FlexApp: Flexible and Low-Latency xApp Framework for RAN Intelligent Controller

Chieh-Chun Chen*, Mikel Irazabal†, Chia-Yu Chang‡, Alireza Mohammadi*, Navid Nikaein*
*EURECOM, Sophia-Antipolis, France, Email: {chieh-chun.chen, alireza.mohammadi, navid.nikaein}@eurecom.fr
†OpenAirInterface, Sophia-Antipolis, France, email: mikel.irazabal@openairinterface.org
‡Nokia Bell Labs, Antwerp, Belgium, email: chia-yu.chang@nokia-bell-labs.com

***Abstract*—RAN openness is a vision for 5G and beyond to avoid unnecessary lock-in effects. In this regard, the O-RAN alliance provides a new architecture with RAN intelligent controller (RIC) for both non-real-time and near-real-time cases, together with running applications, i.e., rApps and xApps. However, two key challenges remain in the current xApp framework: platform lock-in and xApp reusability. Therefore, we introduce a novel FlexApp framework to address both issues, as well as a new E2* interface that consumes less latency and CPU utilization. This new interface speeds up the xApp development process. Our performance evaluation of the FlexApp prototype shows that it can realize scalability and ultra-low latency operations ($<$10 ms), as well as the capability of two-level abstraction for xApp development.**

I. Introduction

The roll-out of 5G elevates user experience among distinct aspects and brings new opportunities to reshape mobile networks. Specifically, some key features, such as flexibility, openness, and intelligence, are expected to evolve monolithic infrastructures in legacy Radio Access Network (RAN). In this regard, standardization development organizations and industry fora, such as O-RAN Alliance and Telecom Infra Project, are formed to not only concretize this vision into requirements, but also standardize open network interfaces among disaggregated RAN entities (e.g., Distributed Unit [DU] and Centralized Unit [CU]). Therefore, "*Open RAN*" is made possible with adequate programmability and extensibility.

Moreover, RAN Intelligent Controllers (RICs) are highlighted in the O-RAN architecture in Fig. 1 to offer the Software-Defined RAN (SD-RAN) capability for both Near-Real-Time (NearRT-RIC) and Non-Real-Time (NonRT-RIC) cases. The NonRT-RIC [1] is within service management and orchestration framework, and it performs a control loop larger than 1 sec. It relies on data transmitted over the A1 interface [1, ch. 7], like policy guidance (e.g., slice priority) or enrichment information (e.g., application-level metrics). While the NearRT-RIC [2] handles control loops between 10 ms and 1 sec over RAN entities (CU and DU) via the E2-Node, which logically terminates the E2 interface [3]. The O1 interface [1, ch. 7] serves the purposes of operations, administration, and management and is beyond our scope.

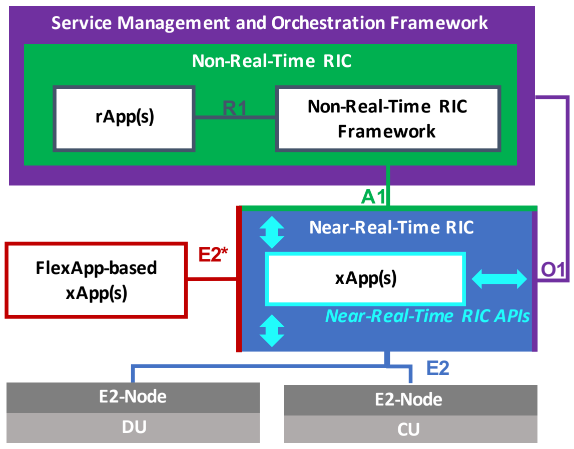


Fig. 1: O-RAN architecture with the proposed FlexApp framework and E2* interface.

To go one more step, two types of applications are utilized: rApps [2, ch. 4] running on NonRT-RIC and xApps [2, ch. 6] running on NearRT-RIC. The rApps provide value-added services related to RAN operation and optimization (e.g., energy saving) using the REpresentational State Transfer (REST)-based protocol over the R1 interface [1, ch. 7] to the NonRT-RIC. While the xApps utilize the E2 interface based on the Service Models (SMs) [4] to interact with RAN Functions (RFs), which provide controllable RAN functionalities in an E2-Node. For example, Key Performance Measurement (KPM) SM provides RAN statistics and performance metrics.

