CArDS: Dealing a New Hand in Reducing Service Request Completion Times

Karima Saif Khandaker
Huawei Technologies
Munich, Germany
karima.khandaker@huawei.com

Dirk Trossen
Huawei Technologies
Munich, Germany
dirk.trossen@huawei.com

Ramin Khalili
Huawei Technologies
Munich, Germany
ramin.khalili@huawei.com

Zoran Despotovic
Huawei Technologies
Munich, Germany
zoran.despotovic@huawei.com

Artur Hecker
Huawei Technologies
Munich, Germany
artur.hecker@huawei.com

Georg Carle
Technical University of Munich
Munich, Germany
carle@net.in.tum.de

***Abstract*—The cloud-native paradigm advocates agile development and deployment of virtualized micro-services, introducing a flexibility and dynamicity for service endpoints that may exist in many locations of a provider's network, not just data centers. Such ability leaves open the problem of scheduling traffic from clients to those possible locations. In this paper, we position our solution to this problem at the data plane level, avoiding the shortfalls of existing solutions in terms of latency and path stretch. For this, we present a system model for forwarding service requests based on compute information, with a distributed scheduler realizing the traffic steering decision at line rate and with measurable performance gains against existing network-level solutions. We evaluate our solution against several design aspects to provide insights for real-world deployments, while quantifying performance improvements for use cases where such scheduling decisions could indeed be performed at the level of each service request. Here we show that our improvements in request completion time may lead to serving up to 162% more clients within the bounded request time that would ensure acceptable quality of experience.**

***Index Terms*—resource scheduling, service routing**

I. Introduction

Service provisioning in recent years has been largely driven by two trends. Firstly, **virtualization** has enabled the flexible provisioning of *service instances* in a single or across network locations. Technologies have progressed from virtual machines to containers, enabling sub-second availability of service instances. Secondly, the **cloud-native** paradigm postulates the agile development and integration of code, decomposing services into smaller sub-sets, i.e. micro-services, to be deployed and scaled independently, yet chaining within a network towards the original service objective. Herein, the acquired deployment flexibility allows to bring services 'closer' to consumers, localizing up to 72% of the overall network traffic to the customer access networks, as observed in [1].

This work was supported in part by the European Commission under the 5G-PPP project FUDGE-5G (H2020-ICT-42-2020 call, grant 957242).

ISBN 978-3-903176-48-5 ©2022 IFIP

To support this trend, ETSI, classically aimed at telecommunications providers, has defined its Management and Orchestration (MANO) [2] framework to enable long-term placement of compute resources and provisioning of the needed network infrastructure. Those longer-term management decisions are complemented by a lifecycle management (LCM) and control capability of the supporting cloud-native platform [3], [4], realizing shorter-term control policies within the management lifecycle as set by the overall orchestration. This supports threshold-driven adaptation to demand changes, e.g., for activating additional resources in case of measured peak times.

However, service placement and dynamic lifecycle management leave open the problem of where to schedule (service) traffic within an instantaneous set of active service instances, i.e. finding the 'most appropriate' service instance to serve the request at runtime of the overall system. Indeed, recent studies show that service time in typical data centers is good on average, yet its distribution is long-tailed [5], [6] and emphasized on the importance of some form of service instance selection and service rate control within the data centers [7]. For a client outside of a particular data center, this problem becomes harder, as service instance information is not available prior to service instance selection. With this, even in the case of an uncongested network, a significant percentage of service requests might experience a low service quality [6]. The **dynamicity** in many scenarios, corroborated in trials [8], e.g., in terms of client load but also participating clients overall, suggests that the decision, which compute resources to use, must be equally dynamic to avoid herding effects [7], i.e. overload situations in one or more of the local sites.

When realizing scheduling across several network locations, those decisions are often tied into separating the naming of the service from the locator-based routing of IP packets. As a consequence, either application-level or DNS-based approaches are mostly employed. Due to the additional costs for making a suitable decision (see [9], [10] for typical latencies), dynamicity of traffic steering is often limited to a longer-lived **service transaction**, i.e. a sequence of service requests that is semantically bound together, for instance due to state created by the individual requests. Another key issue is the introduction of path stretch through routing requests via decision points, such as load balancers or service brokers, thereby further increasing latency and network utilization.

In this paper, we advocate to utilize knowledge from service placement and lifecycle management layers in the decision to provide a **Compute-Aware Distributed Scheduling** (CArDS) capability directly at the data plane level. This links the constraints of executing the 'best' service to the way its packets are being distributed, something ISP and service provider can agree on to improve the overall service experience.

