EURECOM
Department of Communication Systems
Campus SophiaTech
CS 50193
06904 Sophia Antipolis cedex
FRANCE

Research Report RR-19-339

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

February 25th, 2019

Sagar Arora, Pantelis A. Frangoudis, and Adlen Ksentini

Email: name.surname@eurecom.fr

[1]This work has been supported in part by the European Union's H2020 5G-TRANSFORMER project under grant agreement No. 761536.

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

Sagar Arora, Pantelis A. Frangoudis, and Adlen Ksentini

Abstract

A Multi-access Edge Computing Platform (MEP), as specified in the relevant ETSI MEC standards, offers various services, and the Radio Network Information Service (RNIS) is one of the most important of them. This service 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 current 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 introduce the term *RNIS as a Service*. We present our standards-compliant RNIS implementation based on OpenAirInterface (OAI) and study critical performance aspects for its provision as a virtualized function. In particular, 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 RNISaaS provision.

Index Terms

MEC, RNIS, Network Softwarization, NFV.

Contents

List of Figures

1 Introduction

Radio Access Network (RAN) awareness can prove itself 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 application instances 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 third-party applications deployed at the mobile edge, which can take advantage of standardized interfaces to access RAN-level information at a fine granularity. At the same time, performance optimization advantages aside, this provides new business opportunities to operators, who can build on the value of their data by exposing them to third parties for profit. The volume and diversity of RAN-level monitoring data, and their correlation with application-perceived performance and with the network's operational state, raise the need and create opportunities for *RAN analytics* [3], which can find use in a wide range of scenarios, spanning from network troubleshooting and network resource management [4] to Quality of Experience (QoE) optimized service delivery [5–8].

From the perspective of the network operator, harnessing the potential of these data requires to deal with significant challenges. At the MEC platform level, handling such volumes 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 service grow.

This gets more pronounced in a *MEC-in-NFV* environment [9] 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.

[1]MEC support for network slicing is actively discussed under the ETSI 024 work item [10].

This paper contributes in the direction of better understanding the performance requirements of offering *RNIS-as-a-Service (RNISaaS)* in a MEC-in-NFV environment. In particular, we design and implement a RNIS featuring a *standards-compliant* publish-subscribe API. We compare two candidate solutions for its implementation (RabbitMQ [11] vs. Apache Kafka [12]) and carry out extensive 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.

This paper is structured as follows: Section 2 provides an overview of the MEC architecture and its evolution towards being integrated in an NFV environment. Section 3 briefly presents the design principles, architecture, and implementation of our MEC platform built on top of OAI, with more details on how the RNIS is provided. The RNIS delivers information to applications following the publish-subscribe model, and Section 4 delves into the details of how message brokering is implemented internally using two different candidate technologies. Section 5 focuses on our testing environment, before we present the results of our experimental evaluation in Section 6. We conclude the paper in Section 7.

2 Background

2.1 MEC Architecture

Since its creation in 2013, the ETSI ISG MEC group has been working on the development of standardization activities around MEC. The first released document of ETSI MEC covers the reference architecture [1], which aims to specify the different necessary components. A high-level representation of the architecture is shown in Figure 1. It introduces four entities: (i) The MEC platform that acts as an interface between the mobile network and the MEC application. It has one interface to connect to the mobile network to obtain information (e.g., usage statistics) about UEs and eNodeBs, in the form of an API to the MEC application, while it interacts with the mobile network to offload traffic to the MEC application; (ii) the MEC application that runs on top of a virtualized platform; (iii) the MEC host that may host both the MEC framework and MEC application or only the MEC application by providing a virtualization environment; (iv) The Mobile Edge Orchestrator (MEO) which is in charge of the lifecycle management of MEC applications, and acts as the interface between the MEC host and the BSS/OSS. Another concept introduced by the MEC ETSI group is that of a MEC service, which is either a service provided by the MEC platform itself, such as the Radio Network Information Service (RNIS) [2] and traffic control, or a service provided by the MEC application, e.g., video transcoding. The MEC host is the key element; it provides the virtualization environment to run MEC applications, while it interacts with the mobile network entities, via the MEC platform, to provide MEC services and data offloading functionality to MEC applications. Respectively, the mp2 and mp1 reference points are used by the MEP to interact with the mobile network elements and provides MEC services, like the RNIS and traffic control to the MEC applications. In addition, two MEC hosts can communicate over the mp3 reference point aiming at managing user mobility via MEC application migration among MEC hosts. Moreover, the MEP allows MEC applications to discover MEC services available at the MEC host and to register a service provided by a MEC application.

