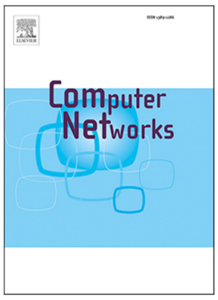


Enabling realistic experimentation of disaggregated RAN: Design, implementation, and validation of a modular multi-split eNodeB

Cristian C. Erazo-Agredo, Luis Diez, Ramón Agüero,
Mario Garza-Fabre, Javier Rubio-Loyola

PII:	S1389-1286(23)00438-3
DOI:	https://doi.org/10.1016/j.comnet.2023.109993
Reference:	COMPNW 109993
To appear in:	*Computer Networks*
Received date :	16 April 2023
Revised date :	21 July 2023
Accepted date :	21 August 2023

Enabling realistic experimentation of disaggregated RAN: design, implementation, and validation of a modular multi-split eNodeB

Cristian C. Erazo-Agredo[a], Luis Diez[b], Ramón Agüero[b], Mario Garza-Fabre[a], Javier Rubio-Loyola[a]

[a]*Center for Research and Advanced Studies, Cinvestav Campus Tamaulipas, Ciudad Victoria, 87130, Tamaulipas, Mexico*
[b]*Dept. of Communications Engineering, Universidad de Cantabria, Santander, 39005, Spain*

Abstract

Software-defined networking and network function virtualization are the key enablers in the division of base station functionalities and the deployment of some of these in central nodes. In this way, eNodeB tasks, traditionally deployed at dedicated nodes, are distributed between Central Units (CU) and Distributed Units (DU), which are closer to the Radio Units (RU). The 3GPP has specified eight possible functional splits between the CU and the DU. Despite their importance, to date, there is a lack of practical approaches that enable the real separation of multiple functional splits into different nodes, which would in turn enable the adaptation of the functional split configuration, according to network conditions and traffic load. This paper presents the design and implementation of an experimental architecture that allows the actual separation of protocol layers into different nodes, enabling the experimentation with up to four functional split configurations. The validation results demonstrate the feasibility of the proposed implementation, which copes with the delay requirements established by the Small Cell Forum in all configurations.

Keywords: eNodeB disaggregation, functional split, practical implementation

1. Introduction

One of the cornerstones of 5G and 6G networks is the architectural shift that, exploiting Software Defined Networking (SDN) and Network Function Virtualization (NFV) techniques, divides the functionalities that were traditionally attached to the base station. This new paradigm moves some of such functions to central nodes, usually deployed at cloud servers, leading to the so-called Cloud-RAN (CRAN) or, more recently, Virtual RAN (VRAN).

The above approach brings several advantages, such as the reduction of costs (both deployment and operational), and the possibility to leverage tighter cooperation between access elements. On the other hand, it also comes with several challenges that should be tackled, being particularly relevant the need to comply with the strict delay requirements of certain services.

Opposed to legacy base stations, which mostly comprise dedicated hardware elements, functions are now divided into Central Units (*CU*) and Distributed Units (*DU*). The former are deployed in the cloud, exploiting virtualization techniques, while the latter are closer to the Radio Units (*RUs*). Depending on which functions are deployed at each of these elements, there are different possible configurations or functional splits. In this sense, Figure 1 illustrates the ones that have been specified by 3GPP, where the *CU* is at the left side of the figure, and the *DU* is at the right. Configurations with a higher index (see Figure 1) place more functions at the *CU*, leading to a greater centralization level, which imposes more strict delay requirements to the fronthaul network, which connects *DU* and *CU*.

This new paradigm opens various research problems and challenges. One initial aspect that needs to be tackled, related to network planning, is the placement of *CUs* and the consequent connections to *DUs*. This can be done in combination with the establishment of the particular configuration (functional split) or centralization level, i.e., specifying which functions are executed at the *CU* and at the *DU*. There are some works that have addressed these problems, both individually and jointly.

Furthermore, there exists the possibility of dynamically changing the functional split, according to the particular network conditions and traffic load. The so-called flexible functional split thus affects the network operation and its management. There are few papers that have explored the potential benefits of this approach, or proposed models to assess its performance.

In this work, we take a different approach, which to our best knowledge has not been explored before. Our aim is to develop a testbed, for lab environments, which enables all the functional splits that separate different protocol entities: O_1, O_2, O_4, O_6 (cf. Figure 1). We use the srsRAN 4G implementation, which can be deployed using Software Defined Radio (SDR) devices, and so the O_8 functional split is intrinsically available. We modify the implementation using wrappers, which keep the original behavior of the classes that conform the srsRAN implementation. We first evaluate the validity of the proposed modifications, and we then thoroughly assess its performance, both in terms of delay, to identify the configurations that are feasible, and computational effort (in terms of CPU load). All the implementation presented in this paper has been made available at a public git repository[1]. Although the developed solution is based on the 4G stack, the adopted design could be straightforwardly

[1]`https://github.com/tlmat-unican/fd-srsran`

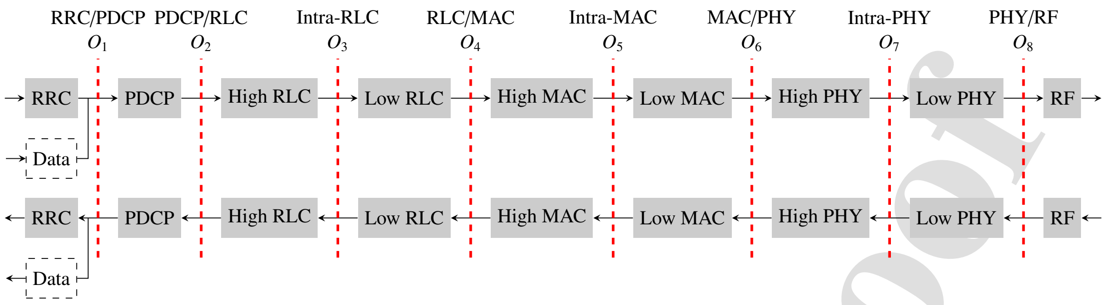


Figure 1: Possible functional splits between the *CU* and *DU* according to 3GPP.

adapted to the 5G stack. In fact, we took the 4G project as our starting point, since there was not a alternative solution for 5G when we began the development tasks.

The paper is structured as follows. Section 2 discusses related work on CRAN and functional split. Section 3 presents the design and implementation of the proposed architecture, which is then validated and evaluated in Section 4. We conclude the paper in Section 5, which provides an outlook of our future work.

2. State of the art

Although the relevance of functional split, and gNodeB disaggregation strategies, has recently grown quite strongly, related works on detailed implementations that enable gNodeB functional split are quite scarce to date. This section gathers the implementations available in the state of the art.

