Arora, S., Frangoudis, P. A. y Ksentini, A. (2019). Exposing radio network information in a MEC-in-NFV environment: the RNISaaS concept. In *IEEE Conference on Network Softwarization (NetSoft 2019).*

DOI: https://doi.org/10.1109/NETSOFT.2019.8806647

Exposing radio network information in a MEC-in-NFV environment: the RNISaaS concept

Sagar Arora, Pantelis A. Frangoudis, and Adlen Ksentini
EURECOM, Sophia Antipolis, France
Email: {name.surname}@eurecom.fr

***Abstract*—The Radio Network Information Service (RNIS) is one of the key services provided by a Multi-access Edge Computing Platform (MEP), as specified in the relevant ETSI MEC standards. It is responsible for interacting with the Radio Access Network (RAN), collecting RAN-level information about User Equipment (UE) and exposing it to mobile edge applications, which can in turn utilize it to dynamically adjust their behavior to optimally match the RAN conditions. Putting the provision of RNIS in the context of the emerging *MEC-in-NFV* environment, where the components and services of the MEC architecture, including the MEP itself, are integrated in an NFV environment and are delivered on top of a virtualized infrastructure, we present our standards-compliant RNIS implementation based on OpenAirInterface and study critical performance aspects for its provision as a virtual function. Since the RNIS design and operation follows the publish-subscribe model, we provide alternative implementations using different message brokering technologies (RabbitMQ and Apache Kafka), and compare their use and performance in an effort to evaluate their suitability for providing RNIS in an *as-a-service* manner.**

***Index Terms*—MEC, RNIS, Network Softwarization, NFV**

I. Introduction

Radio Access Network (RAN) awareness can prove beneficial for a wide range of applications in an LTE/5G and beyond context. A wealth of useful information is constantly generated at the RAN level, such as events pertaining to the network control and data planes, User Equipment (UE) status and capabilities, mobility events, location updates, and information on the radio conditions at the user end. These data were traditionally available only to the network operator via the mobile network equipment's vendor-specific monitoring and management interfaces. However, with the advent of Multi-access Edge Computing (MEC), this situation is expected to change. As per the ETSI MEC standard [1], a MEC platform (MEP) provides a set of services to application instances running at the mobile edge, among which is the *Radio Network Information Service (RNIS)* [2]. This service allows authorized MEC applications to consume RAN-level information, such as UE channel quality indications and location updates, which they can utilize to offer enhanced services and optimize performance. This creates space for innovative, RAN-aware third-party applications deployed at the mobile edge, in areas ranging from network troubleshooting and network resource management [3] to Quality of Experience (QoE) optimized service delivery [4], [5].

From the perspective of the network operator, harnessing the potential of these data requires to deal with significant challenges. At the MEP level, handling such amounts of data and efficiently delivering them to MEC applications is already non-trivial. RAN-level data are generated at high volumes and have to be treated at the edge, where storage, processing and memory resources are typically scarce. Scalability challenges thus emerge as the number of mobile terminals generating data and the number of MEC-hosted applications consuming the RNIS grow.

This gets more pronounced in a *MEC-in-NFV* environment [6] and as Network Slicing finds its way towards edge computing.[1] In this environment, the MEP and its services, including the RNIS, are instantiated on demand over an edge cloud as virtual instances and as parts of a network slice instance. MEC orchestration components thus need to appropriately allocate compute resources to multiple RNIS instances corresponding to multiple MEC tenants.

This paper contributes towards better understanding the performance requirements of offering *RNIS-as-a-Service (RNISaaS)* in a MEC-in-NFV environment. We design and implement a RNIS featuring a *standards-compliant* publish-subscribe API. We compare two candidate solutions for its implementation (RabbitMQ [8] vs. Apache Kafka [9]) and carry out testbed experiments to evaluate their performance capabilities, characteristics, and suitability for the provision of RNISaaS. To the best of our knowledge, although existing works focus on potential applications of the RNIS, this is the first work that addresses the design and implementation of the RNIS component itself, its internal workings, and their performance implications particularly towards MEC-in-NFV.

II. Background

A. MEC Architecture

The first released document of ETSI MEC covers the reference architecture [1], specifying the different necessary components and their interfaces. It introduces four entities: (i) MEC platform (MEP) that acts as an interface between the mobile network and the MEC applications, (ii) MEC host that

This work has been partially funded by the EC H2020 5G-Transformer Project (grant no. 761536).