For this, we interpret the scheduling decision as a **service request routing** problem at the data plane level. We solve this problem through a system model for the access provider (e.g., an ISP, company, or mobile telecommunications network) that realizes a service-centric decision at the start of every service transaction, while supporting longer-lived **affinities** to specific instances. For realizing those decisions, **compute awareness** plays a key role, which is provided by abstracting the specific compute capabilities of service instances as **compute units**, those being assigned during service orchestration and lifecycle management as an outcome of the demand-supply matching done in the orchestration phase. Utilizing this longer-term planning outcome allows for **minimizing signalling**, reducing the load on the routing system. The compute units are taken into account within a weighted round robin scheduling mechanism, realized in **distributed schedulers** that reside at the network ingress, serving one or more clients connected to them. Through this distribution, path stretch is avoided, while the simplicity of the scheduling decision allows for realizing the needed forwarding decision in hardware or via programmable forwarding technologies. Most importantly, the consideration of service-specific compute capabilities leads to **improved request completion times** and therefore superior end user experience, which is important for use cases in which request times are bounded by acceptable quality of experience.

With this, our main contributions are as follows:

- We introduce a system model for a constraint-based service routing infrastructure as a basis of our solution.
- We propose a lightweight, runtime compute-aware distributed scheduling (CArDS) solution for the access network, realized at its data plane and minimizing the signalling of compute resource information.
- We outline the realization of our solution in routed as well ingress-egress architectures.
- We evaluate our solution along several design aspects, also outlining its performance in a typical content delivery use case. Our evaluations show that distributed operation is feasible at scale, outperforming other data plane solutions by up to 80% and enabling up to 162% more clients being served at acceptable request completion times.

In the remainder of this paper, we present our system model in Section II, followed by insights from related work in Section III. We outline our solution in Section IV, followed by an analysis in Section V, before concluding in Section VI.

II. Model & Problem

In this section, we first provide an overview of our networked system, before introducing the assumptions and model to finally formulate the distributed scheduling problem, for which we introduce our solution in Section IV.

A. System Overview

Motivated by our observation in the introduction on locality of most service traffic, our system is comprised of a single network domain, consisting of geographically distributed sites at which **service instances** (SIs) for a given service S are deployed, as illustrated in Figure 1. The network infrastructure consists of **semantic routers** and **forwarding nodes**. Clients issue **service requests** destined to a service identifier S_c, which the incoming semantic router forwards towards a suitable destination, e.g., one of the possibly many **service instances**.

This forwarding decision is realized as a two stage process. First, the router 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 described in Section IV-E. In essence, the semantic router performs an on-path resolution of the service identifier provided in the request to (a direction towards) a possible service instance; with this, the semantic router has taken over the role of the DNS albeit utilizing the compute awareness in the scheduling decision to forward packets. **Forwarding nodes** then simply forward packets to next hop of a semantic router, utilizing suitable encapsulation techniques.

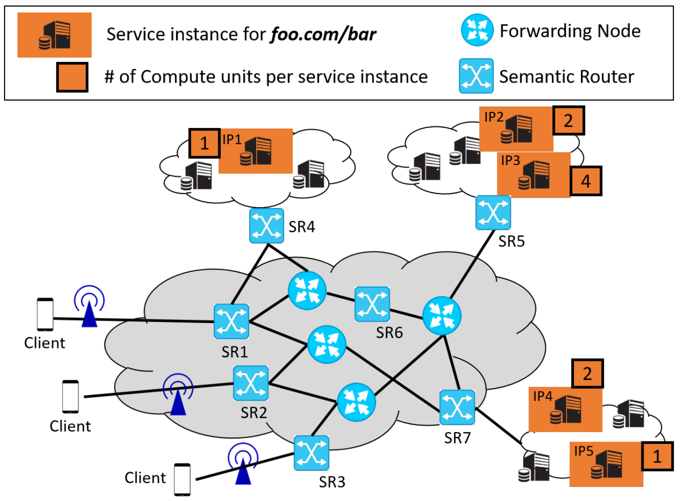


Fig. 1: System overview

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 will need to be sent to the same service instance. For this, we distinguish a *service request*, which can be sent to any available SI, from an **affinity request**, sent to a dedicated SI.

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 that allows for associating the response to the original service request) 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 such as those in [3] could also be used, relying on application protocol specific proxies (e.g., for HTTP) rather than change clients directly.

Figure 1 shows a typical configuration of our system with several distributed sites, hosting SIs of a particular service, named here *foo.com/bar*[1] (the deployed instances are marked here, each assigned their shown IP locator, while other services may likely be deployed in parallel). Each SI for our service is assigned a number of **compute units** (defined in more detail in Section II-B), shown here as numbers assigned to the individual service instances with 10 compute units being the overall capacity of the service in the system.

