2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th Intl. Conference on Data Science and Systems

Lite-Service: A Framework to Build and Schedule Telecom Applications in Device, Edge and Cloud

Wei Ling, Chen Tian, Lin Ma, Ziang Hu
Silicon Valley Infrastructure Software Technology Lab
Huawei Silicon Valley Research Center
Santa Clara, CA, USA

***Abstract*— In cloud, many different service frameworks with new concepts such as *Function-as-a-Service (FaaS)*, *Serverless*, and *Stateless* emerge recently. They are essentially certain instantiations of Micro-Service architecture, in which, a software application is implemented as a set of "loosely-coupled services" [32]. From design perspective, these services are considered as the most fine-grained units implementing business logics. Existing frameworks mostly focus on deployment and management of Micro-Services in cloud. However, the internal management within Micro-Service at thread level and function level does not attract enough attention. These missing pieces are critical for telecom applications, in term of programming model, scalability and performance.**

In this paper, we present Lite-Service, a new framework addressing device, edge and cloud operation environments in telecom industry. It adopts FaaS, Serverless and Stateless programming models to separate telecom logics from runtimes in order to improve development efficiency and reduce operation cost. The Lite-Service framework features 2 additional runtimes: function-level runtime for complex telecom function scheduling; thread-level intra-service runtime for fast response to load change. An autonomous self-adaptive scheduler and decentralized service management schemes are proposed to enhance performance and service adaptability in different telecom deployment scenarios. Empirical results show that the Lite-Service framework is effective for building and scheduling telecom applications in device, edge and cloud.

***Keywords—Serverless, FaaS, Stateless, Autonom*ous**

I. Introduction

Cloud service design and programming is transforming from micro-service model to Function-as-a-Service (FaaS) model [9]. The Micro-service architecture describes a particular way of designing software applications as a suite of independently deployable services. The services are often processes, small in size, message driven, and bounded by contexts [32]. Micro-service architecture is known to well reduce development effort and increase operation flexibility. To further reduce the development overhead, FaaS has been proposed as a new service development model in cloud computing. Many cloud providers have released services based on FaaS such as AWS Lambda Step Function [1] and Google's cloud dataflow [2] services. This new model provides a graphical console to arrange and visualize the components of user application as a series of functions, which makes it easy to change workflows and edit the sequence of functions without revising the entire application.

FaaS is also referred to as Serverless computing, since it allows server-side implementation to be transparent to application developers. In Serverless computing, user-developed functions and components are parsed, compiled and executed in an enclosed environment such as containers or virtual machines on the server side. The deployment, load distribution, and resource allocation are all handled by cloud service provider. This new computing model provides dual benefits for both application developers and service providers. In particular, developers now can focus on complex data processing or analysis rather than dealing with service operation and management in a service provider's cloud environment. For cloud providers, this model allows them to attract more services than before and manage their cloud resources more efficiently.

Meanwhile, applying Stateless programming in cloud computing has drawn increasing attention in both industry and academia [4]. The main idea behind "stateless" is to prohibit any unique state or configuration from being saved within a computing unit, such as a hardware device or software thread. This requires developers to remove all mutable or shared variables and make them I/O calls in their functions. A stateless program reduces development and runtime overhead for providing high availability and load balancing, especially in cloud datacenters architected as a collection of standalone resources, e.g. memory, disk, CPU, GPU, FPGA, etc.

The disadvantage of separating state and computation unit is the latency for storing or retrieving data from external resources. However, with the development of RamCloud [16] and Redis [29] as well as faster networks, powerful NIC, and DPDK enabled PCI-e, remote state caching more and more becomes a system design choice [21]. According to [4], using stateless functions and RamCloud, throughputs of services, such as Firewall and NAT, are comparable to that of using local state storage.

In cloud computing, FaaS and Stateless programming models can be seamlessly combined to develop distributed and high-available applications easily. Adopting Stateless model in FaaS ensures that the parallelism is fully leveraged at function level instead of object level [17][26]. The transformation minimizes the overhead of state affinity and synchronization. With FaaS enabled cloud such as AWS step

978-1-5386-6614-2/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2018.00123

function, Google cloud dataflow and Microsoft Azure, developers can scale a stateless function to hundreds or thousands of machines easily by leveraging storage services provided at different locations.

The transformation of Micro-Service towards Serverless and FaaS also brings advantages to telecom industry. Telecom industry often operates same services on a wide spectrum of devices with diversified characteristics, such as legacy operation environment, real time embedded operation environment and modern data center environment [33]. Developing and maintaining different software for the same service is a huge waste. The Serverless and FaaS solves this problem by separating telecom service logics from runtime and having one telecom service software for all deployment conditions. However, all existing Serverless and FaaS frameworks cannot fully satisfy telecom industry. First, these frameworks are developed for cloud operation only, without consideration of resource constrains. Unlike cloud environment, resources of on premise telecom devices are often extremely limited. Each service is constrained with small resource footprint. Second, the latency and scheduling requirements in telecom industry is more strict and complex than current available Serverless and FaaS frameworks can offer. Therefore, a light-weight and highly real-time framework is required to host various telecom services.

In this paper, we introduces a new framework adopting FaaS, Serverless, and Stateless to address all these issues for telecom applications. The paper is organized as follows. Section II describes the motivation of developing a new framework for telecom applications. Section III introduces the design and programming model of the proposed Lite-Service framework. Section IV expatiates several key features and implementations of Lite-Service framework. Section V illustrates the use of the framework and presents interesting experimental results. This paper ends with some concluding remarks in Section VI.

II. MOTIVATION

Telecom software are transformed from monolithic architecture to service-oriented architecture to embrace the cloud development and operation model. Meanwhile, the emerging 5G network standards are defined using service-oriented approach. Although the software trend in telecom industry essentially conforms to the Micro-Service architecture definition, i.e., structuring independently deployable modules and decoupled functions that communicate with each other using messages, many challenges still need to be addressed.

First, breaking latency-sensitive telecom application into a large amount of small functions using existing frameworks significantly increases resources usage and sacrifices performance. Taking the telecom 3GPP Wireless Access application [18] as an example, it consists of more than 20 services such as Radio Mobility service, Information Exchange service, RRC Connection Establishment service, and Transmission Resource Management (TRM) service, etc. In the TRM service [3] (Fig. 1) alone, there are 28 major functions, each containing more than 10K lines of code. Adding up all functions in other services, there are more than 400 major functions in a 3GPP access application. Folding this 3GPP Wireless Access application into a set of function-level fine-grained Micro-Services using current FaaS framework with each function hosted in a container, will incur huge resource usage, inter-function communication overhead, and performance loss. In practice, only large self-contained telecom service should be implemented as an independent Micro-Service. The functions should stay in each micro-service and be managed separately.