[1]MEC support for network slicing is actively discussed under the ETSI 024 work item [7].

may host both the MEC framework and MEC applications, or only MEC applications, by providing a virtualization environment, (iii) MEC application, that is executed on top of the Network Functions Virtualization Infrastructure (NFVI) of the MEC host, with a MEP Manager (MEP-M) component in charge of MEP configuration and MEC application lifecycle management (LCM), and (iv) Mobile Edge Orchestrator (MEO) which is in charge of the lifecycle management of MEC applications, acting as the interface between the MEC host and the operator's OSS/BSS. ETSI MEC also introduced the concept of MEC services, which are either provided natively by the MEC platform, such as the RNIS [2], or are provided by MEC applications, e.g., video transcoding. MEC applications can discover MEC services available at the MEC host and register their own via the Mp1 reference point.

The MEC architecture is defined to run independently of the NFV environment. However, specific ETSI MEC activities have focused on the integration of MEC in NFV [6]. This involves running the MEP/MEP-M as a VNF, and delegating MEC application instantiation to a regular ETSI MANO NFVO, and LCM to a VNFM, via standard ETSI NFV interfaces [10].

III. RNIS as a Service (RNISaaS)

The RNIS is a key MEC service, allowing third-party applications to access contextual information on the UE end. Once the MEP is envisioned to be executed as a VNF, it is important to assess its performance, and particularly the performance of the RNIS service in a virtualized environment. Given the volume of data handled by the RNIS and the potentially stringent delay requirements for their delivery to interested applications, the results of such a study can be important for the MEC operator to appropriately dimension the resources to allocate to each RNIS virtual instance and to set up the management mechanisms for their automatic scaling to meet the performance requirements of the MEC tenants. At the same time, such results can provide insight on the choice of the suitable technologies for the implementation of specific internal RNIS mechanisms.

A. Implementing a MEP on top of OpenAirInterface

OpenAirInterface (OAI) [11] is an open-source implementation of a complete, 3GPP-compliant 4G mobile system, covering the RAN and the Evolved Packet Core (EPC), with current developments focusing of 5G technology. As the MEP is the MEC element that interfaces directly with the 4G network, we implemented the Mp2 interfaces that allow to interact with the OAI EPC and eNodeB. These are necessary to manage traffic steering towards MEC applications, and to gather RAN-level information on the UEs' environment and context, which will be exposed via the RNIS API (and/or other standardized interfaces such as the location API [12]). To implement the first one, we adopted the Control and User Plane Separation (CUPS) paradigm introduced by the 3GPP [13]. CUPS proposes to separate the data plane and the control plane functions at the S/P-GW. The S/P-GW has

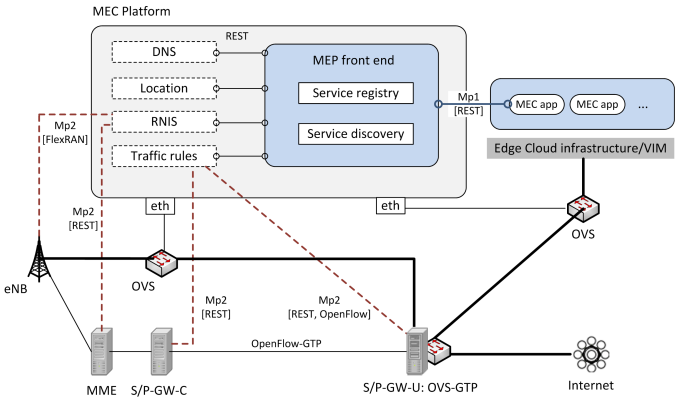

Fig. 1. Architecture of our MEC Platform.

been split into two entities: S/P-GW-C and S/P-GW-U (C for control plane; U for user plane). The former one is in charge of managing the signaling control to create the user-data plane, where the latter is in charge of forwarding the user plane data. The S/P-GW-U is connected to the Internet and the edge virtualization platform. In response to a request from a MEC application (or when requested by the MEO via the MEPM), the MEP installs traffic rules on the S/P-GW-U to offload traffic to the MEC application. In our solution, the OpenFlow protocol is used for this purpose on the Mp2 reference point. In our OAI-oriented MEC platform, the S/P-GW-U is based on a patched version of the OpenVSwitch (OVS) software, which adds support for matching GTP packets.