The figure shows a fully routed deployment of our system, i.e. semantic routers hold next hop information of other semantic routers in routing tables, similar to IP routing. One implementation choice may use IPv6 extension headers [12] to carry necessary service information from one SR to another. However, we may also deploy our solution through a simpler ingress-based architecture. Here, the semantic routers are only deployed at the network ingress, utilizing the SIs locator information as next hop information for forwarding packets, i.e. sending a service request directly to the SI via available IP routing. This model is akin to what the authors in [13] suggest. Such ingress realization may also foresee the placement of the semantic router directly at the client. Alternatively, the semantic router may be an IPv6-based service element, utilizing methods such as Service Function Chaining (SFC) [14] or Segment Routing [15] for routing service and instance requests towards the (ingress) semantic router. In the remainder of this paper, we will utilize the ingress-based model to simplify the description of the operations.

B. System Model

Based on our overview in the previous subsection, we model the network as a directed graph consisting of semantic routers as the nodes and the set of connecting links as the edges of the graph. Forwarding nodes, although part of a real-world deployment, are part of the interconnecting link between two semantic routers in this model. For any link, we model the propagation latency as a sequence of i.i.d random variables, i.e. the resulting propagation latency is time-varying. We also assume that suitable network planning provides enough capacity for each link in our network, thus focussing our model purely on the impact of distributing any incoming service requests to any of the distributed service instances and the resulting waiting times at those servers.

We assume that the SIs for a given service are already deployed in the network, using methods such as those proposed in [16]. Furthermore, for each server Sv in the distributed sites, we assume a total processing capacity of C_{Sv} ($C_{Sv}>0$). We furthermore assume that any SI for service S hosted on a server Sv is assigned a compute unit C_S with $C_S \leq C_{Sv}$, where the total compute units assigned to all SIs hosted on said server do not exceed C_{Sv}[2]. 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 SI will receive from its physical server resource. In addition, each SI is assumed to hold a local buffer to store incoming requests, applying non-preemptive execution of the service the SI implements. The above assumptions are not restrictive, based on insights, e.g., documented in [17], that represent many real-world use cases.

Our model of compute units supports two different **scaling strategies**, namely parallel execution and scaled processing. In the former, service requests are executed with the given processing time but possibly in parallel with the compute unit denoting the number of parallel execution points in the SI. This is akin to *scaling out* in existing container platforms with parallel containers being created. Alternatively, the compute unit may represent a factor to scale the processing time proportional to the compute unit assigned to the SI.

C. Problem

The CArDS problem consists of deciding at **runtime** how to assign incoming service requests to corresponding SIs according to their compute capabilities, as expressed in their respective compute unit assignment, while adhering to the **instance affinity** of the overall transaction.

Our objective is to maximize the system's processing throughput by minimizing the (service) request completion time (as the sum of the delays at semantic routers and SIs, together with network propagation delays) for individual requests. The fitting of any solution to this problem to other objectives, such as jitter minimization, is left out of this paper.

III. Related Work

Before outlining our solution in Section IV, let us briefly discuss our insights into existing solutions for this problem, having guided the goals for our solution.

[1]We use a URL notation purely for readability purpose, with binary formats, such as defined in RFC8609 [11], likely being used in real implementations.

[2]We see the assignment of those SI-specific compute units as part of the overall placement process, providing an input into our scheduling solution.

The work in [18], which serves as one of our comparison solutions in Section V, aligns with the distributed nature and data plane realization of our solution, when applied to a single function SFC. However, it does neither provide cross-site compute awareness nor does it support flexible affinity of service requests, both employed in our solution, leading to the performance disadvantages observed later in Section V. Existing CDN solutions, such as Global Server Load Balancing (GSLB) [19], DNS over HTTPS (DoH) [20] as well as HTTP-level [21] or, more recently, transport-level [22] indirection realize load balancing capabilities at higher layers of the system, incurring latencies through, e.g., DNS resolution, or inefficiently route requests from client to resolution system and back before routing the actual request to the selected instance, all of which leads to inefficiencies in highly dynamic scenarios, as we will observe in our evaluation in Section V-D.

Routing protocols, such as EIGRP [23], provide the capability to utilize load and delay information as input into the routing decision. EIGRP, however, does not provide any support for affinity, therefore leading to problems in scenarios in which individual packets of higher-level service requests may be scheduled to different locations. The dynamic anycast solution in [13] addresses this problem by mapping IP anycast addresses (carrying a service identifier) onto binding IP locators of service instances, thereby supporting affinity towards previously mapped instances. However, the metric-based decision requires frequent signalling towards all ingress nodes of the network, creating an additional signalling load that would increase with finer-granular load and delay reporting.