The authors of [1] introduce a proof-of-concept for functional splits 2 and 5 over a TDM-PON-based CRAN, which uses the Mobile-Central Office re-factored as a Datacenter (M-CORD). The authors present results with such split configurations, which are compared using the reliable and non-reliable transport protocols, TCP and UDP, respectively for both uplink (UL) and downlink (DL). The authors found that split 5 provides larger throughput than split 2.

Alfadhli et al. discuss in [2] an analysis of the latency for Option-7, Option-8, and Option-9 functional splits. The authors demonstrate that the latter, for the fronthaul interface design, tends to increase the statistical multiplexing gain, allowing the *RU* to use the simplest structure with the lowest fronthaul latency, as well as almost removing the interface delay at the fronthaul, while greatly reducing the optical conversion delay. In fact, the authors claim to have experimentally demonstrated that Option-9 can support lower latency at the fronthaul, thus allowing longer fronthaul link lengths in integrated fiber-wireless access networks.

Younis et al. propose in [3] a task offloading solution for Mobile Edge Computing (MEC) environments. The solution is formulated as an optimization problem that is proven to be NP-hard. Then it is afterwards changed into a linear one, which is finally solved using existing tools. Interestingly, the proposed solution is evaluated over a testbed based on Open Air Interface (OAI) framework, which already implements functional splits for RU/DU/CU disaggregation. Differently to our work, the authors in [3] use an existing implementation to evaluate a solution on top of it, but they do not develop or broaden the existing splits in OAI. Similarly, Matoussi et al. [4] also leverage the OAI framework to evaluate a flexible functional split algorithm that considers traffic and computational requirements, along with power consumption. The authors propose a solution based on Particle Swarm Optimization (PSO), which is first analytically assessed over a simulation environment, to be afterwards validated using the OAI framework. As it was the case of [3], Matoussi et al. do not aim to extend the adopted SDR solution, but to exploit an existing one. In this sense, the scope of our work is rather different, since we describe the implementation of new functional split features in an existing framework.

Martinez-Alba et al. introduce in [5] adaptive functional split mechanisms to switch from PDCP-RLC to MAC-PHY, or vice versa, at runtime, without interrupting user traffic. The proposed method uses a replication-based approach, with three variations: hard, soft, and custom. It is tested through experimental validation for packet losses and end-user latency. Experimental results demonstrate that hard migrations do not add a remarkable delay, but they are prone to induce packet losses. Moreover, soft migrations are suitable to reduce packet losses at the cost of introducing a delay of up to 15 *ms*. A trade-off between both key performance indicators is achieved with custom migrations, which show a linear relationship between latency and packet losses.

Another work of interest is [6], where Rodriguez et al. analyze and implement the 7.3 intra-PHY functional split in a C-RAN architecture, with the main goal of keeping the centralization benefits, while reducing the required fronthaul capacity. The 7.3 split configuration is then compared with the 7.1 and 7.2 functional splits. Experimental tests demonstrate that the 7.3 split implementation outperforms the other ones in terms of required fronthaul bandwidth and achievable distance between *RU* and *DU*, when considering suitable multithreading of (de)coding functions.

Furthermore, the authors of [7] present a prototype to assess the impact that flexible functional split might have on energy consumption. Other studies focus on the reconfiguration of the

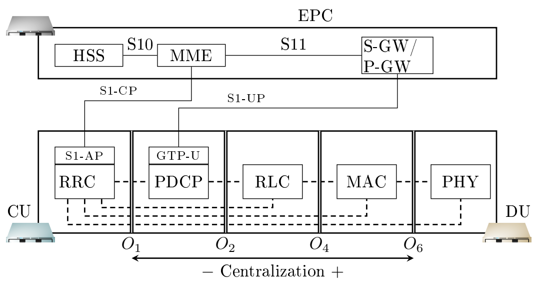


Figure 2: Functional description the implementation

underlying network so that it can comply with the strict requirements of the functional splits. For instance [8, 9] propose and assess solutions to reconfigure the fronthaul network, based on optical technology, while the authors of [10] use an Ethernet-based alternative. There exist some works that pay attention to the access element resources, like [11], where the authors propose F-RAN, an architecture that performs split selection according to the available radio resources.

Relevant stakeholders, including operators, vendors, and academia, under the umbrella of the Open Radio Access Network (O-RAN) Alliance, are, at the time of writing, fostering this network disaggregation paradigm, promoting flexible, virtualized and open RANs [12, 13]. They exploit the functional split approach established by the 3GPP, in particular Options 2 and 7.2, and they introduce the same functional elements: ORU, ODU, OCU (O-RAN RU, DU, and CU, respectively). Our work complements this initiative since we aim to promote a design that would enable using various functional splits. On the other hand, and as was already mentioned, our design and development have been conceived for testing and research, while the O-RAN Alliance focuses on networks in production.

3. Design and implementation

This section describes the implementation carried out to develop the fully disaggregated 4G base station (eNodeB). In general, we will refer to splits between *CU* and *DU* entities, while the *RU* is assumed to be a separate node, hosting only the radio interface. Nevertheless, some split configurations could actually represent a *DU-RU* separation. As mentioned in Section 1, the implementation described in this paper is intended for experimentation and research. In that sense, although the functional separation of the different entities is provided, the communication between them does not follow the standardized F1 interface.

The solution described in this paper is based on the srsRAN 4G implementation. In this sense, Figure 2 shows a general view of the final implementation, where dotted and solid lines correspond to implemented and existing interfaces, respectively. As can be observed, the final solution separates the main layers of the protocol stack, and the RRC entity from the rest of the stack. This way, the *CU* always hosts the RRC and the *DU* the PHY layer, while the particular placement of the rest of the stack can be configured.

The implementation described in this paper would allow configuring the following splits, all of them defined by the 3GPP [14]:

- *Option 1* (O_1), which places the RRC at the *CU*, and the rest of entities at the *DU*;
- *Option 2* (O_2), where the RRC and PDCP are located at the *CU*, and the *DU* hosts the other entities. This split is adopted by the O-RAN architecture [15] for the *CU-DU* separation;
- *Option 4* (O_4), which places the RRC, PDCP and RLC at the *CU*, and both MAC and PHY layers are moved to the *DU*; and
- *Option 6* (O_6), where only the PHY layer remains at the *DU*; this is fostered by the Small Cell Forum [16] for the *DU-RU* separation.

Our implementation does not limit the disaggregation into two entities, namely *CU/DU*, but it would allow instantiating the stack layers in a larger number of nodes. For instance, the current implementation would allow using the *CU-DU-RU* base station configuration proposed by the Small Cell Forum, where *Option 2* is used between the *CU* and *DU*, and *Option 6* separates *DU* and *RU*.