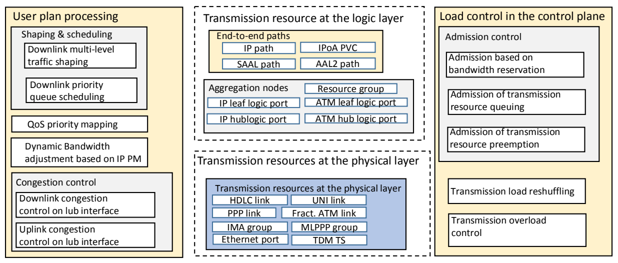
User plan processing
Shaping & scheduling
Downlink multi-level traffic shaping
Downlink priority queue scheduling
QoS priority mapping
Dynamic Bandwidth adjustment based on IP PM
Congestion control
Downlink congestion control on lub interface
Uplink congestion control on lub interface
Transmission resource at the logic layer
End-to-end paths
IP path
IPoA PVC
SAAL path
AAL2 path
Aggregation nodes
Resource group
IP leaf logic port
ATM leaf logic port
IP hublogic port
ATM hub logic port
Transmission resources at the physical layer
Transmission resources at the physical layer
HDLC link
UNI link
PPP link
Fract. ATM link
IMA group
MLPPP group
Ethernet port
TDM TS
Load control in the control plane
Admission control
Admission based on bandwidth reservation
Admission of transmission resource queuing
Admission of transmission resource preemption
Transmission load reshuffling
Transmission overload control

Figure 1. Telecom 3GPP Wireless Access Transmission Resource Management (TRM) Service, contains 28 major functions, each labeled in a rectangular box.

Second, all existing service development frameworks supporting Micro-Service architecture only focus on inter-service management, not within a service. Runtime supports such as name service and load balancers are developed at service-level. However, intra-service scheduling requirements, as well as function-level and thread-level parallelism, cannot be explicitly expressed and exploited by using existing frameworks. For example, 3GPP wireless RRC Connection Establish service that consists of 21 major functions as shown in Fig. 3 has typical telecom application requirements on performance, concurrency, priority and scheduling. In particular, several groups (chains) of functions in this service can be executed in parallel, but functions in a group have to be executed in a special order. What's more, there are strict constraints that need to meet, such as function group (chain) priorities, user session, and message arriving orders, etc. All these function-level runtime requirements must be handled by function developers.

Besides, development and operation efficiency is another factor to consider when telecom applications are developed and executed as services. For example, due to the lack of intra-service support, or function-level runtime support, on resource constrained devices, developers have to manually optimize their code to reduce CPU usage and memory footprint inside each function, which impose significant development overhead and result in long development cycle with multiple code bases for different platforms. This ironically contradicts to the fast development idea of Micro-Service.

Finally, it's important for telecom industry to have a unified service framework that has auto tuning capabilities to adapt to different deployment environments. Telecom carriers operate in a diversified hardware environment ranging from modern server blade to network devices, from centralized data center to remote base stations. Applying one software and one framework across all hardware platforms in cloud, edge, and device is impossible unless the framework has the capabilities of auto-adapting, intelligent scheduling, and self-tuning.

III. Lite-Service framework

The proposed Lite-Service framework is developed to address the challenges mentioned in the previous section. It follows FaaS programming paradigm and allow developers to be only responsible for developing Stateless functions (i.e., no state data needs to be stored) and editing function relationship.

A. *Lite-Service Framwork and Programming Model*

In a Micro-Service based framework, a single layer runtime system is used to manage all services. A resource management system is often introduced if services need to be deployed on a large scale of distributed machines. In contrast, the Lite-Service framework introduces additional runtime layers to support in-service function management as shown in Fig. 2.

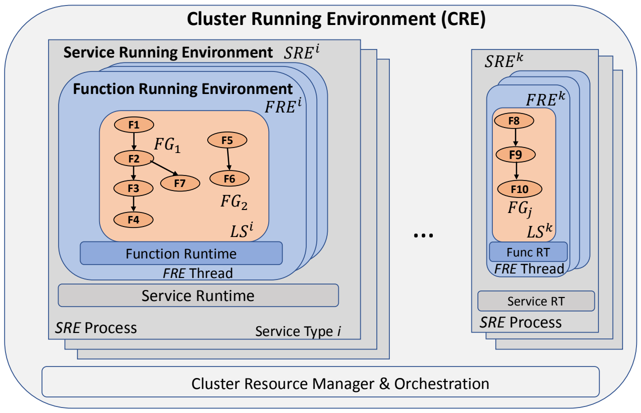

Figure 2. Lite-*S*ervice framework with 3 layers of runtims: *CRE, SRE, FRE*. Each *CRE* can hold one or multiple *SRE*s processes of different service types; each *SRE* can support one or multiple *FRE* instances of the same type by threads; each *FRE* can run one or multiple function groups of the same service (*LS*) type, within single thread.

1) *Function and Service*

The basic element of Lite-Service framework is Function (***F***). A function represents a user business logic. Functions can be associated together through Events (***E***) as a Function Group (***FG***) in a form of directed acyclic graph (DAG):

$$\boldsymbol{FG =< F_1, F_2, \ldots F_n \mid E_1, E_2 \ldots >}$$

where < | > defines a DAG consisting of elements (listed at left side of |) associated with certain dependency (listed at right side of |). The Service (***LS***) in this framework contains a set (denoted by brackets { }) of independent function groups:

$$\boldsymbol{LS = \{FG_1, FG_2, \ldots FG_n\}}$$

Service holds user application logics. A service type ***i*** is denoted as $\boldsymbol{LS^i}$, and defined by a fixed set of functions and function DAGs (Fig. 2). Currently in Lite-Service, the service types are one-to-one matched with that of Micro-Service.

How to group functions into different services is a developers' decision based on 1) application requirements 2) different programming style, execution models and concurrency methods offered at different layers in Lite-Service framework. Normally a group of tightly-coupled functions with special scheduling requirements should be grouped into one service, such that its concurrency and scheduling will be handled by intra service runtimes. No concurrency, threading or runtime related instructions need to be taken care in each function.