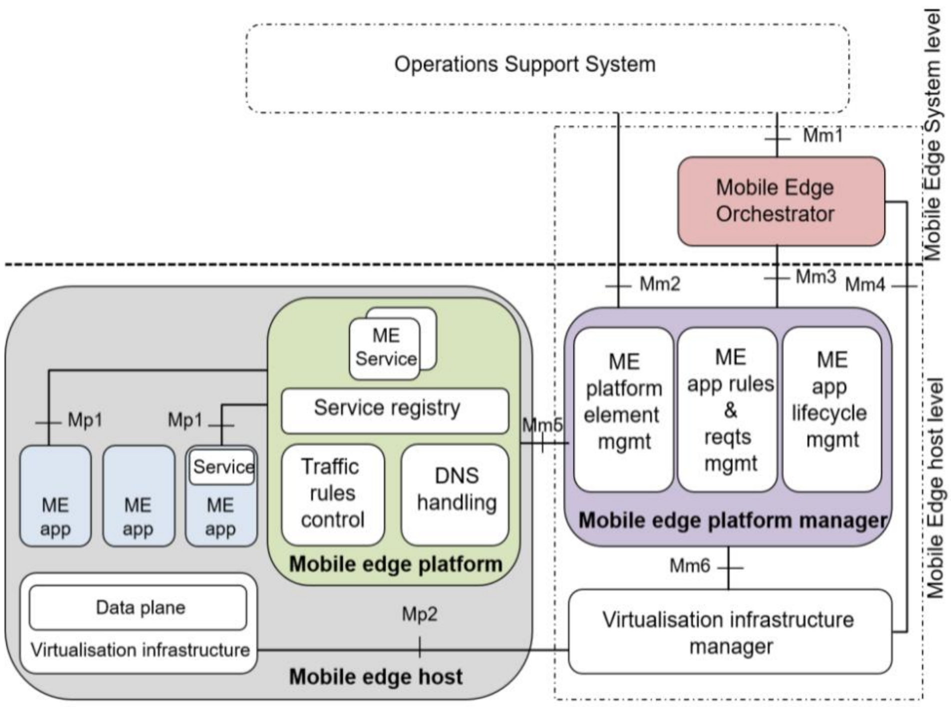
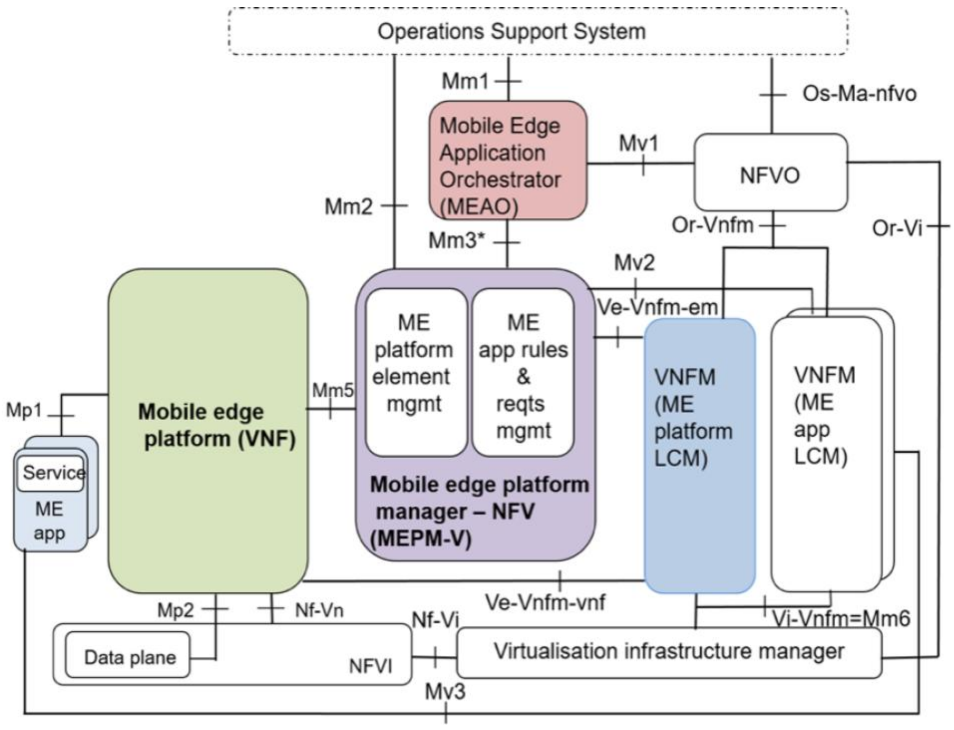


Figure 1: High-level representation of the MEC architecture.

The MEO is in charge of the instantiation and orchestration of MEC applications. The MEPM element is in charge of the life-cycle management of the deployed MEC applications. The MEPM is in charge of the MEC platform configuration, via the Mm5 reference point, such as the MEC application authorization, the type of traffic that needs to be offloaded to a MEC application, DNS redirection, etc. Regarding the services offered by MEP, it provides the RNIS which is one of the most critical MEP features. Through this service, a MEC application can be aware of the radio conditions at the UE end. This information can be provided to the application in near real time, which allows it to react quickly and perform adaptations and performance enhancements, thus being quite beneficial for improving QoS. One of the use cases of this service is in a multimedia delivery context, where a media streaming application hosted at the mobile edge can adapt the codec schemes per UE according to the latter's radio conditions, to optimize user experience.