On the other hand, the FlexRAN [14] protocol is used to implement the Mp2 interface towards the eNodeB to obtain radio statistics and expose them via the RNIS API. FlexRAN is a flexible and programmable software-defined RAN platform that separates the RAN control and data planes via a new, custom-tailored southbound API. There are other functions that our MEP can provide (service registration and discovery, DNS, etc.), but they are outside the scope of this paper.

Fig. 1 provides a global overview of our MEC platform's architecture, and its interfaces with MEC applications and network elements.

B. OAI-based RNIS Implementation

The RNIS is exposed by the MEC platform via the Mp1 reference point. This service provides up to date radio network related information to authorized mobile edge applications. This information can be provided with different granularity, using the IMSI, IPv4 or IPv6 address as UE identifiers. For example, the RNIS can provide RAN information per UE, for all the UEs under a specific cell coverage, by Quality Class Identifier (QCI) value, and using various other combinations. Our RNIS implementation offers two methods for fetching this information: First, it provides a simple request-response mechanism where applications can access the RNIS using a REST HTTP interface. Second, it exposes a publish-subscribe interface, where an application can subscribe to a set of notifications to get updates on a range of parameters. The

latter provides more up-to-date, near-real-time information on the radio conditions and gives the opportunity to applications to receive notifications across a rich set of criteria and their combinations. In the ETSI MEC 012 specification [2], these notifications have been divided in eight categories: cell change, UE measurement report, Radio Access Bearer (RAB) establishment, RAB modification, RAB release, UE timing advance, UE carrier aggregation reconfiguration, and S1-U bearer information. The operation of the OAI-MEP publish-subscribe mechanism is illustrated in Fig. 2.

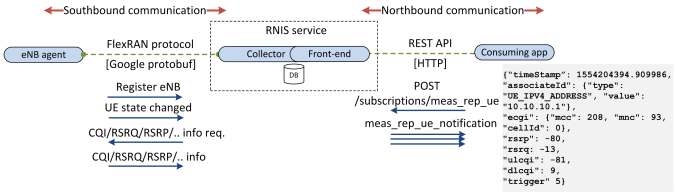

Fig. 2. Operation of the RNIS provided by our OAI-based MEC platform.

At the southbound interface (between the eNodeB and the MEP), the FlexRAN agent of the eNodeB sends messages in raw format including several information on the UE radio context, e.g. CQI, RSRP and RSRQ. Our RNIS implementation includes two components, as shown in Fig. 2. The first component is the *collector*, which receives, parses, and stores the messages coming from the eNodeBs it manages, and formats them appropriately in JSON as specified in [2]. Every notification has a different message structure. The formatted messages are forwarded to the second component, i.e., the *broker/publisher*, which classifies the messages according to the different filtering criteria and notifies the subscribed MEC applications. The messages can be filtered on a per eNodeB (cell) or on a per UE basis. That is, a MEC application can subscribe to notifications related to an entire cell or a set of UEs. Section IV covers the implementation of the broker/publisher in more detail.

IV. RNIS MESSAGE BROKER IMPLEMENTATION

For implementing the RNIS broker component, we have experimented with Apache Kafka [9] and RabbitMQ [8]. The two candidate technologies have fundamentally different design and implementation, which is reflected in their performance, as we shall show in Section V-B.

A. Message Brokers

1) RabbitMQ: This is a traditional message queuing system which implements the Advanced Message Queuing Protocol (AMQP)[2] and is built in Erlang. It follows the standards for AMQP 0.9.1 and can also support AMQP 1.0 via a plugin. In RabbitMQ, all the messages first arrive to an *exchange*, which distributes them to different queues based on a routing key or message header value. Once a message arrives in a queue, the RabbitMQ server pushes it to the consumer(s) listening to the queue.

[2]https://www.amqp.org/

2) Apache Kafka: This is a distributed streaming platform designed around a distributed commit log [9]. It supports consumer clusters, i.e., running multiple consumer instances in a consumer group. In Kafka, the messages are published according to topics and each topic has multiple partitions. A copy of the message is stored in each partition. (Depending on the replication factor there can be more copies in other Kafka clusters.) Once the messages have arrived in the partition, they can be pulled by the consumer groups, if the latter have subscribed to the particular topic.