An IDE is built to help users to edit functions and function DAGs together with event function mapping (like Fig. 3). With single click, service can be compiled and deployed to its running environment at target machines.

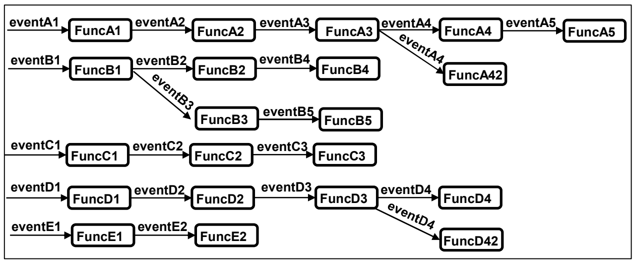

Figure 3. Composition of 3GPP RRC Connection Establish Service. There are 5 independent function groups running in parallel but triggerd by different events; In each function group, there are multiple dependent functions.

2) *Function Running Environment (FRE)*

Functions within a telecom service share special processing order and scheduling priority rules, therefore are suitable to run in a dedicated runtime. In Lite-Service framework, we introduce a new layer of running enviornment: Function Running Environment (***FRE***). *FRE* of type ***i*** is denoted as $\boldsymbol{FRE^i}$, which handles the service of the same type. *FRE* is fully managed by Lite-Service framework without users' intervention.

The *FRE* sits on top of a single thread, and does not manage the thread. The concurrency of the *FRE* is achieved through co-routines. Functions can be preempted as co-routines by function runtime if they are blocked by long I/O calls. As a result, multiple functions can share the same thread resource concurrently.

The functions are loosely coupled --- function-to-function invocation is through events. Event function relationship can be one-to-one, one-to-many, or many-to-many mapping. The event driven *FRE* execution model provides all the following flexibilities: 1) users can update each function without interrupting the service; 2) users can also add new branching in the function group for debugging on-the-fly; 3) function group execution can be priority-based, i.e. once a high priority event arrives, *FRE* can execute the higher priority event for a different function group after completing current function execution.

3) *Service Running Environment (SRE)*

Service Running Environment (***SRE***) in Lite-Service framework provides parallel running environment for *FRE*s through multiple threads. *SRE* is operated in a process. It consists multiple *FRE*s of same type and a service runtime. *SRE* of type ***i*** is denoted as $\boldsymbol{SRE^i}$. In current model, *SRE* type and *FRE* type are one-to-one matched. *SRE* equivalents to a traditional Micro-Service, but its contents are much enriched.

SRE is responsible for two important tasks: 1) it implements an adaptive fuzzy scheduling mechanism to start threads on the same or different CPUs, or requests CPU resources from cloud based on loads, policies, and CPU availabilities; 2) it handles the special user session ordering and load sharing among *FREs*.

4) *Cluster Running Environment (CRE)*

The main functionality of Cluster Running Environment (***CRE****)* is to schedule, deploy, and manage different Micro-Services, e.g. *SREs*, in cluster. Existing open source cloud running environments, such as Mesos [30], can apply to *CRE*.

IV. Implementation

Lite-Service framework is implemented in C/C++ to avoid unnecessary overhead introduced by language runtime. We heavily leveraged shared memory within a runtime. Lite-Service framework also adapts stateless model for better high availability and load balancing.

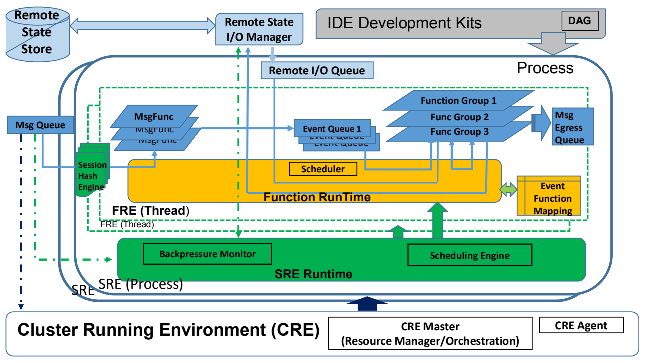

Figure 4. Lite-Service framework implementation

A. FRE Implementation

One of the motivations to introduce *FRE* is to have a fine-grained control on what's inside a service and to decouple with user functions. The main functionality of *FRE* is to schedule user functions based on priority in a highly concurrent way.

1) *Message and Priority Handling*

Each type of ***FRE*** can be configured to process multiple messages as shown in Fig. 4. By definition, each message is associated to a message function (MsgFunc) and a function group. In most telecom applications, one user session may contain multiple different types of messages and have to be executed in order. This constrain is also fulfilled in the Lite-Service framework. *FRE* fetches a new message from *SRE* message queue (Msg Queue), and checks the *SRE* session hash engine for the session constrain of this message. The details of session hash engine is explained in next subsection. If the message is eligible for execution, *FRE* will run the Message Function (MsgFunc) first to convert the message to an internal event, and put it to corresponding event queue. *FRE* then picks event from the non-empty event queue of highest priority, and invokes corresponding functions. (Periodically, *FRE* can clean up lower priority event queues using timer event and high water mark mechanism.)

FRE allows developers to define priorities for messages and function groups. *FRE* makes scheduling decisions by observing an arriving event's priority to minimize delay of time-critical messages for diversified quality of service.

2) *High Concurrency in FRE*

The concurrency within an FRE thread is achieved by co-routines, i.e. functions stack that can suspend and resume their execution while keeping their state, just like context switch. Considering the fact that remote state access latency can vary significantly depending on where the storage device is located in Serverless and Stateless environment, we add co-routine support in *FRE* to maximize CPU utilization while a function is I/O blocked,

In Lite-Service framework, we make the Stateless operations, e.g. blocking or long I/O calls, a dedicated API. Our API-based approach makes user function clean and simple. By calling these APIs, users do not need to construct remote I/O access messages and manage message sending and receiving in their own logic, which is intrusive and not portable. When these APIs are called, *FRE* function runtime initiates an I/O request, then appends current calling function stack as co-routines. Multiple waiting co-routines are put into a list for each function group. *FRE* runtime can switch to process another event or ready co-routine, and check the responses of I/O requests every time it finishes processing a user function. If I/O response is ready, *FRE* function runtime will update the state of the co-routine from waiting to ready, and process the ready co-routine by switching to the stack of blocked function from where it left and continuing the execution as if there was no execution switching.