2.2 MEC-in-NFV model

As described in the precedent section, the MEC architecture is defined to run independently from the NFV environment. However, the advantages brought by Network Functions Virtualization (NFV) and the goal of integrating and running all MEC entities in an NFV environment, has led the MEC ETSI group to update the reference architecture. The proposed document [9] updates the reference architecture as shown in Figure 2. As it can be noticed, the MEC platform and the MEPM will run as VNFs. The MEO has become the MEAO; it keeps the main functions, apart from the fact that it should use the NFV Orchestrator (NFVO) to instantiate the virtual resources for the MEC applications as well as for the MEP. The MEC application lifecycle function has been moved to the VNF Manager (VNFM). Moreover, the VNFM is in charge of the lifecycle management of the MEP as well as the MEPM. Another important difference between the reference architecture and the NFV-oriented one, is the appearance of new interfaces (Mv1, Mv2 and Mv3) and the usage of interfaces defined by the ETSI NFV [13].



Operations Support System
Mm1
Os-Ma-nfvo
Mobile Edge Application Orchestrator (MEAO)
Mv1
NFVO
Mm2
Mm3*
Or-Vnfm
Or-Vi
Mv2
Ve-Vnfm-em
ME platform element mgmt
ME app rules & reqts mgmt
Mobile edge platform manager – NFV (MEPM-V)
VNFM (ME platform LCM)
VNFM (ME app LCM)
Mp1
Mm5
Mobile edge platform (VNF)
Service
ME app
Mp2
Nf-Vn
Nf-Vi
Ve-Vnfm-vnf
Vi-Vnfm=Mm6
Data plane
NFVI
Virtualisation infrastructure manager
Mv3

Figure 2: High-level representation of the MEC-in-NFV architecture.

3 RNIS as a Service (RNISaaS)

The RNIS is one of the important MEC services to be provided by a MEP. It allows 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 check 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 the delivery of these data 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.

This section starts with a brief introduction of our MEP implementation based on OAI. A detailed description of our implementation of RNIS-as-a-Service (RNISaaS) follows.

3.1 Implementing a MEP on top of OpenAirInterface

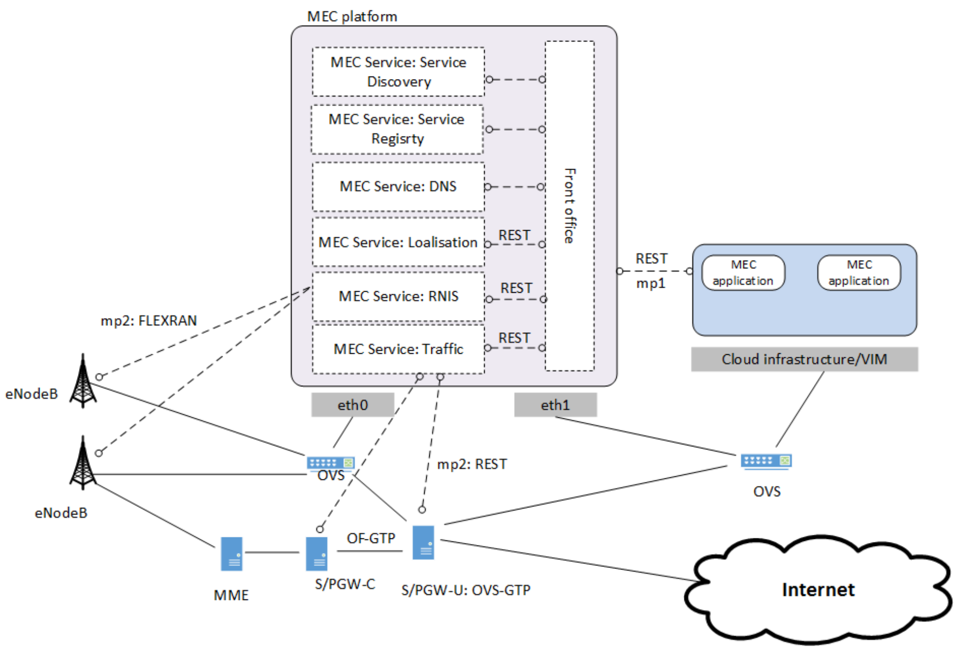


Figure 3: Architecture of our OpenAirInterface-based MEC Platform.

OAI is an open source implementation of LTE components, covering the RAN and the Evolved packet Core (EPC), with current developments focusing of 5G technology. As MEP is the MEC element that interfaces directly with the 4G network, we started by implementing the Mp2 interfaces that allow to interact with the OAI EPC and OAI eNodeB. The first one is needed to implement the traffic redirection at the EPC, as per the request of the MEC application hosted at the edge. The second is needed to interact with eNodeBs 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 [14]). To implement the first one, we adopted the Control and User Plane Separation (CUPS) paradigm introduced by the 3GPP [15]. CUPS proposes to separate the data plane and the control plane functions at the S/P-GW. Figure 3 illustrates the global overview of the MEP-OAI platform. The S/P-GW has 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, while 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. As per the MEC application request (or when requested by the MEO), the MEP installs traffic rules on the S/P-GW-U to offload traffic to the MEC application. In our solution, the OpenFlow protocol was adopted as the Mp2 interface. In the OAI-MEP platform, the S/P-GW-U is based on a patched version of the OpenVSwitch (OVS) software which supports matching GTP packets.