3) Distinctive characteristics: The two candidate technologies have some distinctive differences:

- **Routing Capability**: RabbitMQ provides various exchanges (direct, fan-out, headers, topic) and extensive capabilities for routing the messages (pattern matching, header matching), whereas in Kafka the messages can only be routed according to topics.
- **Message Storage**: In Kafka, messages are available even after consumption (depending on the message retention period), which is not the case with RabbitMQ, where messages can only be consumed once.
- **Multiple Consumers**: Kafka supports multiple consumer groups subscribing to the same topic. On the contrary, in RabbitMQ if there are multiple consumers listening to the same queue, then the messages they have subscribed for will be pushed in a round robin manner.

RabbitMQ is implemented using the header exchange: a message is routed according to its header, which acts like a routing argument. Every subscriber has its own dedicated queue and these queues have new header values for every new subscription. The higher the number of subscriptions, the higher will be the number of routing arguments. Every subscriber has one RabbitMQ consumer instance running locally, on the same machine were the RNIS application is running. As per the ETSI RNIS specification, the messages have to be delivered to the subscribers via the HTTP protocol. A consumer instance sends an HTTP post request to the callback URL of the subscriber, which is provided by the latter at subscription time, together with the rest of the subscription information.

In **Kafka** this implementation is slightly different. Messages are routed according to topics. For each subscription, there is one topic and one consumer instance (running locally, as with RabbitMQ) which listens to these topic partitions. This consumer instance belongs to a consumer group (one consumer group for one subscriber). Kafka provides the facility for consumer groups to subscribe to the same topic. Every consumer group maintains an offset value which helps to fetchthe messages sequentially or in a random manner. This feature provides the ability to merge similar topics, having similar filtering criteria chosen by subscribers. This reduces the number of topics.

Concluding, in RabbitMQ there is a single dedicated consumer instance posting the notifications to the subscriber, while in Kafka there are lot of consumer instances (belonging to same consumer group) posting the notifications to the

subscriber. We should finally make the following remarks regarding our implementation:

1) Message batching is not considered for any implementation. Messages are posted as soon as they are produced. This provides a near real time view of the network to the notification subscriber.
2) Both message brokers are used in unacknowledged mode. The producer is not waiting for an acknowledgement from the broker. This is done to improve end-to-end (E2E) latency. We should note, though, that both the brokers were found to be reliable in our tests, with no message loss.

V. Performance evaluation

A. Testing environment

Our experiments were performed on a host with a 4-core Intel (i5-3470 @3.2 GHZ) CPU, 500 GB hard disk and 16 GB RAM, running Ubuntu 16.04 LTS. The application was written for python 2.7.12; pika 0.11.2 and confluent-kafka 0.11.4 are the respective python libraries of RabbitMQ (v3.77 with Erlang 21.0.6; standard settings) and Apache Kafka (v2.0.0, Java 1.8.0_181; Java VM settings provided by confluent [15]).

There is one cluster of Apache Kafka and, similarly, only one RabbitMQ broker. The replication and clustering capabilities of Kafka or RabbitMQ were not explored. All applications (RNIS application, broker, subscribers and message producer) were executed on the same host to avoid the effects of network delays on the results of our measurements. Also, the brokers were given the highest priority on the CPU(s) they were running using the *nice* Linux utility.

B. Experimental results

To get insight on which broker is more suitable to implement the RNIS, a set of experiments were performed. Their results are presented in this section.

1) Increasing numbers of subscribers: Considering that every subscriber has subscribed to eight different notifications, we measured the effects of increasing the number of subscribers on E2E latency for both the message brokers. We define E2E latency as the interval from the time instance when specific data to which an application has subscribed are received by the RNIS from the eNodeB over the FlexRAN-based southbound interface (thus generating a publication), until the moment they have been successfully delivered to the consuming application.

Fig. 3 illustrates the effects on E2E latency with increasing number of subscribers. The average E2E latency for both brokers is less than 50 ms, which indicates that both are suitable for near real time applications. When the number of subscribers increases, in RabbitMQ the number of queues starts increasing, which results in increasing workload (replicating messages for every subscriber) for the exchange. The number of routing headers is the same for every subscriber. The number of RabbitMQ consumers posting messages to MEC applications is the same as the number of subscribers. All the subscribers are subscribing to similar eight notifications. In Kafka, this results in eight topics for all the subscribers and when the number of subscribers increases the consumer groups linearly increase. This results in a growing number of consumer groups on topic partitions. There are eight consumer instances in each consumer group posting the notifications to the subscribers. Therefore, the increased number of consumer instances in Kafka in comparison with RabbitMQ results in lower E2E latency for the latter.