Co-routines consume less memory and have less CPU overhead compared to threads. Most importantly, in Lite-Service framework, the number of waiting co-routines are proportional to I/O latency (Fig. 8). This auto-adaptive characteristic perfectly hides I/O latency from user logic. As a result, throughput of a service is maintained constantly high regardless of I/O latency changes.

3) *Backpressure Propagation in FRE*

To minimize the internal queue buffer usage and at the same time prevent message from being dropped when the number of *FRE* instances is insufficient to handle incoming messages, we introduce backpressure in *FRE* and propagate backpressure to the message queue that is controlled by *SRE.* The internal event queues keep 2 watermarks, i.e. high and low. When the high watermark is reached, no more message is allowed to be fetched into the internal queue buffer of *FRE,* which increases the size (pressure) of *SRE* message queue. If the low watermark is reached, multiple messages will be fetched from *SRE* message queue, which reduces the size (pressure) of *SRE* message queue.

B. SRE Implementation

The main functionality of *SRE* is to schedule the parallel execution of *FREs* through multiple threading. The *SRE* scheduler has the adaptive capability that frees user from tuning the Lite-Service framework for different deployment conditions.

1) Thread-Level Concurrency

Comparing to process-level parallelism, thread-level parallelism in *SRE* requires less memory, has less communication overhead and can be started quickly. Today, thread-level parallelism is not supported by most available Serverless frameworks [1][2]. The scalability and parallelism of these frameworks can only be achieved by creating additional process-based service container in cluster.

We measured the startup delay of current available Severless and FaaS solutions in Fig. 5. AWS Lambda startup time is about 3 seconds [34]. The Lite-Service *FRE* thread and *SRE* process startup time is about 2 ms and 4 ms respectively. The warm startup option of AWS Lambda has been proposed in [12], the idea of which is to have extra logic implemented in user function to keep Lambda container running (warm) at least once every 5 minutes, otherwise the container will be moved out of system memory. The startup time of AWS Lambda with warm option is about 30ms to 100ms [34]. The Lite-Service *FRE* thread-level reaction time is 15 to 50 time faster than that of Lambda function.

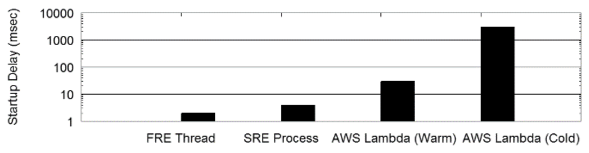

Figure 5. Service startup delay comparison.

2) Session Hash Engine

We introduce session hash engine (Fig. 4) in *SRE* to control execution orders of messages for each user session. The session hash table temporarily stores messages that cannot be executed due to sibling message of the same session is being processed in one *FRE*. Buffered message are fetched by *FRE* after their sibling message is completely processed.

The session hash engine has two major benefits. First, it avoids traditional session lock, which may serialize the processing of function groups and *FRE* instances in *SRE*. Second, since *SRE* creates multiple instances of *FRE* to process multiple messages in parallel, session hash table can also balance the load among *FRE* instances. Specifically, due to Stateless, any available *FRE* can steal a ready message in the session hash table to run, which makes the load being shared evenly among *FREs*.

Note that due to run-to-completion in *FRE*, all queue sizes within *FRE* are limited such that the total number of sessions that need to be buffered in the hash table is normally quite small. When session hash engine reaches maximum, no more new message will be fetched and backpressure is triggered.

C. Adaptive Auto Tuning Implementation

Current available Serverless frameworks, such as AWS Lambda [1] and Google Data Flow [2] rely on cloud orchestration to handle computing resource partitioning, load balancing and concurrency-related operations. The scheduling decision in cloud orchestration, for example by Mesos, is based on the feedback of independent monitoring agent on each server. This approach is not accurate, and has high latency, thus not suitable for telecom services, especially on embedded system.

The Lite-Service framework, in contrast, provides the in-service scheduling engine, which minimizes user intervention, and dynamically adapts to changing loads and different deployment conditions, e.g. embedded or cloud environment.

1) Fuzzy Scheduling Engine in SRE

The *SRE* scheduling engine monitors the backpressure and resource usage of the service, and predicts appropriate amount of resource required. This autonomous scheduling feature is critical for Serverless framework like Lite-Service, because the framework cannot have a fixed scheduling mechanism to fit different user logics and different deployment environments. Determining the correct number of threads and resource at runtime is very challenging because user functions are independently developed and dynamically loaded, and message rate is not predictable. To achieve the autonomous scheduling goal, runtime data collection is implemented in *SRE* to monitor user function behaviors without adding any intrusive instructions in user logic. In addition, we need to make sure the monitoring instructions consume as little CPU resource as possible and not interfere with user functions. We add the following counters in addition to measuring the time variances:

- R_{msg}: Message processing rate.
- $CPU\%$: CPU utilization of a thread.
- C_{empty} : Message queue empty counter. The counter increases when the runtime tries to fetch a message but message queue is empty.
- C_{empty}/R_{msg}: Ratio of message queue empty counter and message rate. We call it message queue emptiness indicator, which is independent of current message rate.
- C_{msgHW} : Message queue high watermark overpass counter. It indicates the buildup of a backpressure in *SRE*

- $C_{coroutine}$: Average (sampling mean) number of co-routine per function group. *SRE* may increase co-routine memory limit if this counter increases.

The major challenge of using these counters is that the moving average and sampling mean of these counters fluctuate quite a lot for each sampling period of 1 to 4 seconds. There is no clear-cut threshold that can be defined for further actions. To cope with this problem, we introduce a fuzz scheduler engine in *SRE.* Users only need to define a set of rough thresholds $\boldsymbol{T}$ in their resource management rules and corresponding actions (Table 1). These monitored counters will go through 1) fuzzyfication process using selected member functions, and 2) fuzzy control logic's inference engine to produce an action using weighted average method.

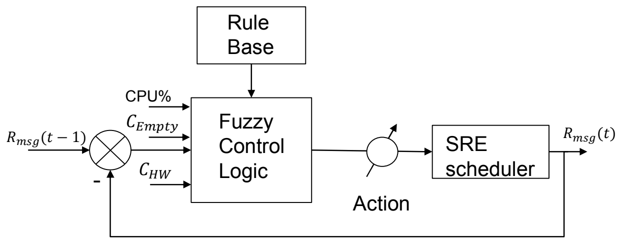

Figure 6. *SRE* fuzzy scheduler engine

The Fuzzy inference engine benefits the Lite-Service scheduling from the following aspects:

- Different rule bases can be dynamically loaded as scheduling policy instead of hardcoded in framework. Table 1 is an example Fuzzy Rule Base that is used by Fuzzy Control Logic (FCL) inference engine to produce actions to *SRE* scheduler for requesting or releasing resources. Using this method, the fuzzy scheduler can be further customized and tuned for different service quality requirements and deployment environments.
- In the future, more data can be collected from *FRE* and *SRE* on-the-fly, and fed to FCL to better tune the system.