Message brokers like MQTT [24] or those utilized in existing content delivery networks (CDNs) use an application server to direct the service request to the most appropriate server. Small affinities, particularly down to a single request, would require, often significant, additional signalling. This in turn would lead to more messaging and increased latency due to the additional resolution step, while directing the traffic via the application element introduces path stretch, further increasing the experienced request completion times.

Resolution systems, such as the DNS, often do not allow for constraining requests to resolve a service name, while service brokers may use richer constraints but may lack the necessary network constraints; a situation the IETF's Alto efforts [25] address albeit with the efficiency drawbacks found in other application level solutions. More importantly, DNS requests are rate limited and incur latencies in the order up to 100ms per resolution [9], [10], making its use prohibitive at a request level, while caching of DNS results also leads to problems of possible stale entries delivered to clients.

Earlier work on fair queuing is also relevant here. The work in [26] propose the use of hierarchical fair queuing to provide network load balancing by scheduling packet flows over available paths. However, the solution requires a priori knowledge of each flow type's share of assigned resources and arrival rates, which also limits the supported dynamicity of the affinities towards specific service instance. A similar limitation applies to the work in [27].

IV. Compute-Aware Distributed Scheduling

We now outline the realization of CArDS, performed at the ingress semantic router (see Figure 1). We start with our goals, followed by the individual aspects to realize them.

A. Goals

The analysis of related work in Section III has guided the design for CArDS, resulting in the following design goals:

Utilize available compute awareness: We believe that the awareness of what compute resources are available in the distributed sites within the operator's network is key to improving the overall performance of the service, since it allows for distributing more traffic to those SIs that are capable of serving requests. *Specifically, we consider the assignment of SI-specific compute units in our request forwarding decision.*

Support affinity of transactions: The scheduling of service requests must take into account the semantics of longer-standing transactions, otherwise leading to distributing requests over several network locations, which necessitates a distributed data management system to be in place. *Specifically, we aim to support affinity of any length, including down to scheduling single requests for fully stateless services.*

Distributed realization to avoid bottlenecks: In order to advance over centralized, e.g., broker, solutions, such as those in [24], we aim at realizing our solution on the data path of the service request, i.e. within the ingress semantic router (see Figure 1). *This necessitates a distributed realization, avoiding thus any latency through centralizing the decision, aiming to still perform close to a centralized solution.*

Keep signalling overhead to minimum: Routing based on constraints, such as load or latency, enables service awareness but often comes with the drawback of needing to frequently signal necessary constraint information to the decision points in the network. *Our solution aims at avoiding such frequent update, instead utilizing long-term compute unit assignments.*

Realization at the forwarding plane: Given the intended scheduling on the data path of the service request, forwarding performance can easily be hampered with additional operations to be performed on incoming packets that need forwarding. *In order to ensure suitable performance, our solution is intended for realization at the forwarding plane, i.e. at speeds suitable for high-performance switches.*

While previous work may realize some aspects above, we see the novelty of our solution in addressing them jointly, while providing superior performance. Our guideline for achieving our goals is to prioritize the first three over the last two albeit being conscious of their importance.

B. Assumptions

For the assignment of resources, expressed as compute units, to service instances, the system model in Section II-B applies here. We foresee two possible execution points to realize our solution, namely directly at the client or at the ingress semantic router. We assume the support for instance affinity, as outlined in Section II-A. The impact of such affinity is evaluated in Section V-D in a use case driven analysis.

C. Mapping Compute to Routing Constraints

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. For this, we assume the integration of the compute metric assignment in placement methods and service orchestration operations, such as those outlined in [16] and provided by solutions such as ETSI MANO or other platforms.

In order to turn the compute unit assignments into routing constraints, the service orchestration flattens and joins the SI-specific compute units, shown in Figure 1 for the example identifier *foo.com/bar*, into a **compute vector** C_{CV} for a specific service identifier C that represents a set of SIs.

D. Distributing Routing Constraints

The compute vector C_{CV} 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 C.

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.