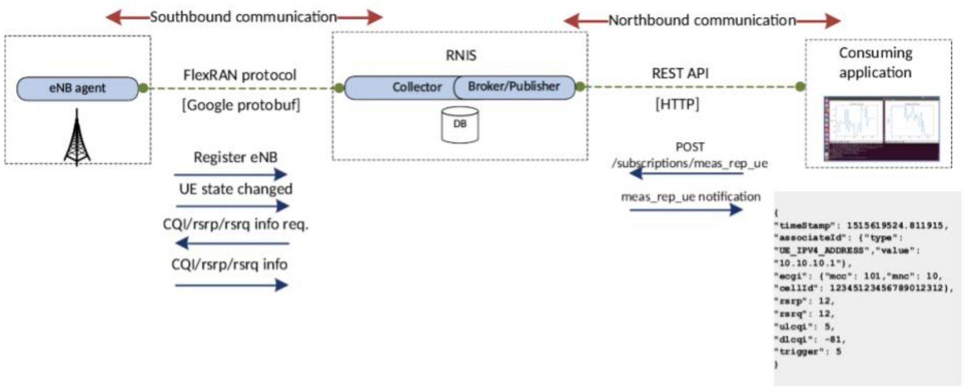
On the other hand, the FlexRAN [16] protocol is used to implement the Mp2 interface towards the eNodeB to obtain radio statistics and expose them via the RNIS API. FlexRAN is a flexible and programmable software-defined RAN platform that separates the RAN control and data planes via a new, custom-tailored southbound API. There are other MEC services that the OAI-MEP can provide (service registration and discovery, DNS, etc.), but they are outside the scope of this paper.

3.2 OAI-based RNIS Implementation

The Radio Network Information Service (RNIS) is provided by the MEC platform via the Mp1 reference point. This service provides up to date radio network related information which can be utilized by any authorized mobile edge application. Broadly speaking, the RNIS can provide the below information regarding a UE:

- Change of cell coverage.
- Timing advance, carrier aggregation reconfiguration etc.
- User plane measurement-related information.
- Bearer formation information and its parameters.

This information can be provided with different granularity, using as a UE identifier its IMSI, IPv4 or IPv6 address. 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 ask for specific information from the RNIS using a RESTful HTTP interface. Second, it exposes a publish-subscribe interface, where an application can subscribe for a set of notifications for getting 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 subscribe for notifications across a rich set of criteria and their combinations. In [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 Figure 4.



```
{
"timeStamp": 1515619524.811915,
"associateId": {"type":
"UE_IPV4_ADDRESS","value":
"10.10.10.1"},
"ecgi": {"mcc": 101,"mnc": 10,
"cellId": 12345123456789012312},
"rsrp": 12,
"rsrq": 12,
"ulcqi": 5,
"dlcqi": -81,
"trigger": 5
}
```

Figure 4: 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. QCI, RSRP and RSRQ. These messages need to be treated and formatted in a JSON format as specified in [2]. OAI-MEP provides a RNIS service to subscribe to all the eight different types of notifications, as described above.

In our implementation, we have divided the RNIS in two components, as illustrated in Figure 4. 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. Every notification has a different message structure. The formatted messages are forwarded to the second component, i.e., the *broker/publisher*. It classifies the messages according to the different filtering criteria, as specified in [2]. For the proposed implementation, 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. Thus, the broker classifies the messages according to these filtering criteria and notifies the subscribed MEC applications. Section 4 covers the implementation of the broker/publisher in more detail.

4 RNIS message broker implementation

As described in Section 3, the RNIS comprises a collector and a broker/publisher component. For the latter, we have experimented with two message brokers, i.e., Apache Kafka [12] and RabbitMQ [11]. The two candidate technologies have fundamentally different design and implementation, which is reflected in their performance, as we shall show in Section 6.

4.1 Message Brokers

4.1.1 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 the messages 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.

4.1.2 Apache Kafka

This is a distributed streaming platform and is designed around distributed commit log [12]. It is designed for handling very high message rates and for supporting consumer clusters (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.

4.1.3 Distinctive characteristics

- **Routing Capability**: RabbitMQ provides various exchanges (direct, fanout, 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.

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

- **Message Storage**: In Kafka the messages are available even after consumption, which is not the case with RabbitMQ. In RabbitMQ, messages can only be consumed once and in Kafka it can be consumed till the time its available, depending on the message retention period.

- **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 round robin manner.

RabbitMQ is implemented using the header exchange: messages are routed according to their headers, 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 more 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 [2], the messages have to be delivered to the subscribers via the HTTP protocol. These consumer instances send an HTTP post request on the callback URL of the subscribers, provided by them at subscription time, together with the rest of the subscription information.

In **Kafka** this implementation is slightly different because of its architecture and provided features. In Kafka, messages are routed according to topics. For every 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 in fetching the messages serially 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. Whereas, 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. The messages will be 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 the E2E latency. We should note, though, that both the brokers were reliable during the whole testing scenario and there was no message loss.

