g., for video playback. With this in mind, random scheduling at packet-level already improves on the maximum number of clients that can be served within the above latency by 12.5%, almost 2000 more clients, compared to the 1 minute affinity model. CArDS is able to further improve on this by serving almost 24000 more clients with the same service completion time compared to the random packet level scheduling. Overall, with CArDS we can serve 162% more clients within the bounded latency compared to the long-lived affinity scheduling, with improving by about 133% more clients compared to random scheduling at packet level.

The main takeaway for Scenario 3 is that CArDS performs superior compared to long-lived transaction solutions as well as random scheduling, even in high load settings.

We presented CArDS as a solution to integrate compute awareness with the steering of service requests at the data plane level. Our analysis demonstrates that this integration leads to significant performance improvements over both network-level and application solutions, while our design-related analysis provides useful insights for deployment of our solution. Most importantly, our solution allows for supporting up to 160% more clients in a use case where request times are bounded by acceptable user experience; an advantage that would significantly lower costs for service delivery.

5. Conclusion

This deliverable has discussed possible unification of the SBA and, more generally, mobile network architecture. The central place in this unification is given the notion of a general networked resource, from which a holistic approach to building the mobile network architecture is taken. In a sense, the entire concept is similar to the concept of computer architecture, which successfully operates on similar principles for more than fifty years. When a resource (a peripheral) is added to the system, it is seamlessly interconnected with the central processing unit and integrated in the system, its capabilities and properties are recorded and the component is automatically ready for use. A central aspect of this architecture is multiplexing of requests onto the existing peripherals, i.e. their scheduling by the operating system.

Our SBA unification follows the same principles. We give the networked resources meaning similar to that of computer peripherals, we interconnect them via a robust, resilient and scalable fabric, we equip them with appropriate APIs, needed for their full integration in the entire system, and we finally multiplex, i.e. schedule, the complex service requests onto the available resource set. As this latest step, i.e. scheduling of requests, is the central aspect of this project, and thus of this deliverable, we give it special attention and demonstrate its operation in detail.

The deliverable thus describes in detail a possible realization of the resource scheduling component. We specify in detail the assumptions we make (e.g. the operating environment), its design, evaluation and implementation. Our evaluation of scheduling, i.e. the positive and encouraging results we showed, gives hopes that the entire unification we presented in this deliverable is the direction to follow when designing future generations of mobile networks.

6. References

[1] FUDGE-5G Consortium, “FUDGE-5G Platform Architecture: Components and Interfaces”, Deliverable D1.2. Available online at: https://fudge-5g.eu/download-file/455/UBFW5Rja2ByFBkQdYkQd

[2] "P4 Language and Related Specifications". Available online https://p4.org/p4-spec/docs/P4-16-v1.2.0.html , Retrieved 2 December 2019

[3] Carpenter, B. and B. Liu, "Limited Domains and Internet Protocols", RFC 8799, DOI 10.17487/RFC8799, July 2020, <https://www.rfc-editor.org/info/rfc8799>.

[4] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

[5] Mosko, M., Solis, I., and C. Wood, "Content-Centric Networking (CCNx) Messages in TLV Format", RFC 8609, DOI 10.17487/RFC8609, July 2019, <https://www.rfc-editor.org/info/rfc8609>.

[6] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[7] Jiang, Sheng & Li, Guangpeng & Carpenter, Brian. (2020). A New Approach to a Service Oriented Internet Protocol. 273-278. 10.1109/INFOCOMWKSHPS50562.2020.9162749.

[8] eBPF Foundation, “What is eBPF?”, available at: https://ebpf.io/what-is-ebpf/

[9] Karima Saif Khandaker, Dirk Trossen, Ramin Khalili, Zoran Despotovic, Artur Hecker, Georg Carle. CArDS: Dealing a New Hand in Reducing Service Request Completion Times. IFIP Networking 2022.

[10] The FUDGE-5G Consortium, “Deliverable 1.2: FUDGE-5G Platform Architecture: Components and Interfaces”, 2021. Available at: https://www.fudge-5g.eu/download-file/455/UBFW5Rja2ByFBkQdYkQd

[11] 3GPP, “System Architecture Working Group 2”. Available at: https://www.3gpp.org/specifications-groups/sa-plenary/sa2-architecture/home

[12] 3GPP, “TS23.501; System architecture for the 5G System (5GS); Release 17”, 2022. Available at: https://www.3gpp.org/ftp/Specs/archive/23_series/23.501/23501-h00.zip

[13] NGMN, “NGMN Cloud Native Enabling Future Telco Platforms”, 2021. Available at: https://www.ngmn.org/publications/cloud-native-enabling-future-telco-platforms.html

[14] Hantouti, H., Benamar, N., Taleb, T. and A. Laghrissi, "Traffic Steering for Service Function Chaining", IEEE Communications Surveys & Tutorials (Volume: 21, Issue: 1, First quarter 2019), DOI: 10.1109/COMST.2018.2862404.

[15] Ruozhou Yu. R, Xue, G. and X. Zhang, "QoS-Aware and Reliable Traffic Steering for Service Function Chaining in Mobile Networks", IEEE Journal on Selected Areas in Communications (Volume: 35, Issue: 11, November 2017), DOI: 10.1109/JSAC.2017.2760158.

[16] Blöcher, M., Khalili, R., Wang, L. and P. Eugster, "Letting off STEAM: Distributed Runtime Traffic Scheduling for Service Function Chaining", IEEE INFOCOM 2020 - IEEE Conference on Computer Communications, DOI: 10.1109/INFOCOM41043.2020.9155404.

[17] The Open Networking Foundation, “OpenFlow Switch Specification,” 2015.

[18] Alia Atlas and Joel M. Halpern and Susan Hares and David Ward and Thomas Nadeau, "An Architecture for the Interface to the Routing System", RFC 7921, DOI 10.17487/RFC7921, 2016, < https://www.rfc-editor.org/info/rfc79219>.