Beyond the above recap, two key challenges remain. First is the interoperability of xApps across different NearRT-RIC platforms. Due to the lack of a standardized interface between NearRT-RIC and xApps, the current xApp uses a platform-specific NearRT-RIC Application Programmable Interface (API) [2, ch. 7] to communicate with NearRT-RIC (i.e., light blue arrows in Fig. 1), which clearly violates Open RAN vision. Second, the scalability of xApp deployments is an issue due to the latency and overhead incurred by the NearRT-RIC API.

To make matters worse, all messages sent to the xApp must first go through NearRT-RIC and then be exposed via the NearRT-RIC API to the embedded xApp.

To address these challenges, we propose the FlexApp framework, which will not only provide interoperability for any native or third-party xApps, but also enable ultra-low latency operations ($<$10 ms). This framework relies on our newly-designed **E2* interface** between NearRT-RIC and xApps (i.e., dark red lines in Fig. 1), which can be viewed as a natural extension of the E2 interface and is considered as a potential candidate for standardization. Further, this framework enables the virtualization of RFs recursively to produce high-level abstraction and trigger future intent-based networking.

In the following, we first compare the FlexApp framework with other open-source works in Section II. An overview of the proposed framework is presented in Section III, and its design details are given in Section IV. Our FlexApp prototype in Section V demonstrates its low-latency in a scaled deployment and two-level abstraction ability for recursive xApp deployment.

II. Related works

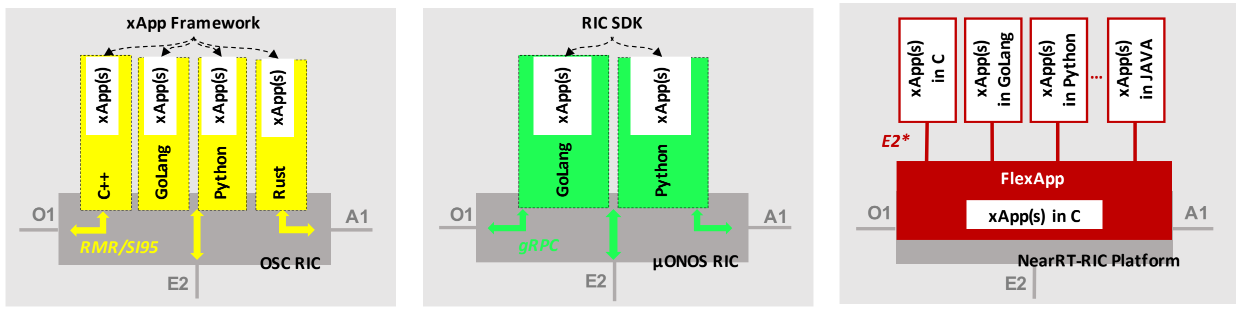
To avoid lock-in with a closed and/or proprietary xApp framework, we compare the FlexApp framework to two others in the open-source community: one developed by the O-RAN Software Community (OSC) and the other by the Open Networking Foundation (ONF), both on their respective NearRT-RIC platforms.

The OSC presents its xApp framework [5] (see Fig. 2a) on its own NearRT-RIC platform (OSC RIC [6]) and introduces the RIC Message Router (RMR) to communicate with the xApps through its Software Development Kit (SDK). The RMR is a library that abstracts the message transport mechanism (e.g., Nanomsg, Nanomsg next generation, or Socket Interface-95 [SI95] [7]) to be used by xApps to send/receive messages to/from an E2-Node. The message routing and endpoint selection in RMR are based on a pair of message type and subscription ID, which need to be translated into the endpoint via a table entry. A non-negligible overhead and latency are observable when this framework is employed due to the process of selecting an endpoint for each message [8].

Moreover, the ONF introduces their RIC SDKs (see Fig. 2b) to develop xApps on their own NearRT-RIC platform (µONOS RIC) [9]. These xApps communicate with the µONOS RIC using gRPC-based APIs, which provide HTTP request/response communication. Such an SDK encapsulates some of the complexities of dealing with individual gRPC services (e.g., threading and session management) to offer routing capabilities, and it requires xApps to provide the endpoint address (e.g., DNS server hostname, IP address, and port number) [9]. Despite claims that gRPC is faster than other REST-like transaction implementations, when compared to the RMR on top of SI95 (RMR/SI95) [10], gRPC cannot achieve the same throughput and with only acceptable latency when the message rate is less than 10,000 per second.