TABLE I. FUZZY CONTROL LOGIC RULE BASE EXAMPLE

		$\boldsymbol{C_{msgHW} \geq T_{HW}}$	$\boldsymbol{C_{msgHW} < T_{HW}}$
$\boldsymbol{CPU\% > T_{cpuHigh}}$	$\boldsymbol{C_{empty}/R_{msg} \geq T_{HW}}$	0	0
	$\boldsymbol{C_{empty}/R_{msg} < T_{HW}}$	+2 CPU	0
$\boldsymbol{CPU\%}$ **in range**	$\boldsymbol{C_{empty}/R_{msg} \geq T_{HW}}$	0	-1 CPU
	$\boldsymbol{C_{empty}/R_{msg} < T_{HW}}$	+1 CPU	0
$\boldsymbol{CPU\% < T_{cpuLow}}$	$\boldsymbol{C_{empty}/R_{msg} \geq T_{HW}}$	-1 CPU	-1 CPU
	$\boldsymbol{C_{empty}/R_{msg} < T_{HW}}$	0	-1 CPU

2) Dynamic Resource Reservation Scheme

Current cluster scheduling scheme, such as Mesos [30] and Kubernetes [28], are mostly based on resource offering through a central resource scheduler (Mesos Master and Marathon). The centralized cluster resource scheduler dynamically partitions a cluster, and allocates resources for different service containers. While a Mesos framework can use 'filters' to describe the kinds of resource that it would like to offer [14], it does not have access to a view of individual service on a specific node. Therefore, the overall resource partition, in term of CPU, thread, memory, and network usage may not be optimized for each service. To address this problem, we add resource request capability in *SRE* to help individual service to request preferred resources.

The scheduler we implemented in *SRE* can proactively reserve resource from *CRE* Master (Fig. 4). The *SRE* monitors its load (backpressure) and requests CPU resources or releases them as needed. If additional CPU cores are needed by an *SRE* service, a preferred resource partition is to have new CPU on the current running server of *SRE*, so *SRE* can efficiently manage the parallel processing through multiple threads. To achieve this goal, *SRE* proactively sends a reservation request to *CRE* master asking for more CPUs on the same node. It also creates new *FRE* threads in the same time. When the reservation is granted for the container where the *SRE* is running, the pre-allocated *FRE* threads will be activated on those new CPUs, which leverages the processing capability. If the reservation is not granted, *CRE* will create a new container for the *SRE* somewhere else.

The service based resource reservation scheme adds a new option of swiftly adding thread-level parallel processing capability within a service and effectively handling load changes with small overhead.

D. CRE

CRE is responsible for cluster-level resource management and *SRE* deployment. We implemented *CRE* as an in-house framework on top of a customized Mesos. *CRE* master handles *SRE* resource reservation requests. If a CPU resource is available or can be pre-empted at *SRE* specified node, *CRE* can update the *SRE* container with additional CPU/Memory resource through *CRE* Agent on that node without stopping or restarting the *SRE* container. If there is no resource available or can be pre-empted at *SRE* specified node, *CRE* will find an available resource somewhere else and deploy *SRE* following the traditional Mesos resource allocation and deployment mechanism. This new resource reservation scheme brings new challenges in overall resource partition. The *CRE* master should have the intelligence about how to partition certain critical services and make local or nearby resource pre-emptable when needed. Currently we use a heuristic-based solution, e.g. high priority and low priority service are initially deployed to the same node. The resource of low priority service can be removed dynamically and reassigned to high priority service through Docker container update command.

V. EXPERIMENT

In this section, we present two sets of test results to show the advantages of Lite-Service framework. One set of tests compares the performance and memory usage of telecom service implemented 1) using Lite-Service framework, 2) using simple event driven functional programming (E-F) model, and 3) without using any framework (original monolithic implementation). Another set of tests shows the autonomous tuning capability built in the Lite-Service framework that frees users from platform-dependent concurrency, throughput and QoS related operations in cloud and embedded environment.

A. *Lite-Service Framework Efficiency Test*

In this section, the Service-Under-Test (SUT) is the Wireless Radio Access Network (RAN) Transmission Resource Management (TRM) [3] service, the functional block of which is shown in Fig. 1. TRM service manages physical resources, logical resources, load sharing, and QoS control of wireless 3G/4G/5G access network. This service consists of 28 functions in 4 groups, and is deployed on 3 test platforms respectively, as shown in Table 2. The messages are generated by an in-house developed simulator. The message rates are picked to simulate the normal operation throughputs for these platforms. CPU utilizations are measured in these tests. Lower CPU usage means better performance since more rooms for higher loads.

TABLE II. TRM SERVICE TESTING ENVIRONMENTS

Platform	Deployment Environment	Operating System	Message Rate (msg/s)	Feature
ARM-based Small embedded device	Small wireless base station (Memory Size is 150MB)	Linux	1000	1 thread
ARM-based embedded edge appliance	standard wireless base station	Linux	3000	3 threads
x86-based Cloud server	Cloud-RAN [35]	Linux	2000	3 threads

The CPU utilizations test results are shown in Fig. 7. There are 3 different implementations with distinct legends. The black bar shows the performance of original TRM service code, i.e. monolithic implementation that uses direct function calls. Concurrency management and scheduling are all implemented by users in each function. The green stripe bar shows the performance of TRM service implemented based on the simple E-F model. In this implementation, there is neither Serverless concept nor priority and session based scheduling in runtime. This implementation is primarily used as the benchmark to obtain a performance upper bound. The grey bar shows the performance of TRM service implemented using our Lite-Service framework. This implementation not only decouples user logic (functions) with runtime, but also adds concurrency management, priority scheduling and session handling. All 3 implementations use C/C++ and are deployed on 3 different platforms listed in Table 2.