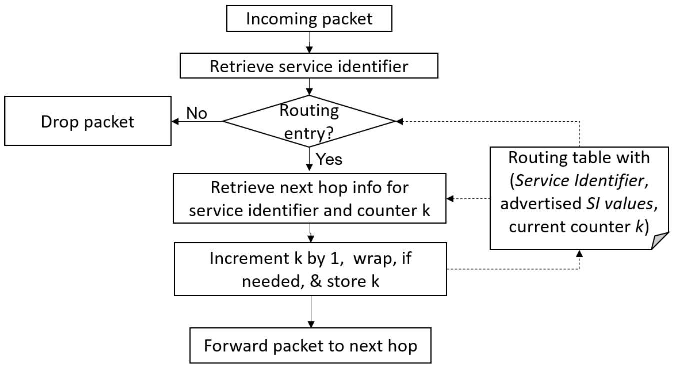
As a consequence, extensions to existing routing protocols, to distribute the computing vector among all routers, will unlikely cause much additional overhead to the routing protocol performance. As an alternative, a service management system may directly signal the routing information to the ingress semantic routers only. While we leave the specifics of the distribution to a real-world implementation, we are confident that the aforementioned rather static nature of compute unit assignments will not pose any real challenge.

E. Ingress Scheduling Decision

This routing constraints are used for scheduling a packet at an ingress semantic router to one of the possible many service instances as follows, shown in Figure 2:

After checking for a routing table entry for the service identifier S provided in the request, the suitable next hop (or SI destination) is selected through a **weighted round robin**, with the weights w_{SI_i} being the compute unit assignments (as described in Section II-B) for the i-th service instance in the compute vector S_{CV} of the service identifier.

In order to avoid the need for implementing multiplications for the weights (i.e. compute units), 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 SIs 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. SI 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.



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

Fig. 2: Scheduling decision at ingress semantic router

F. Realization at the Data/Forwarding Plane

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[3] can be used for realizing such operations at line speed. Using structured binary names [11] 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. The work in [28] has shown the realization of a constraint-based service forwarding system in P4 at line speeds and its supported operations could be used to realize CArDS scheduling decisions.

V. Evaluation

In the following, we first evaluate the impact of various design aspects on CArDS' performance (Section V-B), covering the comparison against two other dynamic scheduling mechanisms in Section V-C. Finally, Section V-D evaluates CArDS' performance in a video streaming use case to analyse the effect of packet-level versus application-level requests with longer affinities and subsequently, the performance of CArDS against random packet-level request scheduling.

A. Setup

The CArDS implementation was written in Python and the evaluations were performed using an event-based simulator of custom Python libraries. The evaluation setup is illustrated in Figure 3 and specified in Table I, unless otherwise stated. The distribution of compute units across individual sites is indicated in Table I, where *S0* refers to the first site, and the compute units per SI are indicated in brackets. CArDS is realized in ingress semantic routers, shown in Figure 3 within an ingress-based architecture as outlined in Section II-A.

All service requests are to one service function only, sent as single packet requests. Each client sends service requests to its

[3]https://p4.org/p4-spec/docs/P4-16-v1.2.0.htm

assigned ingress, with the network load varied by configuring the total number of clients. A 100% workload (network load equal to maximum processing capacity of all service instances) is simulated using 1550 clients (shown as a dotted grey line in Figure 4), which are distributed equally across the 5 ingress nodes. Our simulations vary the network load between 20% and 110%, represented by 315 and 1710 clients, respectively.

The main metric of the performance evaluation is the **mean request completion time (RCT)** of service requests, referring to round trip time from issuing a request at the client, processing at one of the service instance and returning to the client. The scheduling latency is considered to be negligible with link latencies and server processing times [18] as shown in Table I. As a scaling strategy, we utilize parallel processing (as explained in Section II-B) of incoming requests at service instances, unless otherwise stated.

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.

TABLE I: Evaluation configuration parameters

Scenario	Parameter	Configuration / Distribution, Value
All	Link Latencies	Inter-Site: exponential with average 3000µs, Intra-Site: exponential with average 700µs
	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 Dist. *(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 Dist. *(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 Dist. *(Uniform across sites, imbalance within site)*	S0[1,1,1,7], S1[1,1,1,7], S2[1,1,1,7], S3[1,1,1,7], S4[1,1,1,7]

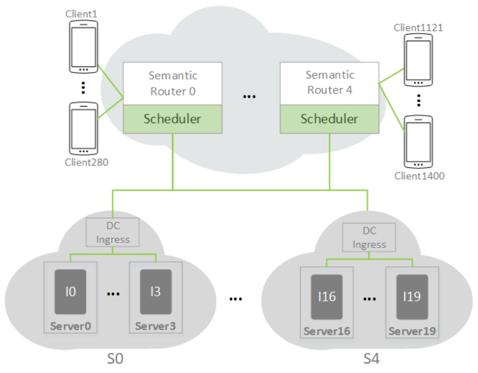


Fig. 3: Evaluation network topology

B. *Scenario 1: Design Aspects of CArDS*

This section evaluates two key design aspects, namely that of distributing the scheduling decision as well as the additional use of site-local load balancers in conjunction to CArDS.