As can be observed, the proposed scheme adopts splits defined by different initiatives and extends the disaggregation options that are available for experimentation purposes. Existing solutions, such as the aforementioned OAI or the recent release (February 2023) of srsRAN with 5G stack, implement the options defined by O-RAN, although they are not configurable, so *CU* and *DU* nodes always deploy the same functions. As we elaborate in Section 5, we plan to tackle the combination of both approaches in our future research. In the work presented herewith we focus on the solution design and on its feasibility analysis. In this sense, we do not compare its performance with alternative solutions because: (1) it would require modifying such solutions to gather comparable metrics, and this is left for our future work; (2) some splits do not match, and the comparison may not thus be fair.

In the following, we first describe the software design that has been adopted to modify the existing SDR code, and we then depict the communication flow between the involved modules.

3.1. Software design

This section describes the object-oriented design adopted to implement all functional splits. As a main design principle, it aims to be as transparent as possible to the existing code, in order to ease its adaptation to new releases.

The eNodeB stack of the srsRAN[2] is implemented in C++, where each layer is represented by one class. The communication between classes is done by means of simple inheritance

[2]srsRAN project: `https://www.srslte.com/`

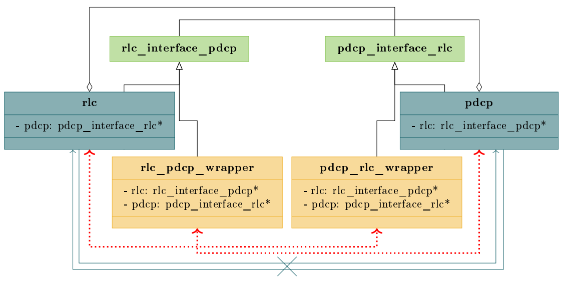

rlc_interface_pdcp
pdcp_interface_rlc
rlc
- pdcp: pdcp_interface_rlc*
pdcp
- rlc: rlc_interface_pdcp*
rlc_pdcp_wrapper
- rlc: rlc_interface_pdcp*
- pdcp: pdcp_interface_rlc*
pdcp_rlc_wrapper
- rlc: rlc_interface_pdcp*
- pdcp: pdcp_interface_rlc*

Figure 3: Example of the split implementation design applied to PDCP-RLC split (*option 2*).

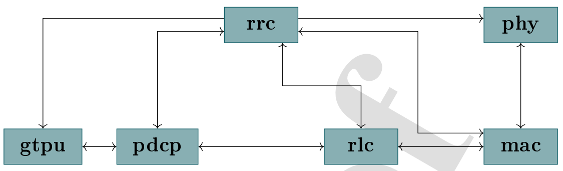
to implement interfaces, as exemplified in Figure 3, where the original classes are shown in different shades of green, and the modules that have been added to the implementation are yellowish. The figure also illustrates class inheritance (closed arrows), aggregation (diamond arrows), and simple association (open arrows).

As can be observed, the `rlc` layer class inherits, in the original implementation, from the `rlc_interface_pdcp` interface (read as RLC interface for PDCP), defined as an abstract class. In turn, the `pdcp` class is initialized with a pointer of such interface, to communicate with the `rlc` instance when needed, thus abstracting the actual RLC implementation from how it is used. Similarly, `rlc` class is initialized with a pointer of the `pdcp_interface_rlc`, to communicate with `pdcp`. The figure also shows an association between `pdcp` and `rlc` classes, to indicate that they would eventually interact with each other.

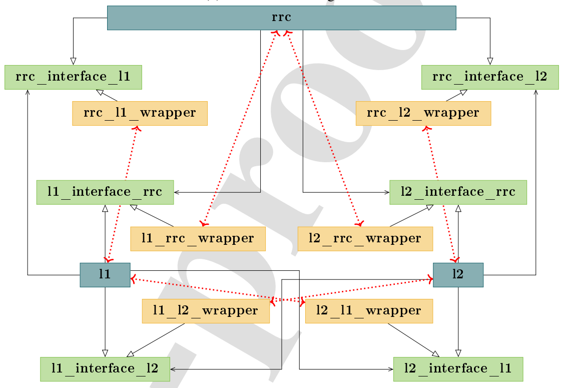
We have followed the *Adapter Design Pattern* to modify the interaction between the protocol stack classes with minimal impact on the existing code. For each pair of entities that interact we have added two wrapper classes, which intercept the function calls both ways.

As can be observed in Figure 3, the new classes, `pdcp_rlc_wrapper` and `rlc_pdcp_wrapper`, also inherit from the existing interfaces. Then, the initialization of the layer classes, `rlc` and `pdcp`, is modified to pass around pointers to instances of the corresponding wrappers, without requiring modifications of the layer classes. In addition, the wrapper classes are also initialized with interface pointers in both ways, for two different behaviors. As can be observed, `pdcp_rlc_wrapper` keeps a pointer to `pdcp_interface_rlc`, which is actually a pointer to `pdcp`. This way, when the `rlc` class needs to call the PDCP implementation, it actually calls `pdcp_rlc_wrapper`, which in turn calls the `pdcp` class. This first behavior does not provide any advantage with respect to the original one when working locally, but it allows us to intercept the interaction between layers. The wrappers also implement communication utilities, so that intercepted calls can be sent to remote entities, as we will describe in detail in Section 3.2.

The second behavior enabled by the wrappers is to receive remote calls. For that, they are also initialized with pointers to the classes that implement the peer interface. For example, in Figure 3, the `rlc_pdcp_wrapper` holds pointers to both



(a) Overall class diagram



(b) Detailed interaction between layers

Figure 4: Implementation class diagram. Inheritance is represented with closed arrows, association with open arrows and solid lines, and general relationships with open arrows and dotted lines.

`pdcp_interface_rlc` and `rlc_interface_pdcp`, actually to the `pdcp` and `rlc` instances, respectively. This is also done for the `pdcp_rlc_wrapper`. Hence, when a wrapper receives a remote call, it is forwarded to the target class. Altogether, a remote call from `rlc` to `pdcp` would comprise the following steps:

1. First, `rlc` calls `pdcp_rlc_wrapper`, using the `pdcp_interface_rlc`.
2. Then, `pdcp_rlc_wrapper` communicates the call to its peer `rlc_pdcp_wrapper`.
3. Finally, `rlc_pdcp_wrapper` calls `pdcp`, using the `pdcp_interface_rlc` interface.

As can be seen, the new behaviors do not require modifications in the internal layer classes, but only during their initialization. Following this approach, the setup of a *CU/DU* is defined by setting the hosted layers and configuring the corresponding wrappers to operate locally or remotely. As a consequence, when some specific layers are not instantiated, depending on the configuration, wrapper classes would have null pointers to interfaces. For instance, the *CU* configuration of the PDCP/RLC functional split (O_2 in Figure 2) would require instantiating the RRC and PDCP layers, and configuring the `rrc_pdcp_wrapper` and `pdcp_rrc_wrapper` to operate locally, while the other wrappers (`pdcp_rlc_wrapper`, `rrc_rlc_wrapper`, `rrc_mac_wrapper`, and `rrc_phy_wrapper`) would be configured to operate remotely. At the same time, the *DU* setup will have a mirror configuration: it will instantiate the RLC, MAC and PHY layers, using the appropriate wrappers to communicate them: `rlc_mac_wrapper`, `mac_rlc_wrapper`, `mac_phy_wrapper` and `phy_mac_wrapper`, operating locally, and the other two remotely.