For the single thread small embedded system (left cluster), CPU utilization for all 3 implementations are very similar, CPU usage of using Lite-Service framework is about 1% less than that of original TRM implementation. For the embedded edge appliance with multiple threads (middle cluster), both the Lite-Service implementation and simple E-F model implementation show 2% less CPU usage comparing to that of the original TRM implementation. For the cloud deployment scenario (right cluster), the CPU utilizations of the 3 implementations are 72%, 55%, and 57% respectively. There is about 20% performance gain in using Lite-Service framework comparing to the original implementation. Comparing to simple E-F model, more features, such as backpressure handling, priority and session based scheduling, are built in Lite-Service runtime, this is why the Lite-Service framework consumes 1% to 2% more CPU than the simple E-F model.

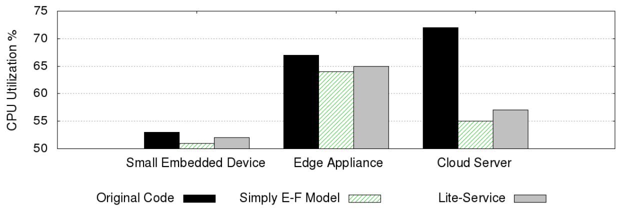

Figure 7. Radio Access Network TRM service test results: the 9 results come from 3 distinct implementations on 3 testing hardware environments.

At the same time, using Lite-Service framework, exact same TRM function code is used for all three deployments, only runtimes are reconfigured to fit different deployment environments. The cost saving of decoupling user logic from runtime operation logic, and making user to develop and maintain only 1 TRM service code for all deployment situations is significant. For the most resource constraint case, i.e. small embedded device (single thread), excluding TRM function code, the Lite-Service runtime code size is configured to as low as only 20KB and the runtime memory overhead is about 40KB. In all three deployment environments, executable size of TRM service implemented using Lite-Service framework is more compact, which is about 19.5% smaller than the original monolithic implementation, and does not add any overhead in dynamic memory usage, which meets the requirements of telecom applications. In addition, the saving of removing platform dependent operation and scheduling code from user function is also significant. There is about 30% user function code deduction. This is because only TRM logic needs to be implemented in user function. Concurrency, scheduling, and platform dependent operations are all handled by the Lite-Service runtime. Overall the test results indicate that Serverless and FaaS based Lite-Service framework is well suited for telecom applications.

B. *Adaptive Auto Tuning Test*

Another important feature that differentiates Lite-Service framework from current available Serverless FaaS frameworks is its adaptive auto tuning capability. In this section, we present 3 auto tuning test results respectively for 1) Stateless I/O latency auto handling, 2) cloud resource reservation scheduling, and 3) function concurrency auto adjustment.

1) *Test Setup*

We choose 2 typical telecom services as Service-Under-Test (SUT) and implement them using the Lite-Service framework. The SUT1 is Wireless 3GPP RRC Connection Establish Service [18] as shown in Fig. 3. This service handles 5 different message types in order to manage user device (cell phone) connection sessions. It invokes 5 function groups and a total of 21 functions. The SUT2 is 5G Edge Computing Service [7], which handles 8 types of messages to deal with 8 major functionalities, e.g. local routing setup, traffic steering, service mobility continuity, QoS, and charging. There are totally 35 functions in this service.

An in-house developed remote state store is used for Stateless tests. The remote state retrieving delay can be adjusted to simulate real world remote I/O latency. A gateway that can emit telecom messages for SUT1 and SUT2 is used too. The telecom message sending interval can be adjusted in gateway to simulate message rate changes. There is flow control between gateway and SUT to prevent message drop.

2) *Auto-adaptive Co-routine Concurrency in FRE*

In this test, the *FRE* auto-adaptive co-routine concurrency feature is tested together with the traditional multiple-thread concurrency. Two tests are performed using SUT1 on single CPU core. For the first test, we enable auto-adaptive co-routine scheduling in one *FRE* with single thread, and for the other test, we enable dynamic thread scheduling in *SRE* without co-routine.

Fig. 8 shows the test results. On top diagram, the curve with 'x' symbols indicates the throughput of single threaded co-routine approach. The curve with square symbols shows the throughput of the multiple-thread approach. The corresponding *FRE* thread number and co-routine number in the FRE are shown as solid black bar and purple hollow bar respectively. The lower diagram shows the remote state I/O delays in microseconds (us) varying across time intervals. The remote state access delay is around 500 us within the first 8 time intervals, then reduced to 250 us until 13th time interval, and finally raised to 2000 us until 19th interval, the end of the experiment.

For the traditional multiple-thread approach, *SRE* adjusts thread count periodically at each time interval of 10 sec. *SRE* fuzzy scheduler detects the remote I/O delays and incrementally adds *FRE* threads to compensate I/O waiting time if delay goes up (e.g. from the 13th interval to the 19th interval). If the I/O waiting time decreases (e.g. from 8th interval to 13th interval), *SRE* removes *FRE* threads. For the single-thread co-routine approach, it does not require any fuzzy scheduler to adjust the number of waiting co-routines. Instead, it auto-adjusts the number of waiting co-routines according to the remote I/O delay instantly, much faster than the periodical adjustment being used in the multiple-thread approach. As a result, the sharp raise of remote I/O delay (e.g. at 14th interval) drastically reduces the throughput of multiple-thread approach, but the throughput of co-routine approach is much smoother than that of multiple thread approach. In addition, the co-routine approach achieves 30%~60% higher throughput compared to the multiple-thread approach. This indicates that *FRE* co-routine scheduling is more efficient than thread scheduling. From these test results, we conclude that *FRE* function-level runtime with auto-adjust co-routine capability is good for Stateless design.

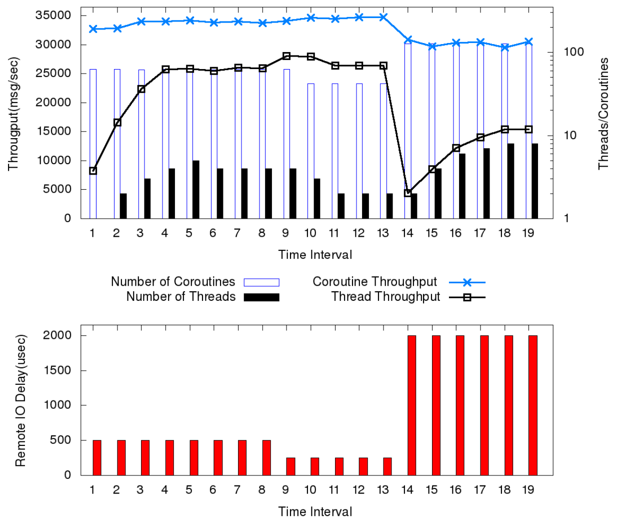

Figure 8. Co-routine concurrency vs. multiple-thread concurrency (SUT1).