Central vs Distributed Scheduling: In order to understand the effects of distribution as well as scaling the number of distributed ingress semantic routers, we run a test case with only one, centralized, scheduler. Given that our network latency is independent from the number of traversed links, we neglect the effects on path stretch, and therefore real-world network latency, when realizing a single central scheduler, in order to focus on the scheduling impact only.

We expect to observe better central scheduler performance compared to distributed ones, as the centralized scheduler will reduce contention by sending only one request to the same compute unit in one round of scheduling, thereby likely creating a smoother utilization of service instances. In contrast to this single central use of a counter (as explained in Section IV-E) in our scheduling decision, those counters are now distributed across schedulers without any synchronisation among them, which in turn increases the probability of conflicting scheduling decisions. Our tests report an increase of request completion times of around 11.3%, only when the load approaches maximum capacity (i.e. 100% of the total compute capacity). *In other settings, we do not observe any important difference between centralized and distributed scheduling. In conclusion, distribution as such does not appear to be problematic for scheduling.*

However, there may be settings in which the ingress nodes, and therefore the number of schedulers, scale disproportionally beyond the number of compute sites. In fact, any deployment with heavy compute centralization fits this situation. It is therefore essential to check the effects of distribution when this disproportion holds. To address this, we run tests in which number of clients and schedulers alike scale up, ranging from 315 to 1710 as indicated above. Thus, for each load, we check if that load is better scheduled by exactly one, a small, moderate, etc. up to an extremely large set of schedulers. *The finding from the previous paragraph applies here too: only extreme loads cause deterioration of distributed scheduling compared with the centralized one.* Although this deterioration grows with the scheduling distribution scale (e.g. at 100% load, a 29% increase in RCT for 50 schedulers is observed, as opposed to 11% increase for 5 schedulers), it stays in all cases within reasonable bounds. For example, in the worst case, for the highest load and 1550 schedulers, the service completion times are around 50% higher than with the centralized scheduler. Considering that 1550 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.

Scheduling to instances directly vs via site-local load balancers: CArDS supports scheduling service requests directly to instances using their service identifiers or scheduling to the DC ingress instead, which is then responsible for directing the traffic inside the data center. The latter mode is preferable for deployment scenarios which may not want to expose the instances directly to network-level routing but use DC-internal mechanisms instead. For our evaluation, we assumed such DC ingress at the site level that acts as a simple 'random' load balancer, i.e. being unaware of the computing capabilities of the instances but instead routing packets to one of the local instances uniformly at random. *We found that the 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 than that of directly scheduling to instances, while when the load is 80% of the compute resources, 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. This is further observed in Scenario 2.

C. Scenario 2: Comparison with other Resource Schedulers

To evaluate the improvement over existing (network level) solutions, we compare against two other distributed, dynamic scheduling approaches: STEAM [18] and Random Scheduler. The scheduling approaches were selected as viable alternatives to CArDS that are easier to implement and do not require additional load information or signalling. The random scheduler, like CArDS, is also positioned at the ingress nodes, but is compute-unaware and so does not consider the instance's compute capacity in the scheduling decision. It performs a random load balancing across the sites by selecting an instance uniformly at random from all the instances of the network to schedule a received request to.

On the other hand, STEAM's approach to scheduling uses load estimation and local instance state information to perform batch scheduling at the sites [18]. 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 [18]. Additionally, STEAM does not consider the concept of compute units. Thus, to keep the comparison with STEAM fair, the server policy of processing requests is modified for this comparison in that the servers scale processing rates according to their compute units, instead of utilizing a parallel processing capability (see Section II-B). 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 5 ingress nodes 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 that of Scenario 1.

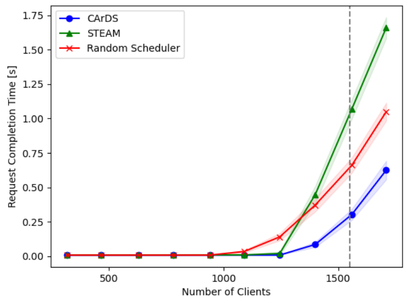
Overall, our comparison allows to observe the effect of factoring compute capabilities into the scheduling decision as opposed to load estimations, as well as performing the scheduling at ingress nodes instead of sites. We further evaluate the impact of the compute unit distribution across sites as well 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 I.