Figure 4 depicts the interaction between the different classes and a generic representation of the detailed interaction between layers. For simplicity, we do not include the variable members of the classes. As can be seen in Figure 4a the involved classes map to protocol layers. Besides protocol stack classes, srsRAN implements a class for GTP-U, which allows user-plane communication between eNodeB and the network Evolved Packet Core (EPC). In all cases, there is a relationship between any class and the one that implements the RRC protocol, which can be either bidirectional or unidirectional. The latter only happens in the case of GTP-U and PHY, where there is only interaction from RRC to the corresponding class. In addition, each entity of the protocol stack interacts with the adjacent layers, following the pattern described above. In Figure 4b we detail the class diagram implemented for the interaction between any two adjacent classes, as well as between them and RRC. In all cases, protocol classes hold a reference to the interface of the counterpart (for instance `l2` holds a pointer to `l1_interface_l2` to communicate with `l1`), which is actually the implementation of the wrapper. In spite of this convoluted interaction, the proposed design allows code re-usability due to the common pattern, and it also simplifies the implementation validation[3]

3.2. *Split working flow*

As mentioned before, the implementation follows the *Adapter Design Pattern* by adding intermediate classes in charge of implementing the split functionality. This way, each time a (calling) layer, `X`, calls a function of a (called) layer `Y` by using the defined interface, the wrapper class gets called, checks the configuration, and decides whether the call is local or remote. When layers `X` and `Y` are together, the function call is forwarded to the local instance, using the legacy implementation. Otherwise, the call is sent to the remote wrapper pair. In Figure 5 we depict the sequence of calls in case of a remote configuration.

As can be observed, the wrapper at the calling side (`X-Y wrapper`) intercepts the function call and serializes the information required to reproduce the input arguments at the other end. Then, the serialized data is sent to the peer wrapper. Once at the destination, the information is de-serialized and the target function of `Y` is called with the appropriate parameters. Eventually, in case the function returns a result, the returned information is serialized, and sent back to the calling wrapper, which passes it to the calling class `X`.

As shown in Figure 5, we distinguish different time periods in both wrappers. At the calling side, T_{s1} is the time required for the serialization. In the current implementation, we use manual serialization, copying the memory content of variables to

[3]A full class diagram involving all the protocol layers, and their detailed interactions can be found in the public repository, `https://github.com/tlmat-unican/fd-srsran`.

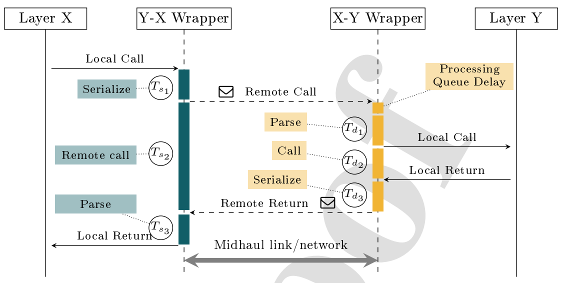


Figure 5: Call sequence of the implementation and identified delay sources.

a sending buffer. However, there are some cases where these variables correspond to complex nested structures or classes. In those cases, we leverage the *C++ ProtoBuffer* library to simplify the serializing and de-serializing tasks. T_{s2} denotes the time elapsed from the sending of serialized data until the returned value is received. Generally, this second time period includes fronthaul/midhaul transmission, waiting time to be processed, and processing time. In this sense, depending on the processing scheme, if the function call rate is high, the destination may require queuing the calls for processing. In addition, as mentioned before, when the function does not return any outcome, the calling side does not wait for it. Furthermore, T_{s3} represents the time required for parsing the return value and sending it back to the calling class.

At the destination, T_{d1} and T_{d3} denote the time for parsing the call and serializing the return value, respectively. Then, T_{d2} is the time taken by the original function implementation to be executed. As can be observed, T_{s2} is not only the sum of all times at the receiver (T_{d1}, T_{d2} and T_d3), but it also contains transmission and potential queuing times. Altogether, the efficiency of the implementation would be given by the ratio between T_{d2} and the total time required for the function call, so that the more relevant the contribution of T_{d2} to the overall time, the less impact caused by the new functionality.

As for the transmission protocol, we have used *ZeroMQ* [17] for different reasons. Firstly, it was already included in the implementation for emulating radio communication, so there are no integration issues. Secondly, *ZeroMQ* takes care of keeping the connection between peers alive, and we do not thus need to take care of how wrappers are instantiated (order). Finally, it provides a set of parallel remote computation patterns that facilitate the processing implementation.

In particular, we have adopted the *Aysnchronous Client/Server Pattern*, as shown in Figure 6, for multi-thread support. In this scheme, a set of clients send tasks to a common remote server in a synchronous manner. In turn, the server dispatches the tasks among a number of local workers and sends the response back to the clients. The communication between clients and server is carried out over TCP connections, while the server uses inter-thread (`inproc`) communications to interact with workers.

We have adapted the defined pattern in order to avoid issues and to improve the system's performance. It is important to

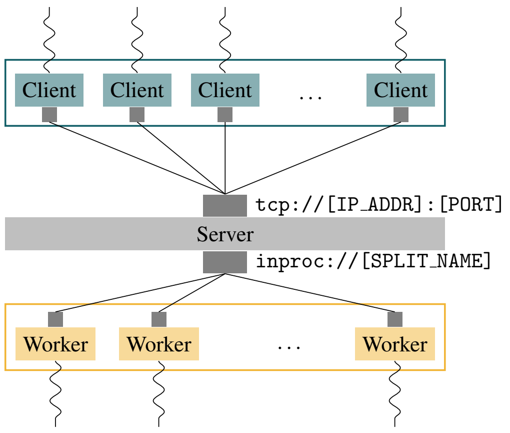


Figure 6: Multi-thread processing design.

note that srsRAN is a multi-thread implementation with complex relationships between threads. In this sense, a function call (for instance `F1`) in a thread could await another function call (`F2`) in another thread to finish. For instance, let us assume that two different threads of a calling layer call `F1` and `F2`, which are implemented on the calling side. When the first thread calls `F1` the interaction with the remote peer is assigned to one client. Then at the called side, the execution of `F1` would wait for `F2` to finish, and the client would also keep awaiting. If the call to `F2` is assigned to the same client both functions would get mutually locked: `F1` waits for `F2` to finish, and `F2` waits for the client to be available. To avoid these issues all threads at the sender side are given independent clients. The current implementation allows both the pre-allocation of clients as well as the creation of new ones on the fly. In this sense, the pre-allocation avoids delays caused by TCP connection in the first interaction. On the other hand, the number of workers on the server side can be also configured. Altogether, the current design allows the independent and parallel processing of function calls of different threads in a transparent way.