3) *Decentralized Cloud Resource Scheduling*

a) *Cloud resource reservation in SRE and CRE*

In this section, we show the effect of *SRE* in auto managing the throughput using *CRE* resource reservation scheme. Both SUT1 and SUT2 are tested in our cloud environment.

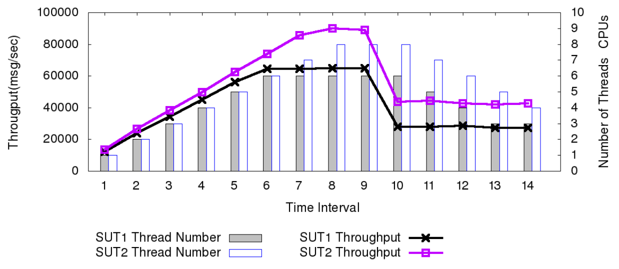

Figure 9. *CRE* & *SRE* throughput adjustment (SUT1 & SUT2)

SRE starts with single thread and single CPU. The gateway sends overwhelming messages to SUT and triggers the backpressure. The *SRE* scheduling engine monitors this change and reserves more CPU resource for current container and adds new *FRE* threads at the same time. The reservation request is unconditionally granted by *CRE* master during these tests. As can be observed in Fig. 9, after 6 time intervals (5 second each), SUT1 obtains 6 CPUs and creates 6 *FRE* threads (solid grey bar). After 8th time intervals, SUT2 obtains 8 CPUs and creates 8 *FRE* threads (purple hollow bar). The throughputs of SUT1 and SUT2 are stabilized at 64500 msg/s (curve with 'x' symbols) and 88000 msg/sec (curve with square symbols) respectively. This is high enough to handle all the gateway demands, thus the backpressure disappears. At the 9th time interval, we reduce gateway throughput by increasing message sending interval from 50 us to 100 us. The message queue emptiness indicator (i.e. C_{empty}/R_{msg} defined in Section IV.C.1) increases for both SUT1 and SUT2. The *SRE* scheduler then starts to release unused threads and CPUs until the end of tests, when SUT1 keeps 3 CPUs and 3 *FRE* threads, and SUT2 keeps 4 CPUs and 4 *FRE* threads to maintain the throughput.

These two test results show that decentralized resource reservation mechanism and adaptive fuzzy scheduler in *SRE* work well in dealing with different loads and different operation conditions. The Lite-Service framework intra-service thread-level scaling capability can benefit from this scheme, which is one of the advantages of Lite-Service framework. In next section we will show the advantages of using multiple thread vs. traditional container-based scaling solutions. The successfulness of resource reservation mechanism requires resource planning ahead of time, or resource pre-emption [31] mechanism being implemented in cloud scheduler.

b) Latency Advantage of Resource Reservation Scheme

The test results shown in this section compare service ready time between decentralized resource reservation approach and traditional centralized resource offering approach. In cloud environment, if *SRE* backpressure is high, *SRE* scheduling engine sends resource reservation requests to *CRE* Master. In our cluster environment, SUT shares CPU resources with low priority services on the same server. Once the reservation is granted, the low priority services will be temporarily suspended by removing its resource and reassigning new resource to SUT.

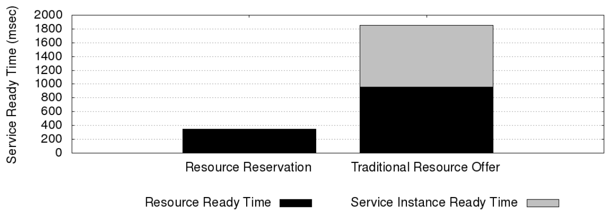

Figure 10. Time difference between resource reservation approach vs. traditional resource offering approach

The service ready time of reservation-based approach is measured from the time *SRE* scheduler initiates resource reservation request to the time *CRE* agent adds CPU resources to the existing container and new *FRE* thread is ready to process messages. The traditional service ready time through resource offering is measured from the time Mesos Marathon detects backpressure in *SRE* to the time new *SRE* is ready to process traffic in new container on a different server. We can see from Fig. 10 that our resource reservation mechanism takes 0.35 sec and traditional offer-based scheme takes about 1.9 sec (1 sec to have new container ready and 0.9 sec to start and initialize service). Our thread based resource reservation approach is 5 times faster.

4) Embedded System Concurrency Auto Tuning

One of the challenges of using Serverless and FaaS based framework on embedded platforms is the uncertainty of CPU resource usages among services and functions that are developed separately and loaded dynamically. It is the responsibility of *SRE* to auto adapt to this dynamic load and resource variation to guarantee throughput of certain critical services. In this section, we present 2 test results to demonstrate how Lite-Service framework deals with this situation.

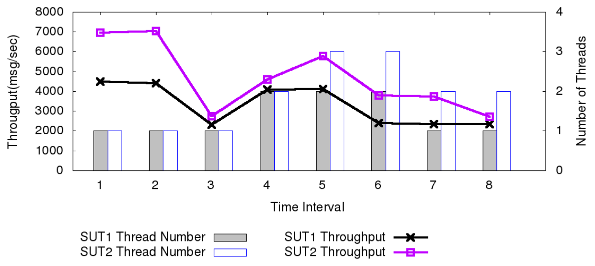

Figure 11. *CRE* & *SRE* throughput adjustment (SUT1 & SUT2)

Fig. 11 shows 2 test results for SUT1 and SUT2. They are tested separately on a 1 core embedded system. The adjustment interval of *SRE* scheduler is 10s. *SRE* starts with 1 *FRE* thread, a constant throughput of 4500 msg/s with 70% CPU utilization for SUT1 (curve with 'x' symbols) and 7000msg/sec with 60% CPU utilization for SUT2 (curve with square symbols).