In contrast, the FlexApp framework (see Fig. 2) extends the standardized E2AP protocol as an E2* interface. This new interface can onboard the xApps into the NearRT-RIC and supports bidirectional communication between xApps and NearRT-RIC while inheriting the existing mechanism in the E2, i.e., Stream Control Transmission Protocol (SCTP) socket. One key aspect of this framework is to avoid platform lock-in; hence, the NearRT-RIC platform in Fig. 2 can be any O-RAN compliant RIC [2] (e.g., OSC RIC in Fig. 2a, µONOS RIC in Fig. 2b, or FlexRIC [11]). This framework also supports xApp development in a variety of high-level programming languages (for example, Python, GoLang, and Java).

Another advantage of this framework is that it enables xApps to specialize the desired set of SMs, effectively decoupling the SMs from the NearRT-RIC. Currently, different formats for a given SM are standardized while letting xApps manage SMs' internal data model (by subscribing to events and actions); therefore, an xApp



(a) OSC xApp framework (b) ONF RIC SDK (c) FlexApp framework

Fig. 2: Comparison of xApp framework from OSC, ONF and the proposed FlexApp

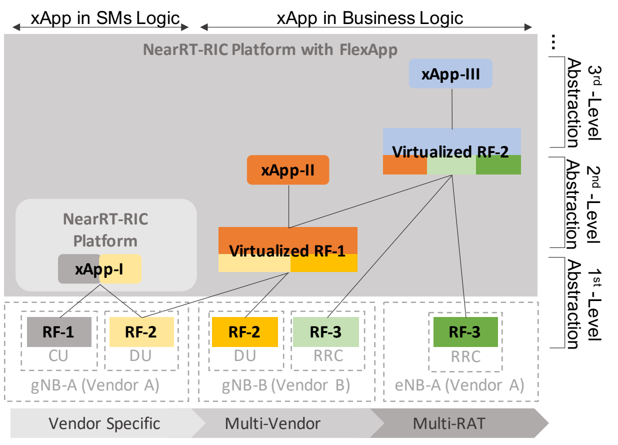


Fig. 3: High-level architecture of FlexApp framework with three xApp examples (xApp-I, xApp-II, xApp-III).

composes the desired SMs from different E2-Nodes and aggregates the data accordingly. In contrast, the FlexApp framework provides composability of RFs and SMs at the intermediate level by using the virtualized RF (vRF) to consolidate the data for xApp.

III. FlexApp Overview

As mentioned in Section I, the FlexApp framework is proposed to tackle both platform lock-in and xApp reusability challenges. A key approach adopted in FlexApp is to offer a novel set of abstractions for the virtualization of RFs to provide a high level of SM compatibility and composability, as depicted in Fig. 3. In fact, this approach can distinguish the xApp development model by relying on SMs' logic from that relying on business logic [2, ch. 7]. To provide more insight, three cases are presented in the same figure.

The first legacy case is xApp-I in Fig. 3, in which this xApp needs to handle different RFs (i.e., RF-1 and RF-2) through the corresponding SMs, data, and control actions and form a virtualized base station to process the data, even in a vendor-specific case. In contrast, the FlexApp framework provides the notion of vRFs to enable flexible SM composition based on the xApp requirements. Therefore, vRF-I in Fig. 3 is formed, which reveals a single SM abstracting data from the 2 RFs from different vendors to the xApp-II. Further, a higher level of abstraction can be built via recursive vRFs referencing, e.g., vRF-2 in Fig. 3 can handle the data from both RF-3 and vRF-1 among different Radio Access Technologies (RATs) to serve the purpose of xApp-III. Such virtualization allows the construction of new vRFs from existing RFs/vRFs as well as the recomposition of SMs, while still complying with E2 interface requirements.

As a concrete example, the Quality of Service (QoS) vRF can be composed by virtualizing other RFs that handle the data rate (e.g., by scheduling resource blocks) and latency (e.g., by managing queues). Furthermore, this vRF can also virtualize another vRF that handles service-specific metrics, such as RAN availability, according to the business logic in xApp.