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. 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. As a consequence, it is expected to perform well in a network with a balanced distribution of compute units across instances. However, when the compute unit distribution is very skewed, i.e. with a large variance in the minimum and maximum compute capacities, it may perform very badly. We detail our findings across the distribution configuration as three sub-scenarios in the following.

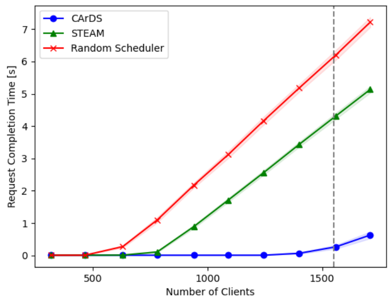
Scenario 2a All three schedulers are expected to perform well with these settings, as we can see in Figure 4a. We also observe that even in such balanced setting, CArDS brings benefits by significantly reducing request completion times in high load settings (i.e. number of clients larger than 1300). Figure 4d depicts the CDF of the request completion times in the setting with 1245 clients, the point where the Random Scheduler performance starts to diverge from the rest. We observe that the tail is very heavy using Random Scheduler, slightly lighter when using STEAM, while there is no tail when using CArDS. Note that the x-axis is in logarithmic scale. This indicates that not only CArDS improves the average performance, but also significantly cuts the tail by distributing the resources fairly among the clients.

Scenario 2b STEAM and Random Scheduler are performing very poorly as depicted in Figure 4b, while CArDS's performance is hardly affected by this imbalance, as it is compute-aware.

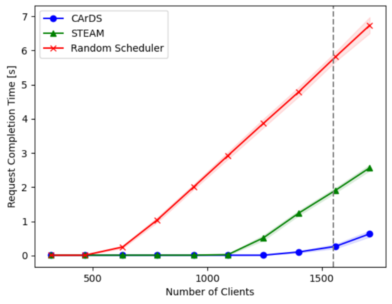
Scenario 2c We observe that STEAM is able to handle



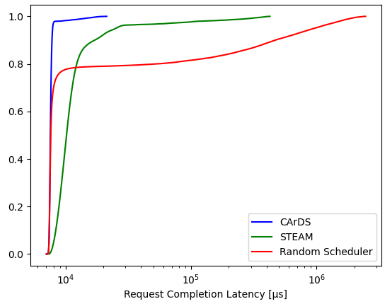
(a) RCT for Scenario 2a



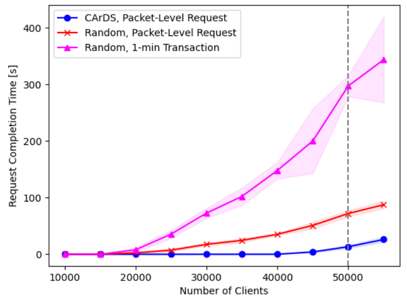
(b) RCT for Scenario 2b



(c) RCT for Scenario 2c



(d) CDF of RCT in Scenario 2a



(e) RCT for Scenario 3

Fig. 4: Results from Scenario 2 comparing the performance of CArDS with STEAM [18] and Random Scheduler (a through d) as well as the performance in Scenario 3 (sub-figure e) in terms of mean request completion times (RCT).

resulting contention within a site, performing similar to 2a, while Random Scheduler provides similar bad performance as in 2b (see Figure 4c). Again, CArDS outperforms both, providing a much lower request completion time at high loads.

The takeaway of Scenario 2 is that CArDS performs superior across compute distributions, compared to both STEAM and Random Scheduler. However, given that STEAM and CArDS both aim at avoiding contention, we identity their combination as a worthwhile study in our future work, in an attempt to have the best of both approaches.

D. Scenario 3: Use Case Driven Analysis

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, which can often be found in localized service scenarios such as those outlined in our introduction and described in [8]. 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 [25], 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 I. Server processing was increased to 600 requests per second [29], 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 are 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 4e.

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 4e. 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. Given the currently limited dynamics in our scenario, our future work will investigate the behaviour under more dynamic client and load conditions, also to compare CArDS against mechanisms such as those proposed in [30]–[33], which apply embedding/placement solutions. We expect that those solutions are unlikely to be efficient in those settings, as the assignment/scheduling decisions performed by these solutions need to be revised over time to adapt to these changes. Such adaptations, however, take seconds due to the high complexity of these solutions, and hence cannot be applied in runtime [30], [33]. Hence, we would expect CArDS to provide improvements over those solutions.

VI. Conclusions

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.

We plan on further studying the possible granularity of compute unit assignments, e.g., driven by shorter term lifecycle control policies, in terms of impact on signalling overhead. We further aim at realizing our design through a network node implementation, while also extending use case centric insights.

References

[1] Cisco, "Cisco annual internet report (2018-2023) white paper," tech. rep., 2020.