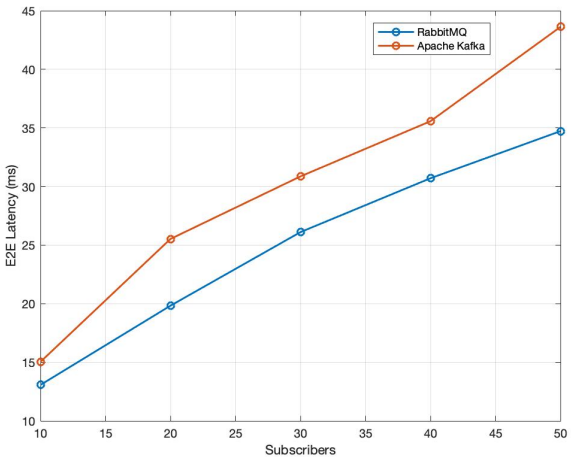

Fig. 3. Effects on E2E latency with increasing numbers of subscribers. Message rate is set to 10/s; each subscriber has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 10,000 * Number of subscribers.

2) Resource utilization: We perform a set of experiments to measure resource utilization for the same message production rate, number of subscribers and number of subscriptions per subscriber[3] and for increasing CPU resources allocated to the broker. In particular, we vary the number of CPU cores assigned to the broker application from 1 to 3, using the *taskset* Linux utility to pin the broker process to a specific set of CPU cores. As shown in Fig. 4-5, Kafka utilizes more resources than RabbitMQ, in part due to the use of Java versus Erlang (for example, how Java handles garbage collection). Second, the message production rate in our experiments is in general kept low; for higher message production rates, it is possible that the curves can deviate. For the experimental settings studied in this work, which we consider realistic, RabbitMQ shows better performance in terms of resource utilization.

We should further note that, as expected, E2E latency consistently reduces with an increase in the CPU resources allocated. This result is important for the operator of the RNIS in an NFV environment: Given a specific workload in terms of the number of UEs (which translates to a specific rate/volume of generated RAN level information) and subscribing MEC applications, and under specific E2E latency requirements, the MEC application orchestrator may appropriately

[3]Message rate: 10/s. Number of subscribers: 50; each of them has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 500,000.

(re-)dimension the resources assigned to a virtualized RNIS instance. This way, it can dynamically scale the number of virtual CPUs allocated to an RNIS instance to match service workload and ensure that it is adequately provisioned to deliver notifications to the subscribed MEC applications in a timely manner, without "overspending" CPU resources.

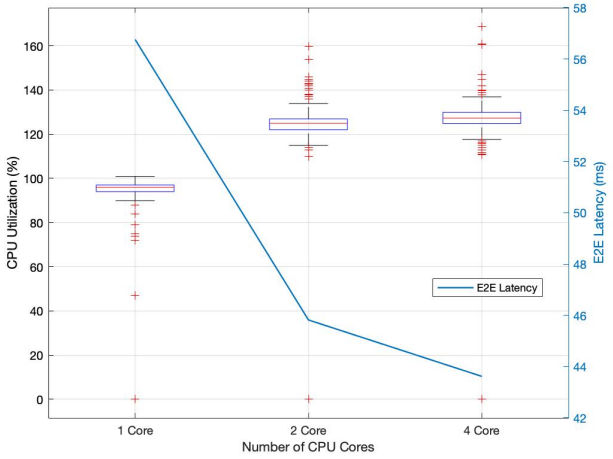

Fig. 4. **Apache Kafka** CPU utilization vs. E2E latency.

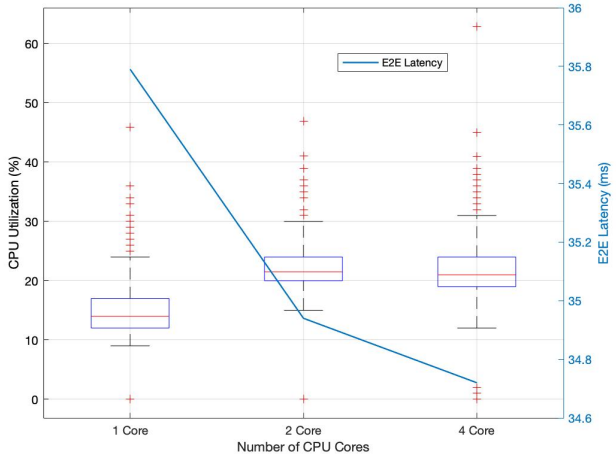

Fig. 5. **RabbitMQ** CPU utilization vs. E2E latency.