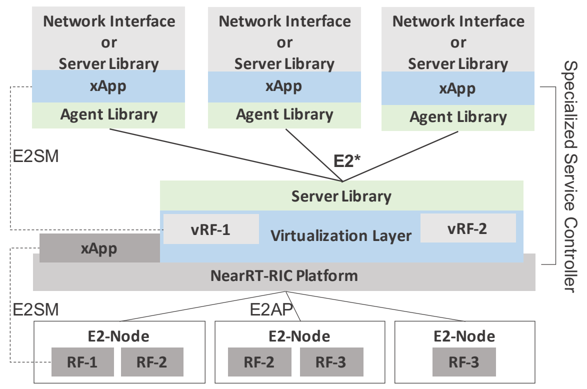
To conclude, our proposed FlexApp framework can benefit both RAN providers and xApp developers. On one hand, it simplifies the development of RFs and lowers the overhead on the E2-Nodes, since more advanced vRFs become composable via multi-level abstractions. On the other hand, it abstracts the heterogeneous RAN deployment and allows the newly composed vRF to be tailored to the desired xApp objectives, thereby simplifying the development of xApp. Furthermore, unlike the relationship between NearRT-RIC and xApps using platform-specific NearRT-RIC APIs (cf. Fig. 2a and Fig. 2b), xApps under the FlexApp framework are independent components. They have independent lifecycles and are treated as "first-class citizen applications" with a dedicated E2* interface to transfer the E2-based context for compatibility.

IV. System Design

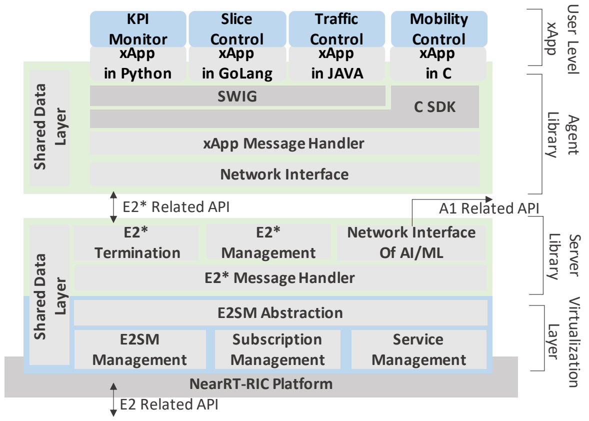
Generally, the FlexApp framework follows the zero-overhead principle to achieve low latency, and it includes four main elements, as shown in Fig. 4a: (1) E2* interface, (2) server library, (3) virtualization layer, and (4) agent library. From the viewpoint of the xApp, a *specialized service controller* is formed by utilizing these elements to realize its control logic. We can see a depiction of such a controller in Fig. 4b, and the design details of these elements are given as follows:

A. E2* interface

The E2* interface supports elementary procedures (e.g., E2* setup request/response) and services (e.g., report and control) similar to the E2AP protocol [3]. First, like the E2 interface, the E2* setup request message is initiated by the agent towards the NearRT-RIC, and, once successfully received, the NearRT-RIC assigns a particular xApp ID and responds with the information of connected E2-Nodes. However, unlike E2, the E2* subscription request message is generated by the agent to the NearRT-RIC, including the E2-Node(s) that this xApp would like to subscribe to. This message is bookkept (e.g., update the subscription request ID) and then forwarded to the corresponding E2-Node by the NearRT-RIC. A similar process is followed by a control request message. In total, four new elementary procedures are added to the E2 ASN.1 specification: E2* Setup, RIC Subscription, RIC Subscription Delete, and RIC control. Note that other encoding and decoding schemes (compared to the standard ASN.1) are supported in the E2* interface using *C11 _Generics*, which can alleviate existing bottlenecks at the CPU.



(a)



(b)

Fig. 4: (a) FlexApp framework design; (b) A specialized service controller based on the FlexApp framework
.

B. Server Library

The server library is essential for extending the NearRT-RIC with a virtualization layer that can aggregate vRFs from RFs/vRFs, as mentioned in Section III. Moreover, it manages the connection towards the agents and multiplexes messages between the virtualization layer and the agents by using the E2* termination, E2* management, and E2* message handler modules in Fig. 4b. Also, this library is designed as an event-driven/callback-driven system, adhering to the aforementioned ultra-lean design principle to impose minimal overhead. Therefore, it activates the virtualization layer only when there are new messages to be handled.

C. Virtualization Layer

The virtualization layer can implement new SMs on top of the newly composed vRF and expose specific information to xApps. From the xApp viewpoint, this layer is the key in the specialized service controller, and there are four modules in this layer (cf. Fig. 4b). First, the E2SM abstraction handles the vRFs and the respective SMs to expose the prepossessed information to the northbound xApp via the E2* interface. Moreover, the E2SM management module handles the information of the supported SMs in the NearRT-RIC. Then, the subscription management module is responsible for the subscription procedure and tracks the existing RF subscription statuses for each xApp. Finally, the service management module takes care of the interaction between xApps and (v)RFs by processing the messages related to the basic services specified in E2SM, including report, insert, control, and policy.more than one xApps sends conflicting messages to the same (v)RFs.