5 Testing environment

The test was 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 in python 2.7.12; pika 0.11.2 and confluent-kafka 0.11.4 are the respective python libraries of RabbitMQ and Apache Kafka. Regarding software versions and settings, RabbitMQ 3.77 with Erlang 21.0.6 was used with the standard settings provided during installation, and Kafka 2.0.0 with Java 1.8.0_181 was used with the recommended Java VM (JVM) settings provided by confluent [17]. The JVM settings used were:

```
-Xms4g -Xmx4g -XX:MetaspaceSize=96m
-XX:+UseG1GC
-XX:MaxGCPauseMillis=20
-XX:InitiatingHeapOccupancyPercent=35
-XX:+ExplicitGCInvokesConcurrent
-XX:G1HeapRegionSize=16M
-XX:MinMetaspaceFreeRatio=50
-XX:MaxMetaspaceFreeRatio=80
-Djava.awt.headless=true
```

There is one cluster of Apache Kafka and one of Zookeeper. Similarly, there was only one RabbitMQ broker. The replication and clustering capabilities of Kafka or RabbitMQ were not explored in this test scenario.

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 command.

6 Experimental results

RabbitMQ and Apache Kafka both have a different architecture, each coming with its advantages and disadvantages. To get a better insight on which broker is suitable to implement the RNIS, extensive tests were performed; their results are presented in this section. The parameters taken into account for each test case are mentioned below every figure.

6.1 Increasing numbers of subscribers

Considering that every subscriber has subscribed to eight different notifications, we observed the effect of increasing the number of subscribers on end-to-end (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.

Figure 5 illustrates the effect 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 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.

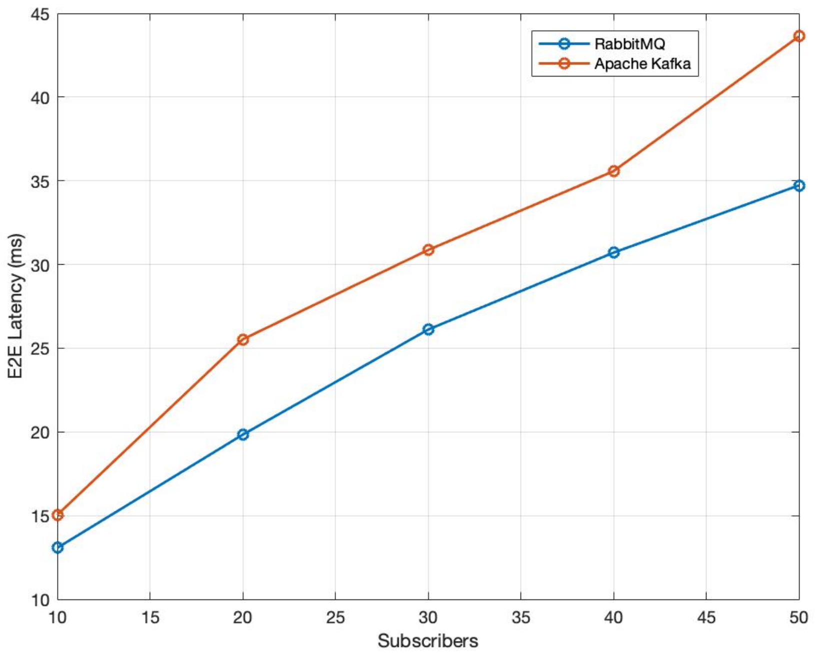


Figure 5: Effects on E2E latency with increasing numbers of subscribers. Message rate is set to 10/s; each subscriber has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 10,000 * Number of subscribers.

We then turn our attention to the E2E delay distribution. To focus on the delivery time of every single message, Figure 5 presents a box plot showing the distribution of E2E latency for a single subscriber taken out of 50 subscribers subscribed to eight notifications.

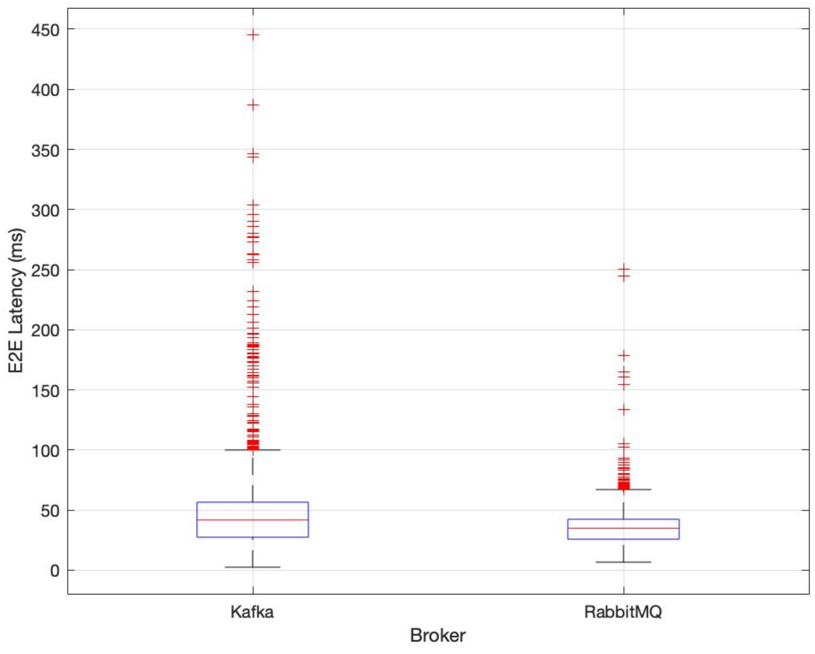


Figure 6: Box plot of the E2E latency in notification delivery for a single subscriber out of 50 concurrent subscribers, each of whom having eight subscriptions. Message rate: 10/s. Total messages produced: 10,000. Messages consumed by this subscriber: 10,000.