As can be seen, ZeroMQ simplifies the management of parallel queries. Nevertheless, modifications to use midhaul standard protocols (such as SCTP) or even new ones not considered in standards for these networks (i.e., QUIC) would not require major modifications. As will be mentioned at the end of the document, the support for other configurable transport protocols will be tackled in our future work.

4. Validation and results

This section presents the validation of the implementation described in the previous section, as well as the analysis of its performance. In order to simplify the measurement campaign, the experiments have been carried out using the *ZeroMQ* radio emulatin provided by srsRAN, which communicates the base station and the *UE*, so that we avoid using radio hardware. It is important to remark that this configuration has no effect whatsoever on the protocol layers involved in the functional split configurations (from PHY to RRC). First, we will analyze the additional delays induced by our implementation, to assess whether they are admissible for the different splits. Then, we will use the developed architecture to obtain planning insights. In particular, we will study the computational load of the different configurations and how it varies for different base station sizes and traffic loads. This provides a good indicator of the resources that would be required for the different setups.

In all cases, the results are obtained using two HP ProLiant ML310e Gen8 v2 servers, equipped with 8 Intel Xeon E3-1241 v3 CPUs (4 cores and 2 threads per core), which work at 3.5 `GHz`. Both servers communicate through a Qnap QSW-M5216-1T switch, with Ethernet ports of 25`Gbps`. By using this high communication capacity, we ensure that the connecting network does not impact the delay characterization. Besides the base station, the setup includes the EPC instance and one *UE*. When the base station is configured to use functional split (any option), one server hosts the *UE* and the *DU*, while both the *CU* and the EPC are deployed at the other. When a functional split is not considered, the *UE* runs in one server, and the whole base station and the EPC are executed at the other one. Finally, the number of clients and workers (see Figure 6) is at least as high as the number of threads using them, for all cases. In this sense, the required number of clients/workers for each split has been analyzed offline, since the number of threads in each communication varies depending on the entities that are separated. Hence, before configuring the actual number of client/workers that were used to obtain the final results, we carried out an initial measurement.

In general, the measurement process works as follows:

1. Deploy the *EPC*, *UE*, *CU*, and *DU* nodes/programs at their corresponding machines.
2. Set the corresponding configurations of all entities by choosing the functional split, cell size in Physical Resource Blocks (PRB), and traffic rate at the *UE*.
3. Start the *EPC*, *CU*, and *DU* nodes/programs, and wait until they successfully establish the corresponding connections.
4. Launch the *UE* node/program and wait until it is successfully connected.
5. Start the transmission of user data from the *UE* node. It generates TCP traffic, using *iperf* at a given rate.
6. Measure the performance using the appropriate tool while the data is being transmitted.

The traffic rates are set according to the cell size. To that end, we have first characterized the maximum achievable rate without splitting for different cell sizes. In particular, for 15 and 25 PRB cell sizes, the maximum rate is set to 10 and 50 `Mbps`, respectively. Otherwise, we set the maximum rate at 100 `Mbps`.

4.1. Split options delays

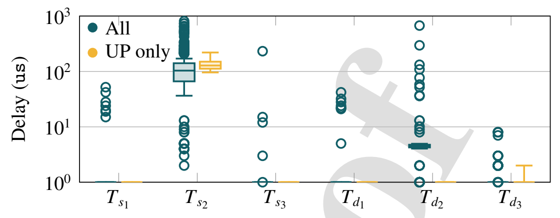
As we mentioned in the previous section, the wrappers perform a set of operations to enable remote communication between protocol layers, which might induce additional delays. In order to analyze the delay caused by our implementation, Figure 7 shows the statistical distributions of the time intervals that were illustrated in Figure 5. The results are obtained during communication that last 60 seconds. The experiment has been repeated for the different configurable splits (O_1, O_2, O_4, and O_6) with a cell of 100 PRBs, sending traffic at the maximum possible rate, and we have recorded the time periods for each call as traversing every wrapper. Figure 7 shows the distribution of such delays using boxplots with logarithmic scale. In all cases, we show the delay obtained from all the calls and the one obtained from calls involving only user-plane traffic (PDU and SDU exchange).

As can be observed in Figure 7a the delay induced with the O_1 split configuration is below 1000 µs in almost all cases. In addition, the results evince that all intervals (cf. Figure 5) are almost negligible, but T_{s_2}. This demonstrates that the serialization/parsing operations do not cause an additional delay. On the other hand, since T_{s_2} includes both the communication and internal processing delays, as well as the accumulated delay at the receiver (T_{d_1}, T_{d_2} and T_{d_3}), the obtained values provide interesting information. According to the results, all delays at the receiver are almost negligible, and values observed in T_{s_2} are so mostly caused by communication and internal processing. Taking into account the high communication bandwidth of the connectivity setup, and the large number of clients/workers that ensure no processing queuing, this delay is therefore due to the communication/processing design itself, and it is the minimum we could expect with our setup.

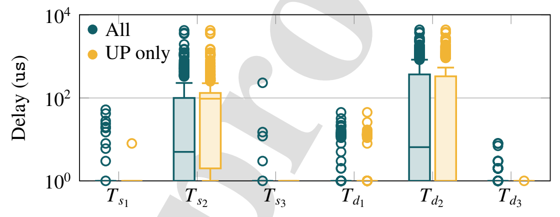
Figure 7b shows a different delay behavior with the O_2 split. Once again T_{s_2} shows higher delays, reaching 4 `ms`. However, we can observe that such delay is in this case due to T_{d_2}, which corresponds to the normal call and cannot be thus avoided. Besides, as we observed in the previous configuration, the delay periods corresponding to serialization/parsing are again almost negligible.

Figure 7c shows that O_4 split presents a similar trend to that observed when discussing the performance of O_2. Again, the only relevant delay is T_{d_2}. Finally, the MAC/PHY split in Figure 7d yields results similar to those of RRC/PDCP. The times included in T_{s_2} are very small compared to the values seen for O_2 and O_4 options. Besides, subtracting the value of T_{d_2} from that observed for T_{s_2}, we could infer the delay induced by our proposed solution, which is, as shown in Figure 7a, mostly due to communication and internal processing. Finally, we observe in all cases that delay distributions are rather similar to the ones seen for user-plane traffic. This is a reasonable behavior, since it is the dominating traffic. In addition, it evinces that there are not delay peaks in the control plane.