D. Agent Library and xApp development

From the perspective of xApp, the agent library is the cornerstone of realizing its logic because it is built on top of this library in the southbound. Furthermore, as shown in Fig. 4a, an xApp can recursively expand the interface at its northbound by reusing the server library (cf. Section IV-B). In detail, two building blocks exist in this library as a part of the SDKs: C SDK and SWIG[1] as shown in Fig. 4b. The C SDK refactors all common functionalities of an xApp (e.g., communication, message handling, encoding/decoding, and data layer) under a single C library to be exposed to xApps for the development of C/C++ based xApp. In contrast, as mentioned in Section II, to enable the xApp development using other high-level programming languages, the SWIG compiler is apoted to create the wrapper codes for the C-based library of E2/E2* interfaces.

To develop an xApp, the northbound SDK[2] is provided in the agent library functions including:

- `init_xapp_api` initializes the xApp by sending E2* setup request to the NearRT-RIC,
- `e2_nodes_xapp_api` can get the information of the connected E2-Nodes from the NearRT-RIC,
- `report_sm_xapp_api` enables the report service by subscribing to the RF with arguments, and
- `control_sm_xapp_api` can control the targeting RFs by specifying the input arguments

E. Summary

In short, the FlexApp framework provides several key takeaways: (a) Forward compatibility, such as dedicated controllers and xApps, (b) Ultra-lean design by extending E2AP without extra overhead, (c) A unified SDK exposing generic APIs for xApp development, and (d) Multilingual xApp development kit.

[1]https://www.swig.org/
[2]https://gitlab.eurecom.fr/mosaic5g/flexric/-/tree/master/examples/xApp/c/

V. Performance Evaluation

In this section, the performance of FlexApp framework is validated in three aspects, based on a prototype implemented on top of the O-RAN compliant FlexRIC [11]. First, we measure the End-to-End (E2E) latency from the E2-Node to the xApp under different numbers of E2-Nodes, which represents the capability to realize ultra-low latency operations even in a scaled deployment. Second, benchmarks are made between the proposed E2* interface[3] and two other transport mechanisms adopted by the OSC and the ONF, i.e., gRPC and RMR/SI95 (cf. Fig. 2). Finally, the efficiency and effectiveness of the xApp control logic are verified after applying the virtualization layer for multi-level abstraction in the FlexApp framework.

Furthermore, as shown in TABLE I, the experiments are evaluated in two setup environments: containerized (cloud-native) and non-containerized (bare metal). To represent the RAN functionalities and perform the report service, we use the E2-emulator provided by the FlexRIC. Within each E2-emulator, the Medium Access Control (MAC) RF is enabled, and the corresponding data is encapsulated into the MAC SM. Then, our self-developed xApp subscribes to the MAC SM with a particular periodicity, and the composed E2SM will be periodically updated by the E2-emulator using E2AP and sent to the xApp using the E2* interface.

TABLE I: Experimental environment.

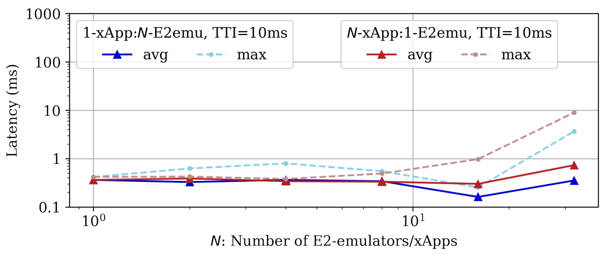
	Cloud-Native	**Bare Metal**
Ubuntu	20.04.5	20.04.1
CPU(s)	4.8GHz x 20	4.7GHz x 12
Memory	64 GiB	32 GiB
Kubernetes	v1.24.5	-
Containerd	v1.6.8	-

A. Latency and Scalability

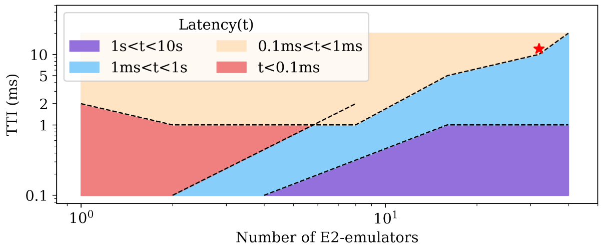
In this evaluation, three Precision Timing Protocol (PTP) synchronized machines are used to deploy the E2-emulator, FlexRIC, and xApp. We cover both containerized and non-containerized environments. The xApp subscribes to the MAC SM and obtains the monitoring data from the E2-emulator(s) via the E2* interface for every Transmission Time Interval (TTI) in a 12-byte message format (including E2AP & E2SM protocols). Two relationships between xApps and E2-emulator are evaluated: 1-to-N and N-to-1.