The plot contains ten thousand values for each broker. It shows that E2E latency for RabbitMQ deviates less from the mean value in comparison to Kafka.

6.2 Increasing numbers of subscriptions

In this experiment, we are increasing the number of subscriptions from 1 to 8, keeping the number of subscribers constant to 50. RabbitMQ exhibits better performance than Apache Kafka in this test case. In RabbitMQ, increasing the number of subscriptions increases routing headers. This results in a growing sorting load on the exchange. The number of consumer instances posting the messages to MEC applications is constant. Hence, the latency will slightly increase with increasing numbers of subscriptions, but not as per the previous scenario where the number of queues was increasing. However, in Kafka the more the subscriptions, the more the number of topics and consumer instances in a consumer group. This combination of increase in topics and consumer instances results in a growing number of consumer instances on topic partitions, which creates a similar impact as per the previous test case.

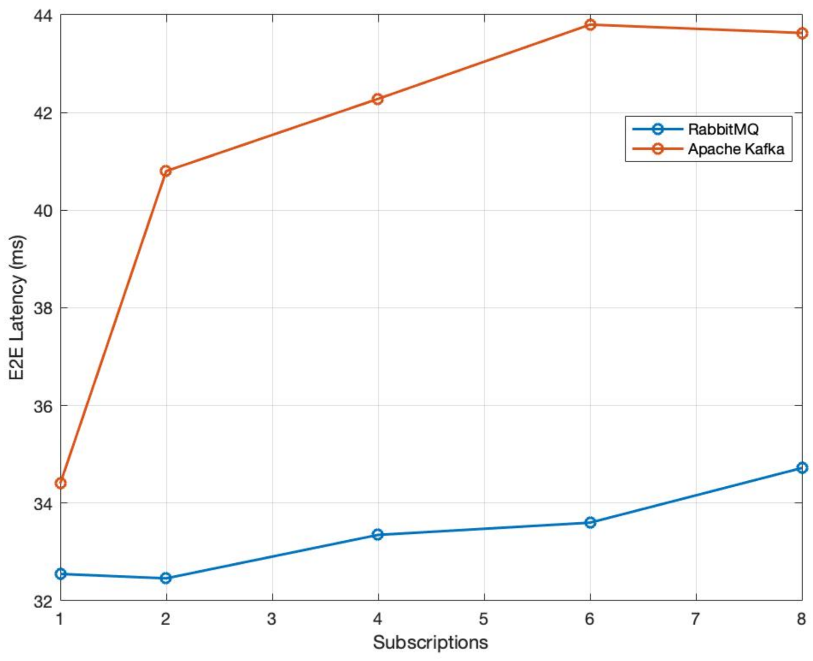


Figure 7: Effects on E2E latency with increasing numbers of subscriptions. Message rate: 10/s, number of subscribers: 50. Total messages produced in each case: 10,000. Messages consumed in each case: 500,000.

6.3 Resource utilization

The average CPU and average real memory utilization metrics are important for understanding the performance of both the brokers. We thus perform a set of experiments to measure resource utilization for the same message production rate, number of subscribers and number of subscriptions per subscriber 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. From Figure 8-11, the Java-based Kafka utilizes more resources in comparison with the Erlang-based RabbitMQ. The first reason for this behaviour is the architectural difference between Erlang and Java (for example, how Java handles garbage collection). Second, the rate of message production 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 (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 deliver notifications to the subscribed MEC applications in a timely manner, without "overspending" CPU resources.