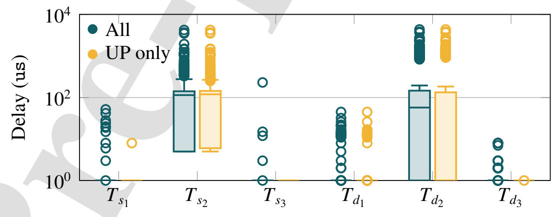
We conclude the evaluation of the delay induced by our implementation with Table 1, where we compare the performance of the proposed architecture with the requirements established by two standardization bodies. To that end, we use the Small Cell Forum (SCF) Release 7.0 [16, Section 2.8] and 3GPP Technical Report 38001 [14, Annex A]. The former pays special attention to small cell requirements, and the latter assumes a macro-base station with a channel bandwidth of 100 `MHz` using the highest modulation (256 QAM). In the table we indicate, for each option and specification, whether the obtained



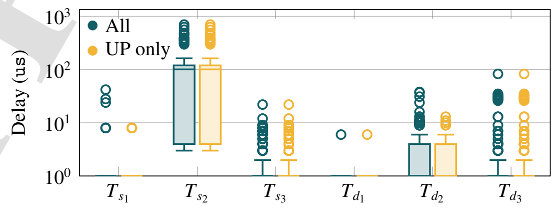
(a) Delays distribution in O_1 - RRC/PDCP split. For each delay, the distribution of overall and user-plane calls is differentiated.



(b) Delays distribution in O_2 - PDCP/RLC split. For each delay, the distribution of overall and user-plane calls is differentiated.



(c) Delays distribution in O_4 - RLC/MAC split. For each delay, the distribution of overall and user-plane calls is differentiated.



(d) Delays distribution in O_6 - MAC/PHY split. For each delay, the distribution of overall and user-plane calls is differentiated.

Figure 7: Distributions of delays in each execution period. Overall results are compared with those obtained considering one user-plane traffic.

performance is compliant with the requirements, using as delay the highest value observed in Figure 7. As can be seen, the proposed implementation in this paper complies, in both cases, with O_1 and O_2 requirements. In addition, it also satisfies the requirements of the Small Cell Forum for O_4 and for O_6, assuming near-ideal conditions, where O_6 split option requires 2 `ms`. On the other hand, the requirements defined by the 3GPP for O_4 and O_6 would not be satisfied. In any case, the performance depends on the particular equipment used, and it is likely that more powerful devices would yield better performances.

Table 1: Feasibility of the implementation in terms of delay compared with requirements by standardization bodies.

	O_1	O_2	O_4	O_6
SCF [16]	30ms ✓	30ms ✓	6ms ✓	(ideal) 250µs ✗ (near-ideal) 2ms ✓
3GPP [14]	10ms ✓	10ms ✓	100µs ✗	250µs ✗

4.2. Computational load

This section describes the metrics that we use to assess the computational effort of the proposed implementation, discussing the results in terms of computational load. As was mentioned earlier, our setup consists of two servers, one hosting the *UE* and *DU*, while the *CU* and EPC run at the other one. We focus on the computational requirements of the *CU*, since it is the entity that could be deployed in shared environments, where the computation resources are shared between competing processes. On the other hand, the *DU* is likely to be deployed in isolation, since it needs to be closer to the *RU*. The results shown hereinafter are obtained by disabling the CPU frequency scaling of the machine, to avoid warm-up periods that can tamper the results.

To measure the computational load of the services running in a machine (in this case, the *CU* implementation), we need to consider the overall number of instructions executed, which depends on the compiler, the Instruction Set Architecture (ISA), the program itself when the service is running (i.e., the executed low-level instructions), and the time spent carrying out the task. In particular, we will use the Giga Instructions Per Second (GIPS) indicator, which is defined as follows:

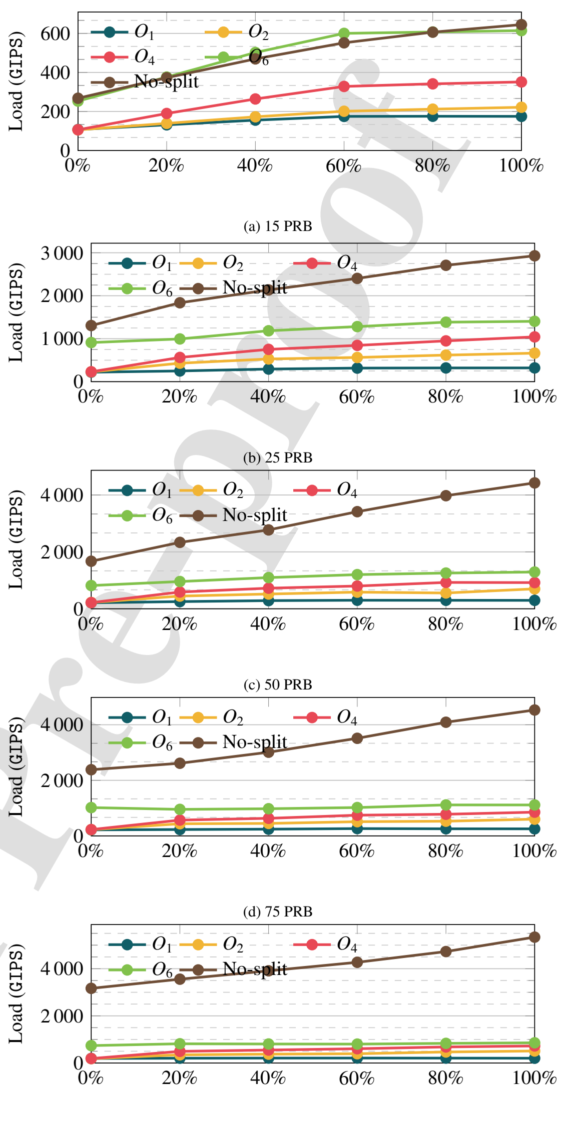
$$GIPS = \frac{I_{count}}{T_{ex}} \cdot 10^{-9} \qquad (1)$$

where I_{count} is the total number of executed low-level instructions, and T_{ex} is the amount of time that the CPU runs the program (in seconds). This information was captured using the `perf` tool[4].

The process used to obtain the GIPS consisted of generating the executable file (sequence of low-level instructions) of each of the nodes/programs, configuring and connecting them. Afterwards, we generate TCP traffic with the `iperf` tool, and we keep track (using `perf`) of the execution time and the number of executed instructions (at the *CU*). This process was repeated for different split options, cell sizes, and empirical cell loads.