3) Discussion: For the given message generation rate (1 message/100 ms), the latency of both brokers was below 50 ms, which makes them appropriate for some real-time applications. However, the number of Kafka consumer instances per subscriber is increased compared with RabbitMQ, where there is one consumer instance per subscriber. This is due to the fact that, for Kafka, if a single consumer instance listens to multiple topics at the same time, then the consumption of messages will be very slow. To compensate for that, for every subscription there is a single instance which improves on latency at the expense of CPU and memory utilization. Furthermore, a Kafka consumer follows the *pull* model, while in RabbitMQ it is based on the *push* model. Therefore, if many consumer instances are polling simultaneously, the broker has to maintain offsets for all consumers. This increases the processing load on the broker. In summary, RabbitMQ appears to be a better option in our settings.

VI. Conclusion

We presented the implementation of a standards-compliant RNIS service on top of an OAI-based MEC platform, and experimental results on its latency and resource consumption performance. We targetted a MEC-in-NFV environment, as recently introduced by ETSI, where virtualized MEC platform components, including the RNIS, are to be executed on top of an edge NFVI, without excluding the case for multiple coexisting virtual RNIS instances, belonging to different tenants and authorized to expose different subsets of RAN-level information. Our results can be used to gain insight about the performance characteristics of the RNIS as a function of the underlying technogies used to implement information delivery, and, importantly, towards dynamically allocating resources to RNIS virtual instances for efficiently providing the RNIS in an on-demand, "as-a-service" manner, satisfying the requirements for timely RAN-level information delivery. We compared the performance of two well-known message brokers (i.e., RabbitMQ and Kafka) for publish-subscribe RNIS message delivery. Our results advocate for the use of RabbitMQ, being more lightweight and thus appropriate for a MEC context, where compute resources are typically more scarce.

References

[1] *Mobile Edge Computing (MEC); Framework and Reference Architecture*, ETSI Group Specification MEC 003, Mar. 2016.

[2] *Mobile Edge Computing (MEC); Radio Network Information API*, ETSI Group Specification MEC 012, Jul. 2017.

[3] J. Pérez-Romero, V. Riccobene, F. Schmidt, O. Sallent, E. Jimeno, J. Fernandez, A. Flizikowski, I. Giannoulakis, and E. Kafetzakis, "Monitoring and analytics for the optimisation of cloud enabled small cells," in *Proc. IEEE CAMAD*, 2018.

[4] Y. Li, P. A. Frangoudis, Y. Hadjadj-Aoul, and P. Bertin, "A Mobile Edge Computing-assisted video delivery architecture for wireless heterogeneous networks," in *Proc. IEEE ISCC*, 2017.

[5] Y. Tan, C. Han, M. Luo, X. Zhou, and X. Zhang, "Radio network-aware edge caching for video delivery in MEC-enabled cellular networks," in *Proc. IEEE WCNC Workshops*, 2018.

[6] *Mobile Edge Computing (MEC); Deployment of Mobile Edge Computing in an NVF environment*, ETSI Group Report MEC 017, Mar. 2018.

[7] *Multi-access Edge Computing (MEC); Support for Network Slicing*, ETSI Std. MEC 024, Jul. 2018.

[8] RabbitMQ. [Online]. Available: https://www.rabbitmq.com/

[9] Apache Kafka. [Online]. Available: https://kafka.apache.org/intro

[10] *Network Functions Virtualisation (NFV); Management and Orchestration*, ETSI Group Specification NFV-MAN 001, Dec. 2014.

[11] OpenAirnInterface. [Online]. Available: http://www.openairinterface.org/

[12] *Mobile Edge Computing (MEC); Location API*, ETSI Group Specification MEC 013, Jul. 2017.

[13] P. Schmitt, B. Landais, and F. Y. Yang, "Control and User Plane Separation of EPC nodes (CUPS)," 3GPP, Tech. Rep., Jul. 2018. [Online]. Available: http://www.3gpp.org/cups

[14] X. Foukas, N. Nikaein, M. M. Kassem, M. K. Marina, and K. P. Kontovasilis, "Flexran: A flexible and programmable platform for software-defined radio access networks," in *Proc. ACM CoNEXT*, 2016.

[15] Running Kafka in Production. [Online]. Available: https://docs.confluent.io/current/kafka/deployment.html