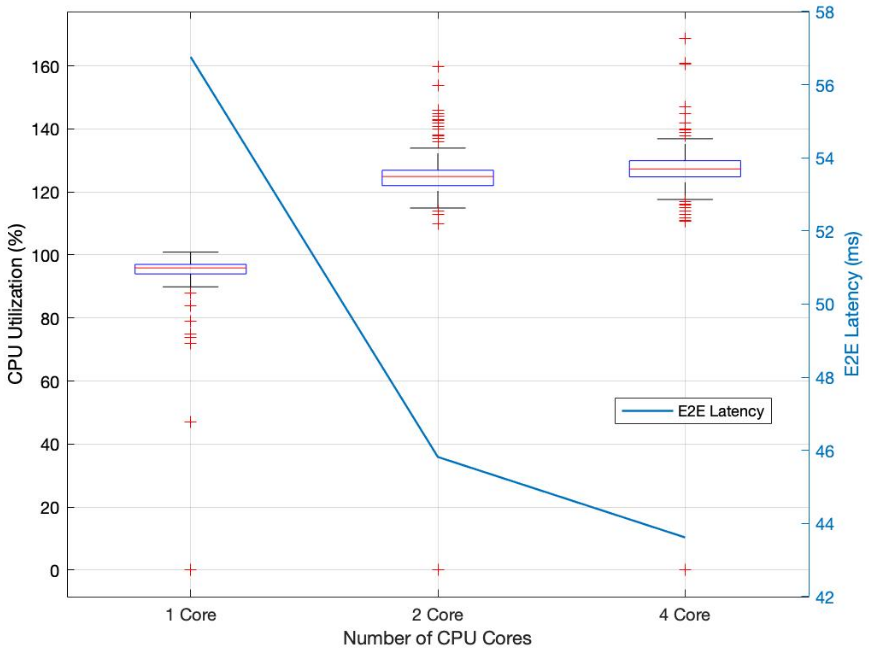


Figure 8: Apache Kafka: Box plot for CPU utilization vs. E2E latency by increasing number of CPU cores. Message rate: 10/s. Number of subscribers: 50; each of them has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 500,000.

6.4 Discussion

The message generation/publication rate for all the test cases was one message every 100 ms. For this rate, the latency of both the brokers was below 50 ms, which makes both of them appropriate for some real-time applications. Comparing Apache Kafka and RabbitMQ, the number of Kafka consumer instances per

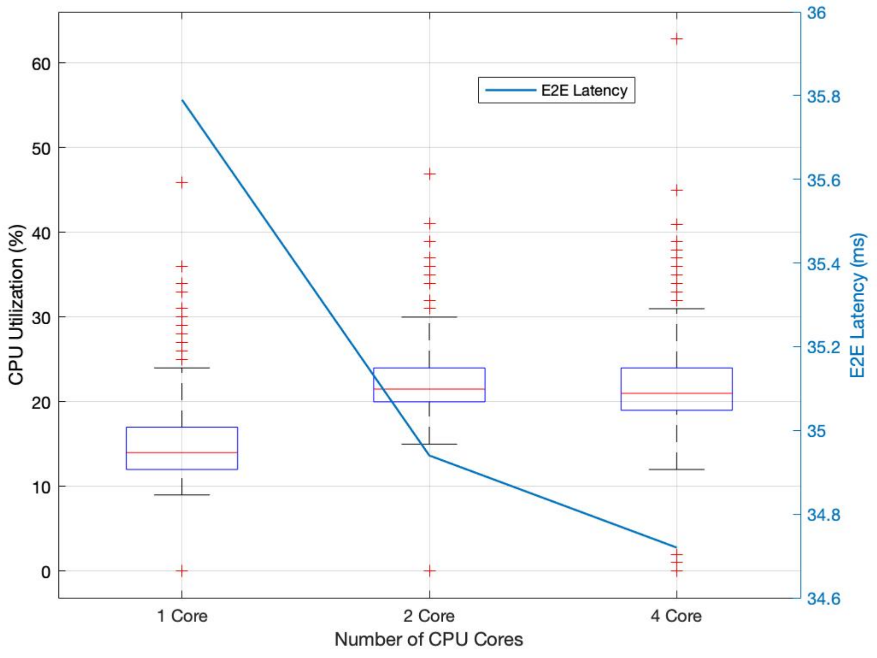


Figure 9: RabbitMQ: Box plot for CPU utilization vs. E2E latency by increasing number of CPU cores. Message rate: 10/s. Number of subscribers: 50; each of them has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 500,000.

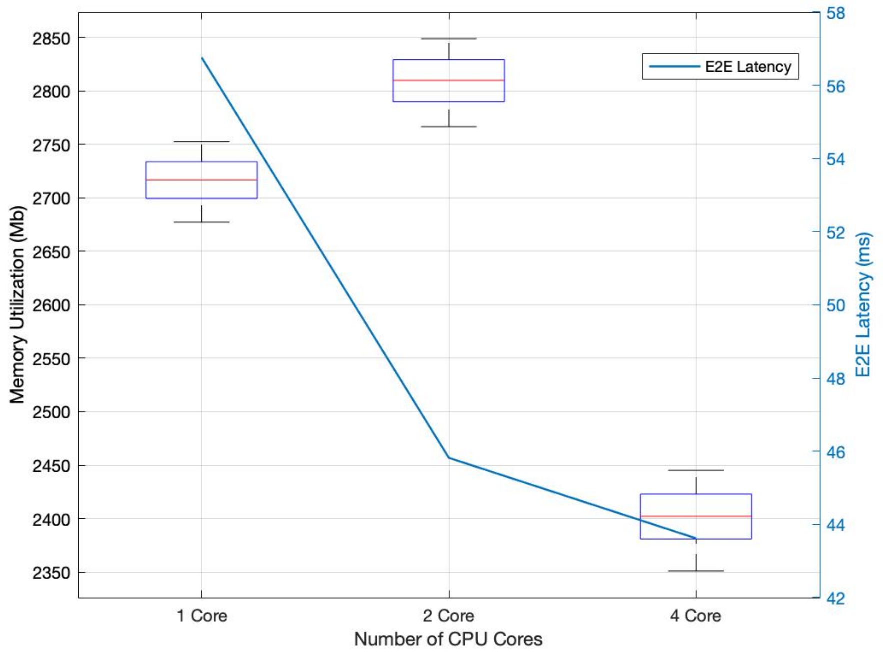


Figure 10: Apache Kafka: Box plot for memory utilization vs. E2E latency by increasing number of CPU cores. Message rate: 10/s. Number of subscribers: 50; each of them has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 500,000.