Figure 8 shows the computational load results (in GIPS) for 60 second communications for different configurations, varying the size of the base station, the relative load of the base station (from 0 to 100%), and the selected functional split. The figure also shows the results without functional split, which thus sets the computational load baseline performance (upper bound). As was expected, the computational load increases as

[4]`perf` allows access to the performance counters, CPU hardware registers that count hardware related events such as number of executed instructions, cache misses suffered, or branches wrongly predicted. `https://perf.wiki.kernel.org/`



(a) 15 PRB

(b) 25 PRB

(c) 50 PRB

(d) 75 PRB

(e) 100 PRB

Figure 8: Results of computational load measured in GIPS varying the cell size (number of PRBs), empirical cell load (from 0 to 100%), and functional split for the *CU* node.

we centralize more functions in all cases. For the RRC/PDCP split (O_1 configuration), the obtained GIPS are nearly constant when the load varies. These results are consistent because this option sets only the RRC layer at the *CU*, and it is only affected by control traffic. In addition, we can observe that when we increase the base station size, the computational load gets higher, being more remarkable when all protocol stack layers are implemented at the *CU* (highest centralization level).

The results also illustrate the strong impact, in terms of computational load, of the PHY layer. For instance, Figure 8a shows that, for a small cell, there is almost no difference between O_6 configuration and the full centralization alternative (no-split). As we increase the cell size, we observe that the load of the split configurations increases at a much lower pace. This is a reasonable behavior, since, below the PHY layer, the SDR implementation works with I/Q symbols, whose number heavily increases with the cell size.

If we look at the impact of the traffic load, we observe slightly different behaviors between the different split configurations. When there is no user traffic, the number of GIPS for less centralized alternatives (O_1, O_2 and O_4) is close to zero. On the other hand, the number of GIPS of the O_6 configuration has a relatively high value, even without traffic.

An aspect that is related to the assessment of the computational load is how it can be reproduced in (extrapolated to) different computing environments [18], for example, when using more powerful machines. To tackle that, we use a synthetic metric, proposed in [19], which is based on the CPU utilization. This metric defines the computational load of Giga Operations Per Second (GOPS), as follows:

$$R = N_{cores} \cdot C \cdot N_{flop} \cdot \mu \qquad (2)$$

where N_{cores} and N_{flop} hold for the number of cores and flops per cycle, respectively, C is the nominal frequency of the processor, and μ the processor utilization. For instance, in our case we have 8 CPUs, at 3.5 `GHz`, each of them able to execute 32 flops per cycle. Hence, at full utilization ($\mu = 1$) the computational load would be: $R = 8 \times 3.5 \times 32 = 896$ `GOPS`.

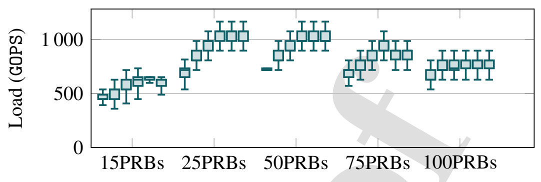
The GOPS metric can facilitate comparisons with respect to other computing environments because it has less dependency on the particular software, since the only term that measures the behavior of the running process is given by processor utilization (μ). On the contrary, GIPS depends on the C++ compiler employed, as the building of the executable file produces a sequence of low-level instructions that the processor will execute according to the fetch-decode-execute cycle.

Note that, although the two metrics (GIPS and GOPS) aim to establish the computational load of the running service (program), there is not a clear and straight relationship between them. We thus expect similar behavior in the corresponding results, but we would not be able to establish an equivalence of an operation to a number of instructions.

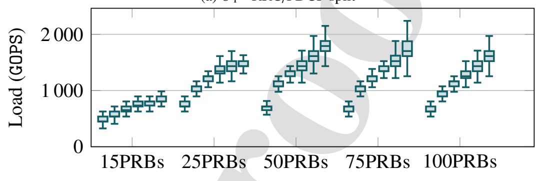
In this case (GOPS), the measurements are carried out by monitoring the temporal evolution of the CPU usage with the *top* tool[5]. In particular, we take CPU usage samples every 100 `ms`, during 60-second experiments.

Figure 9 shows the GOPS distribution for the different split configurations, varying both base station sizes and their relative load, from 0 to 100%. We also include the values obtained when no functional split is used. Overall, we can observe a general increasing trend as we centralize more functionalities, ranging from 500 to 1000 GOPS in the O_1 split and from 100 to 3000 GOPS without split.

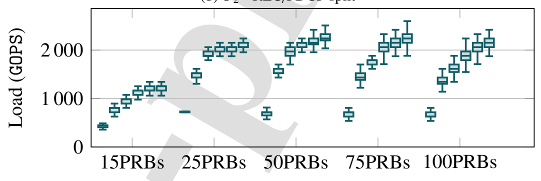
[5]top-tool: `https://man7.org/linux/man-pages/man1/top.1.html`



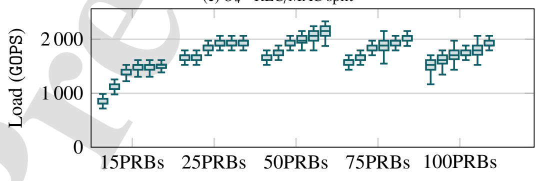
(a) O_1 - RRC/PDCP split



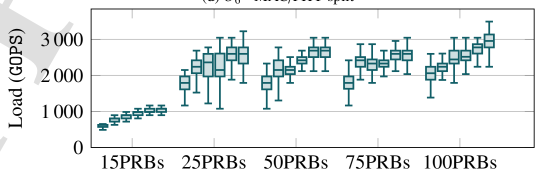
(b) O_2 - RLC/PDCP split



(c) O_4 - RLC/MAC split



(d) O_6 - MAC/PHY split



(e) No-split

Figure 9: CPU usage at the *CU* for different split configuration. For each splits, CPU usage distribution is depicted for different cell sizes (number of PRBs) and different empirical cell load: $\{0, 20, 40, 60, 80, 100\}\%$.

If we focus on the RRC/PDCP split (O_1), Figure 9a evinces that the cell size, defined by the number of PRBs, has almost no impact from 25 PRBs onwards. On the other hand, the impact of the cell load shows a clear relationship with the cell size. As can be observed, with a cell up to 50 PRBs, the cell load saturates around 60% of its relative load. However, its impact is more relevant (saturation at 40%) for cell sizes of 75 and 100 PRBs.

For the rest of the configurations, we can observe that the cell size has little impact beyond 50 PRBs. On the other hand, the dependency of the cell load is different for each split configuration. In this sense, the results obtained with RLC/PDCP and RLC/MAC splits, see Figures 9b and 9c, respectively, show a strong correlation with the cell load for all PRB values. As can be observed, when no user traffic is sent (first value of the cell load) the CPU usage remains rather low, regardless of the cell size, while it rapidly increases as we start sending traffic. Unlike the MAC/PHY split and the no-split configuration shown in Figures 9d and 9e, CPU usage is not negligible, even without any user traffic.