At time interval 2, several dummy services are dynamically loaded. This is to simulate periodically launched low priority maintenance service and AI training task in wireless base station. As a result, available CPU cycle for service SUT1 and service SUT2 are reduced to 17% and 20%; throughput of SUT1 and SUT2 are reduced to 2300msg/s and 2700msg/s respectively. The backpressure of critical service SUT1 and SUT2 starts to buildup. After detecting the backpressure, *SRE* scheduler creates 1 more *FRE* thread for SUT1 (grey bar) and 2 more *FRE* threads for SUT2 (purple hollow bar). Total CPU usage of SUT1 and SUT2 returns back to 33.6% and 33.8%; throughput returns to 4000 msg/s and 5700 msg/s respectively, which are within the pre-defined throughput range. After this adjustment, the backpressure reduced to an acceptable level, system keeps up with ingress message rate.

At the 5th interval, we reduce gateway throughput by adding delays between produced messages. *SRE* scheduler detects that the message queue emptiness indicator is high and reduces *FRE* thread to 1 for SUT1 and 2 for SUT2. The throughput stays at 2340 msg/s for SUT1 and 2721 msg/s for SUT2.

From 2 tests above, we demonstrate that the autonomous fuzzy scheduler in *SRE* is able to combine collected information and follow predefined scheduling policy to make the appropriate adjustment. The *SRE* runtime plays an important role of guarantying quality of service and balance of CPU usage in embedded environment.

VI. CONCLUSION

In this paper we present **Lite-Service**, a framework that leverages FaaS, Serverless and Stateless concepts to reduce development and operation efforts while meeting performance requirement for telecom applications. The Lite-Service framework decouples platform-dependent operations from user logic and achieves the goal of one telecom service code for all cloud, edge and devices. Another novelty of this framework is the introduction of function-level and thread-level runtime that handles specific scheduling and concurrent requirements within a Micro-Service. This is an untouched area in current Serverless frameworks nowadays, but a much needed feature for complex telecom applications that deal with hundreds of functions within a service and require swift response to load changes. In addition, the fuzzy logic based self-adaptive and autonomous scheduling mechanism at function-level, thread-level and resource management level in the Lite-Service framework provides telecom applications with superior adaptabilities to cloud, edge and device environments. We also verified that the proposed reservation-based decentralized resource management scheme seamlessly integrates with thread-level concurrency, and is much faster than the traditional offer-based centralized resource management scheme.

REFERENCES

[1] http://aws.amazon.com/step-functions

[2] http://cloud.google.com/dataflow

[3] Transmission Resource Management SRAN5.0 Feature Parameter Description, HUAWEI TECHNOLOGIES CO., LTD 2011-9-30 https://www.slideshare.net/yascisse/transmission-resource-management

[4] Murad Kablan, Azzam Alsudais, Eric Keller, Franck Le "Stateless Network Functions: Breaking the Tight Coupling of State and Processing", 14th USENIX Symposium on Networked Systems Design and Implementation 2017.

[5] Murad Kablany, Blake Caldwelly, Richard Hany, Hani Jamjoomz, Eric Kellery "Stateless Network Functions", Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, Pages 49-54

[6] Y Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, et al., "An Efficient communication architecture for distributed deep learning on GPU cluster (CMU)" USENIX-ATC 2017

[7] 3GPP TS 23.501 V1.0.0 (2017-06) "Technical Specification Group Services and System Aspects; System Architecture for the 5G System; Stage 2 (Release 15)"

[8] Wentao Shang† and Jinnah Dylan Hosein "Maglev: A Fast and Reliable Software Network Load Balancer", 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), USENIX Association, pp. 523-535

[9] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy, Aleksander Slominski, "Report from workshop and panel on the Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research" First International Workshop on Serverless Computing (WoSC) 2017

[10] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, Scott Shenker "E2: A Framework for NFV Applications", SOSP'15, 2015

[11] Eduardo Morales, Jesse Hoey, L. Sucar "Decision Theory Models for Applications in Artificial Intelligence", IGI Global, 2011

[12] https://serverless.com/blog/keep-your-lambdas-warm

[13] Gul Agha, "Actors Programming for the Mobile Cloud" 2014 13th International Symposium on Parallel and Distributed Computing

[14] Malte Schwarzkopf, Andy Konwinskiz, Michael Abd-El-Malek, John Wilkesx. "Omega: flexible, scalable schedulers for large compute clusters", EuroSys 2013

[15] Neil Deshpande, Erica Sponsler, Nathaniel Weiss. "Analysis of the Go runtime scheduler".

[16] Philip A. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, Jorgen Thelin, "Orleans: Distributed Virtual Actors for Programmability and Scalability", Microsoft Tech Report MSR-TR-2014-41, 2014

[17] Akka documentation, http://akka.io/docs/

[18] 3rd Generation Partnership Project Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA); Radio Resource Control (RRC); Protocol specification (Release 14), 2017 http://www.3gpp.org

[19] Aaron Gember-Jacobson, Aditya Akella, "Improving the Safety, Scalability, and Efficiency of Network Function State Transfers", HotMiddlebox'15, 2015

[20] Sangjin Han+, Norbert Egi, Aurojit Panda, Sylvia Ratnasamy, Guangyu Shi, and Scott Shenker, "Network Support for Resource Disaggregation in Next-Generation Datacenters", Hotnets 2013

[21] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, Thomas F. Wenisch "System-level Implications of Disaggregated Memory", 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)

[22] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, Scott Shenker, "E2: A Framework for NFV Applications", SOSP'15, 2015

[23] https://en.wikipedia.org/wiki/Reactive_Streams

[24] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, Charles E. Leiserson, "Adaptive Scheduling with Parallelism Feedback", PPoPP'06 2016

[25] Jorge Baranda, Jose Nunez-Martınez Josep, Mangues-Bafalluy, "Applying backpressure to balance resource usage in software-defined wireless backhauls" IEEE ICC 2015

[26] Irene Zhang, Adriana Szekeres, DanaAken, Isaac Ackerman, Steven Gribble, Arvind Krishnamurthy, Henry Levy,"Customizable and Extensible Development for Mobile/Cloud Applications" USENIX Symposium on Operating Systems Design and Implementation 2014

[27] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes", 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)

[28] https://kubernetes.io

[29] https://redis.io/

[30] http://mesos.apache.org/

[31] Michael Isard, Vijayan Prabhakaran, Jon Currey 'Quincy: Fair Scheduling for Distributed Computing Clusters', SOSP 2009.

[32] https://en.wikipedia.org/wiki/Microservices

[33] Ericsson 5g systems: Enabling industry and society transformation. In Erisson White Paper, pages 3–14, January 2015.

[34] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, "Serverless Computation with OpenLambda", In HotCloud (2016)

[35] CloudRAN: https://en.wikipedia.org/wiki/C-RAN