As shown in Fig. 5a, both the average and the maximum values of E2E latency are presented for 10 ms TTI in a containerized environment. Therefore, within 1 sec, there are 100 messages to be processed for each connected E2-emulator or xApp. The average

[3]In the FlexRIC implementation, the E2* is labelled as E42 interface.



(a) E2E latency between xApp(s) and E2-emulator(s)



(b) Feasible regions of different latency restrictions

Fig. 5: Latency and scalability evaluation.

E2E latency remains constant when the number of E2-emulators or xApps is low (i.e., 8); however, its values increase when there are either 32 E2-emulators[4] or 32 xApps, but remain below 1 ms for ultra-low latency operations. A similar trend is observed in the maximum E2E latency, where the values are all lower than 10 ms. This guarantees that the FlexApp approach will not produce an internal bottleneck under a 10 ms TTI.

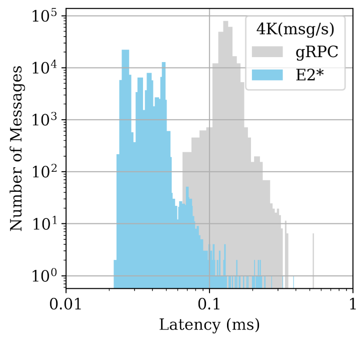
Based on the results shown in Fig. 5a, we can conclude that the FlexApp framework can support 32 xApps using its northbound to monitor SM from an E2-Node, and an xApp can simultaneously monitor 32 E2-Nodes connected to the NearRT-RIC. In parallel, we perform a stress test on the FlexApp framework, which shows that up to 350 xApps could be supported per machine, and the average E2E latency is around 6 ms. This limitation is due to the thread mode implemented in our prototype and the way messages are enqueued. We plan to make improvements in the future.

In addition, we measure the E2E latency in a non-containerized environment to plot the viable regions for different latency restrictions, as shown in Fig. 5b, in terms of various combinations of TTI values and the number of E2-emulators. Take the example case with 1 ms latency restriction, its viable region covers both the light orange and the light red parts in Fig. 5b, since these two parts can support latency below 1 ms. We can see a tradeoff between how frequently the data can be updated and how many E2-Nodes can be monitored simultaneously, e.g., shorter TTI values must come with

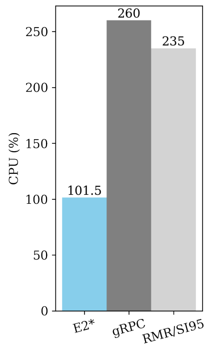
[4]The maximum number of supported E2-emulators is currently limited by FlexRIC; hence, we only evaluate up to 32 E2-emulators.

TABLE II: Latency comparison in milliseconds.

	Min	**Max**	**50%**	**95%**	**99%**	**Total Sent**	**Rate (msg/s)**
gRPC	0.07	>1.0	0.15	0.18	0.20	100K	4K
E2*	0.02	0.39	0.11	0.11	0.11	100K	4K
gRPC	0.20	0.54	0.31	0.37	0.39	25K	0.1K
E2*	0.03	0.21	0.06	0.06	0.06	25K	0.1K



(a) Latency comparison: 100K messages sent at 4K rate (msg/s)



(b) CPU comparison

Fig. 6: Benchmarking on transport mechanisms.

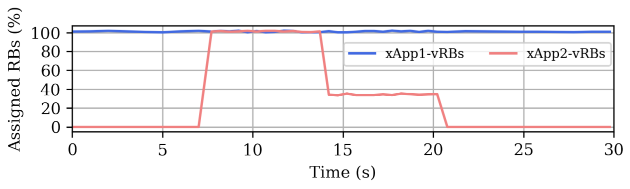
fewer monitored E2-Nodes. Also, the red star in the same figure represents the setting used in Fig. 5a, i.e., 10 ms TTI and 32 E2-Nodes, and this setting is confirmed to be functional for 1 ms latency restriction.