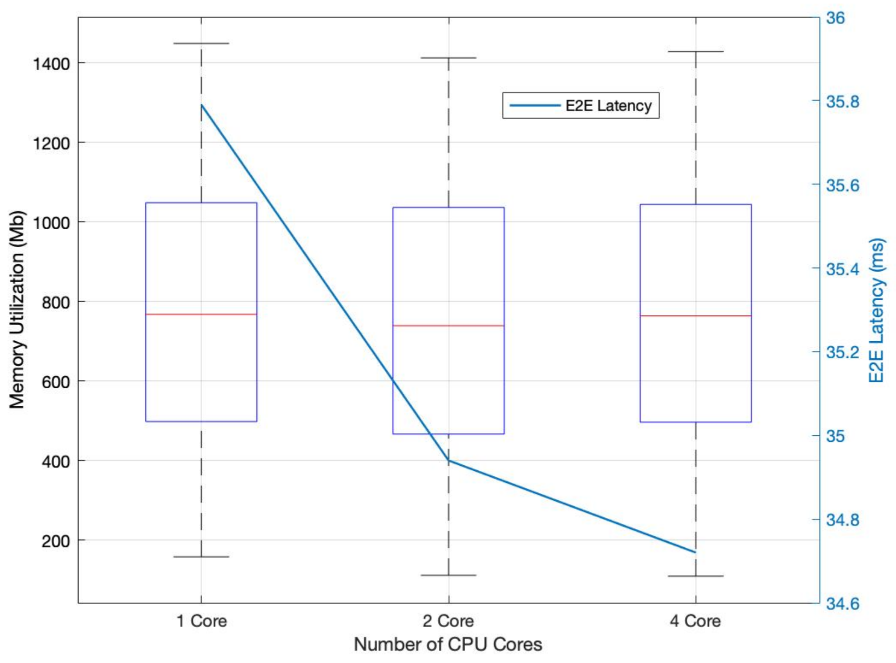


Figure 11: RabbitMQ: Box plot for memory utilization vs. E2E latency by increasing number of CPU cores. Message rate 10/s. Number of subscribers: 50; each of them has subscribed to 8 notifications. Total messages produced: 10,000. Messages consumed: 500,000.

subscribers is increased in comparison with the case for RabbitMQ, where there is one consumer instance per subscriber. The reason for a greater number of consumer instances in Kafka is due to the fact that, 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 heavy CPU and memory utilization.

In Kafka, the consumer implementation is based on the *pull* model and in RabbitMQ it is based on the *push* model. Therefore, if many consumer instances are polling simultaneously then the broker has to maintain offsets for all the consumers. This increases the processing load on the broker.

In summary, for implementing the publish-subscribe mechanism for our OAI-based RNIS, RabbitMQ appears to be a better option. It utilizes less resources, while having better latency performance compared with Kafka. If more RabbitMQ consumer instances are used for a subscriber, then the latency can be lower down at the expense of more resource utilization. We should note that experimenting with very high message rates to verify that the same behavior is observed is part of our ongoing work.

7 Conclusion

In this paper, we presented the implementation of a standards-compliant RNIS service on top of an OAI-based MEC platform, and presented extensive experimental results on its performance. We targetted a MEC-in-NFV environment, as recently introduced by ETSI, where MEC platform components, including the RNIS, are to be executed on top of an edge Network Functions Virtualization Infrastructure (NFVI), without excluding the case for multiple virtual RNIS instances coexisting, each belonging to a different tenant and being authorized to expose different subsets of RAN-level information. In such settings, 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. By using a MEC platform based on OpenAirInterface which we have developed, we compared the performance of two well-known message brokers (i.e., RabbitMQ and Kafka) for the implementation of the standards-based publish-subscribe message delivery functionality of the RNIS. The objectives were to assess the performance mainly in terms of latency, in the spirit of low-latency communication as enabled by edge computing. At the same time, we evaluated performance in terms of CPU and memory utilization, which represent the bottleneck of any virtualized system. The obtained results advocate for the use of RabbitMQ, as it is 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] C. I, Y. Liu, S. Han, S. Wang, and G. Liu, "On big data analytics for greener and softer RAN," *IEEE Access*, vol. 3, pp. 3068–3075, 2015.

[4] 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. 23rd IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 2018.

[5] C. Ge, N. Wang, S. Skillman, G. Foster, and Y. Cao, "QoE-Driven DASH Video Caching and Adaptation at 5G Mobile Edge," in *Proc. 3rd ACM Conference on Information-Centric Networking (ICN '16)*, 2016, pp. 237–242.

[6] S. Peng, J. O. Fajardo, P. S. Khodashenas, B. Blanco, F. Liberal, C. Ruiz, C. Turyagyenda, M. Wilson, and S. Vadgama, "QoE-Oriented Mobile Edge Service Management Leveraging SDN and NFV," *Mobile Information Systems*, vol. 2017, 2017.

[7] 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.

[8] 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 *2018 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, April 2018, pp. 179–184.

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

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

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

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

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

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

[15] 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

[16] 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.

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