[2] "ETSI management and orchestration," tech. rep., ETSI, 2022.

[3] K. Haensge, D. Trossen, S. Robitzsch, M. Boniface, and S. Philips, "Cloud-native 5g service delivery platform," in *2nd Intl Workshop on Mobility Support in Slice-based Network Control for Heterogeneous Environments*, 2019.

[4] Fabrizio Moggio (ed.), "Cloud native enabling future telco platforms," tech. rep., Mar 2021.

[5] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations," in *IEEE 37th Intl. Conference on Distributed Computing Systems (ICDCS)*, pp. 207–217, IEEE, 2017.

[6] X. Zhang, S. Sen, D. Kurniawan, H. Gunawi, and J. Jiang, "E2E: embracing user heterogeneity to improve quality of experience on the web," in *Proceedings of the ACM Special Interest Group on Data Communication*, pp. 289–302, 2019.

[7] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 513–527, 2015.

[8] Steven Poulakos et al., "D5.5: Insights from broadcast, gaming and transmedia experiments," tech. rep., Nov 2019.

[9] "DNS-over-HTTPS performance," tech. rep., SamKnows, 2022.

[10] A. S. Asrese, S. J. Eravuchira, V. Bajpai, P. Sarolahti, and J. Ott, "Measuring web latency and rendering performance: Method, tools & longitudinal dataset," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 535–549, 2019.

[11] M. Mosko, I. Solis, and C. Wood, "Content-Centric Networking (CCNx) Messages in TLV Format," RFC 8609, IETF, July 2019.

[12] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 8200, IETF, July 2017.

[13] Y. Li, Z. Han, S. Gu, G. Zhuang, and F. Li, "Dyncast: Use dynamic anycast to facilitate service semantics embedded in ip address," in *1st Intl Workshop on Semantic Addressing and Routing for Future Networks*, 2021.

[14] J. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," RFC 7665, IETF, Oct. 2015.

[15] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," RFC 8402, IETF, July 2018.

[16] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *IEEE 4th Intl. Conference on Cloud Networking (CloudNet)*, pp. 171–177, 2015.

[17] J.-Y. Le Boudec, *Performance evaluation of computer and communication systems*, vol. 2. Epfl Press Lausanne, 2010.

[18] M. Blöcher, R. Khalili, L. Wang, and P. Eugster, "Letting off STEAM: Distributed runtime traffic scheduling for service function chaining," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pp. 824–833, 2020.

[19] "What is GSLB?," tech. rep., Efficient IP, 2022.

[20] P. Hoffman and P. McManus, "DNS Queries over HTTPS (DoH)," RFC 8484, IETF, Oct. 2018.

[21] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," RFC 7231, IETF, June 2014.

[22] M. Duke and N. Banks, "QUIC-LB: Generating Routable QUIC Connection IDs," tech. rep., IETF, 2021.

[23] D. Savage, J. Ng, S. Moore, D. Slice, P. Paluch, and R. White, "Cisco's Enhanced Interior Gateway Routing Protocol (EIGRP)," RFC 7868, IETF, May 2016.

[24] OASIS, "MQTT Version 5.0," tech. rep., OASIS, 2019.

[25] J. Seedorf and E. Burger, "Application-Layer Traffic Optimization (ALTO) Problem Statement," RFC 5693, IETF, Oct. 2009.

[26] J. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," in *IEEE Transactions on Networking*, vol. 5, pp. 675–689, 1997.

[27] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," in *Computer Communications*, vol. 102, pp. 1–16, 2017.

[28] R. Glebke, D. Trossen, I. Kunze, Z. Lou, J. Rüth, M. Stoffers, and K. Wehrle, "Service-based forwarding via programmable dataplanes," in *1st Intl Workshop on Semantic Addressing and Routing for Future Networks*, 2021.

[29] Stressgrid, "Packets-per-second limits in EC2." Stressgrid, blog, Dec 2019. [Online].

[30] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *TCOM*, vol. 64, no. 9, pp. 3746–3758, 2016.

[31] V. Eramo, E. Miucci, M. Ammar, and F. G. Lavacca, "An Approach for Service Function Chain Routing and Virtual Function Network Instance Migration in Network Function Virtualization Architectures," *TON*, vol. 25, no. 4, pp. 2008–2025, 2017.

[32] A. Satyam, M. Francesco, C. F. Chiasserini, and D. Swedes, "Joint VNF Placement and CPU Allocation in 5G," in *INFOCOM*, 2018.

[33] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: a framework for nfv applications," in *SOSP*, pp. 121–136, 2015.