B. Benchmarking on transport mechanism

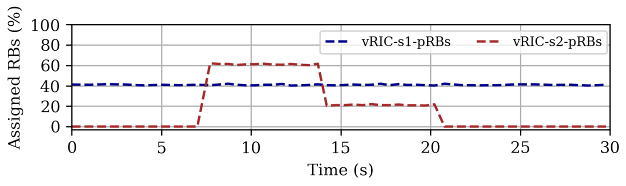
Afterwards, different transport mechanisms between NearRT-RIC and xApp are compared. Since our focus is on the transport mechanism, a simple 1-to-1 relationship between xApp and FlexRIC is applied (similar to [10]), but with two different TTI values, i.e., 250 µs and 10 ms. Specifically, an xApp is used to measure the latency by calculating the time difference from when the indication message is forwarded by the NearRT-RIC to when it arrives at xApp callback function. In Table II and Fig. 6a, both E2* interface and gRPC are compared at different message rates, and the E2* interface outperforms at every percentile value.

Furthermore, in Fig. 6b, we compare the CPU utilization of the xApp to receive the transported messages among the E2* interface, the gRPC, and the RMR/SI95. We can see that the E2* interface consumes considerably less CPU resources (i.e., 101.5% compared with 260% and 235%) than the other two.

In summary, by design, the E2* interface can reach a lower latency than the gRPC because it performs fewer work/abstractions, e.g., a raw SCTP connection in the E2* interface versus an HTTP-based approach in the gRPC. Moreover, the E2* interface consumes less CPU resource at the xApp owing to the simpler FlexApp architecture and fewer messages exchanged between xApp and NearRT-RIC. To provide more details, the xApp using RMR/SI95 must handle the message from the NearRT-RIC for routing management before handling the message, and the message must be encoded/decoded several times between each micro-service.



(a) Percentage of virtual RBs from xApps' perspective



(b) Percentage of physical RBs from RIC's perspective

Fig. 7: Two-level abstraction.

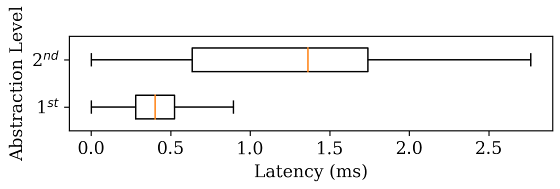


Fig. 8: Control loop round-trip-time.

C. Two-level abstraction and Control loop RTT

To validate the multi-level abstraction approach described in Section III, we build a virtualization layer (see Fig. 4b) in the NearRT-RIC platform and test its ability to interact with two separate slicing control xApps. Based on the respective service requests from these xApps, radio resources from the E2-emulator, i.e., 162 Resource Blocks (RBs) in a 60 MHz bandwidth with numerology 1 can achieve up to 200 Mbps, will be assigned accordingly.

Moreover, to compare different perspectives after and before the two-level abstractions (see xApp-II in Fig. 3), two respective figures are presented in Fig. 7a and Fig. 7b. The former shows the virtual RBs assigned to the xApp by the virtualization layer, whereas the latter represents the physical RBs seen from the NearRT-RIC perspective. Initially, only xApp1 is present, and it needs 80 Mbps for its service. Therefore, only 40% of the physical RBs are requested by the virtualization layer from the E2-emulator (see vRIC-s1-pRBs in Fig. 7b). But from xApp1's point of view, it assumes it receives 100% of the virtual RBs because of the second-level abstraction (see xApp1-vRBs in Fig. 7a). At the 7th second, xApp2 is deployed and requests 120 Mbps instead. In this regard, the virtualization layer requests 60% of the physical RBs from the E2-emulator (see vRIC-s2-pRBs in Fig. 7b), but xApp2 receives 100% of the virtual RBs (see xApp2-vRBs in Fig. 7a), similar to xApp1. Then, at the 14th second, xApp2 lowers its request to 40 Mbps, and thus the corresponding decreases by a factor of three are made in both Fig. 7a and Fig. 7b. Finally, xApp2 releases its request at the 21st second but has no effect on xApp1.

Furthermore, we measure the control loop Round-Trip-Time (RTT) between the E2-emulator and the xApp after considering the virtualization layer as the second-level abstraction. In specific, the measured RTT as shown in Fig. 8 represents the time difference from when the xApp sends a control message until it receives the control acknowledgement. Compared with the measurements conducted in Section V-A, these results are more informative in terms of considering the second-level abstraction as well as the RTT statistics for the control loop. We can see that the RTT after considering the second-level abstraction is approximately three times higher than that of the first-level abstraction case. However, it still fulfills the requirements for control loops in NearRT-RIC (between 10 ms and 1 sec), implying that extra-level abstractions (e.g., third-level abstraction xApp-III in Fig. 3) are potentially feasible.

D. Summary

In this section, three aspects of the performance evaluation are conducted for our FlexApp prototype. These results prove that the FlexApp framework is not only suitable for inclusion in the current NearRT-RIC platform, but also outperforms other xApp frameworks in terms of better performance (i.e., lower latency, less CPU usage) and newer functionalities (i.e., two-level abstraction). In addition, we provide informative results (e.g., feasible region in Fig. 5b and control loop RTT in Fig. 8), which can be used for further analysis of the different xApp deployment possibilities. In summary, the FlexApp framework enables an xApp to perform ultra-low latency operations thanks to the new E2* interface and is built on the virtualization layer, abstracting the raw data from the SM logic to the business logic to simplify the xApp development.

VI. Conclusion and Future directions

In this paper, we presented FlexApp, an xApp framework that supports latency-sensitive use cases and enables multi-level abstractions, as well as a future standardized E2* interface, to communicate between xApp and NearRT-RIC. Additionally, by design, it is compatible with other NearRT-RIC platforms. The performance evaluation of our FlexApp prototype was thoroughly conducted, and the results indicated its benefits include low latency, high scalability, slim overhead, and multi-level abstraction functionality.

In the future, we plan to extend this work in three directions. First, related to the FlexApp framework, we plan to enhance its scalability in its northbound and exploit its virtualization layer to abstract a heterogeneous and multi-X (e.g., multi-tier, multi-RAT) RAN. Then, as for the xApp, we will extend its northbound interface to expand its flexible and recursive relationship toward the NearRT-RIC and explore new xApps (e.g., interference management) for RAN optimization and coordination. Finally, we consider porting existing xApps (e.g., traffic steering) to enrich FlexApp compatibility and interoperability between different NearRT-RIC platforms.

Acknowledgments

This work is funded by the European Commission Horizon 2020 5G-PPP Program under Grant Agreement Number H2020-ICT-2020-2/101017226 “6G BRAINS: Bringing Reinforcement learning Into Radio Light Network for Massive Connections”.

References

[1] O-RAN Working Group 3, “Near-Real-time RAN Intelligent Controller, Near-RT RIC Architecture,” Tech. Rep., 2022, Technical Specification O-RAN.WG3.RICARCH-v02.01.

[2] O-RAN Working Group 2, “Non-RT RIC Architecture,” Tech. Rep., 2022, Technical Specification O-RAN.WG2.Non-RT-RIC-ARCH-TS-v02.00.

[3] O-RAN Working Group 3, “Near-Real-time RAN Intelligent Controller Architecture E2 General Aspects and Principles,” Tech. Rep., 2022, Technical Specification O-RAN.WG3.E2GAP-v02.02.

[4] ——, “Near-Real-time RAN Intelligent Controller, E2 Service Model (E2SM),” Tech. Rep., 2022, Technical Specification O-RAN.WG3.E2SM-v02.01.

[5] O-RAN Software Community. xApp Writer’s Guide v2. [Online]. Available: https://wiki.o-ran-sc.org/display/RICP/Introduction+and+guides

[6] ——. RIC platform source code. [Online]. Available: https://gerrit.o-ran-sc.org/r/admin/repos/q/filter:ric-plt

[7] ——. RMR Developer’s Guide. [Online]. Available: https://docs.o-ran-sc.org/projects/o-ran-sc-ric-plt-lib-rmr/en/latest/developer-guide.html

[8] ——. RMR vs NNG Sending Performance. [Online]. Available: https://wiki.o-ran-sc.org/display/RICP/RMR_nng_perf

[9] Open Networking Foundation. µONOS RIC architecture. [Online]. Available: https://docs.sd-ran.org/master/architecture.html

[10] O-RAN Software Community. Suitability of gRPC for RMR Communications. [Online]. Available: https://wiki.o-ran-sc.org/display/RICP/gRPC+Evaluation

[11] R. Schmidt, M. Irazabal, and N. Nikaein, “FlexRIC: An SDK for next-generation SD-RANs,” in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 411–425. [Online]. Available: https://doi.org/10.1145/3485983.3494870