5. Conclusions

In this paper we describe and evaluate an implementation of cellular base stations that allows the configuration of up to four functional splits. The implementation, which is based on the srsRAN open-source project, is meant for experimentation purposes. In particular, the work described in this paper permits configuring functional split options O_1 (RRC/PDCP), O_2 (PDCP/RLC), O_4 (RLC/MAC) and O_6 (MAC/PHY), while option O_8 (PHY/RF) is intrinsically included by design in the SDR adopted solution.

First, we have described the software design, by depicting class diagrams of the corresponding code, both before and after the changes that we made. We have adopted the *Adapter Design Pattern*, where intermediate pieces of software are included in the legacy code to intercept the communication between protocol layers. In this sense, the impact over the existing code is almost non-existing, and the proposed architecture does not require modifications on the classes that implement the stack. Hence, it would simplify the adoption of this software design to new releases of the SDR solution or new technologies, such as 5G.

We have afterwards analyzed the performance of the proposed architecture in two different and complementary ways. First, we have studied the delay induced by our implementation over different configurations. The results evince that it respects the delay constraints established by the Small Cell Forum for all the possible splits, while the stricter requirements defined by the 3GPP for macro-cells are only fulfilled for the O_1 and O_2 configurations.

We also study the computational load, using GIPS and GOPS metrics, for different split configurations, cell sizes, and traffic loads. GOPS is a synthetic metric that can be used to compare the effort in other computing environments, as it is less dependent on the software used. In contrast, GIPS allows us to measure the computational load by counting low-level instructions that are executed in an elapsed time. As expected, the results show a growing effort as we centralize more functions, while the impact of the other parameters strongly depends on the split setup.

The implementation described in this paper has been made available to the research community in a public repository. We believe that the solution described in the paper can be leveraged by the research community in different ways. One potential application would be to characterize the requirements of computation and communication capabilities for deploying disaggregated RAN, and how different scenarios would impact the overall RAN performance. In addition, we conceive the implementation as a sandbox that can be exploited to develop cooperative solutions over the different RAN elements, according to the selected splits.

In addition, we plan to extend the functionalities of the implementation described in this paper in two directions, which would broaden the applicability of the solution. First, we will analyze the feasibility of using the same design pattern to the recently released 5G implementation of srsRAN, which already features the O_2 split, according to the O-RAN architecture. The solution presented in this paper would thus allow experimentation, both in the RAN side and network segments involved, beyond the O-RAN. As part of these extensions, we will include other communication protocols, instead of ZeroMQ, to communicate the *DU* and *CU*, such as SCTP or even QUIC for experimentation purposes. In addition, we will extend our implementation to support novel architectures where several *DUs* can be attached to the same *CU*.

Acknowledgements

This project was funded the Spanish Government (Ministerio de Economía y Competitividad, Fondo Europeo de Desarrollo Regional, MINECO-FEDER) by means of the project SITED: Semantically-enabled Interoperable Trustworthy Enriched Data-spaces (PID2021-125725OB-I00). The authors also acknowledge the participation of Mr. Alberto Castilla, who participated in the implementation tasks described in the paper, during his MSc thesis.

References

[1] R. K. Saha, Y. Tsukamoto, S. Nanba, K. Nishimura, and K. Yamazaki, "Novel m-cord based multi-functional split enabled virtualized cloud ran testbed with ideal fronthaul," in *2018 IEEE Globecom Workshops (GC Wkshps)*, 2018, pp. 1–7.

[2] Y. Alfadhli, Y.-W. Chen, S. Liu, S. Shen, S. Yao, D. Guidotti, S. Mitani, and G.-K. Chang, "Latency performance analysis of low layers function split for urllc applications in 5g networks," *Computer Networks*, vol. 162, p. 106865, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128619301343

[3] A. Younis, B. Qiu, and D. Pompili, "Latency and quality-aware task offloading in multi-node next generation rans," *Computer Communications*, vol. 184, pp. 107–117, 2022.

[4] S. Matoussi, I. Fajjari, S. Costanzo, N. Aitsaadi, and R. Langar, "5g ran: Functional split orchestration optimization," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 7, pp. 1448–1463, 2020.

[5] A. M. Alba, J. H. G. Velásquez, and W. Kellerer, "An adaptive functional split in 5g networks," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2019, pp. 410–416.

[6] V. Q. Rodriguez, F. Guillemin, A. Ferrieux, and L. Thomas, "Cloud-ran functional split for an efficient fronthaul network," in *2020 International Wireless Communications and Mobile Computing (IWCMC)*. IEEE, 2020, pp. 245–250.

[7] N. Bartzoudis, O. Font-Bach, M. Miozzo, C. Donato, P. Harbanau, M. Requena, D. López, I. Ucar, A. A. Saloña, P. Serrano, J. Mangues, and M. Payaró, "Energy footprint reduction in 5g reconfigurable hotspots via function partitioning and bandwidth adaptation," in *2017 Fifth International Workshop on Cloud Technologies and Energy Efficiency in Mobile Communication Networks (CLEEN)*, 2017.

[8] Y. Alfadhli, M. Xu, S. Liu, F. Lu, P.-C. Peng, and G.-K. Chang, "Real-time demonstration of adaptive functional split in 5g flexible mobile fronthaul networks," in *2018 Optical Fiber Communications Conference and Exposition (OFC)*, 2018.

[9] P. Monti, Y. Li, J. Mårtensson, M. Fiorani, B. Skubic, Z. Ghebretensaé, and L. Wosinska, "A flexible 5g ran architecture with dynamic baseband split distribution and configurable optical transport," in *2017 19th International Conference on Transparent Optical Networks (ICTON)*, 2017.

[10] C.-Y. Chang, N. Nikaein, R. Knopp, T. Spyropoulos, and S. S. Kumar, "Flexcran: A flexible functional split framework over ethernet fronthaul in cloud-ran," in *2017 IEEE International Conference on Communications (ICC)*, 2017.

[11] Y. Li, J. Martensson, B. Skubic, Y. Zhao, J. Zhang, L. Wosinska, and P. Monti, "Flexible ran: Combining dynamic baseband split selection and reconfigurable optical transport to optimize ran performance," *IEEE Network*, vol. 34, no. 4, pp. 180–187, 2020.

[12] S. K. Singh, R. Singh, and B. Kumbhani, "The evolution of radio access network towards open-ran: Challenges and opportunities," in *2020 IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*, 2020, pp. 1–6.

[13] T. D. Tran, K.-K. Nguyen, and M. Cheriet, "Joint route selection and content caching in o-ran architecture," in *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, 2022, pp. 2250–2255.

[14] 3rd Generation Partnership Project (3GPP), "Study on new radio access technology: Radio access architecture and interfaces," 2017.

[15] O-RAn Working Group 1 (Use Cases and Overall Architecture), "O-RAN Architecture Description," 2023.

[16] Small Cell Forum), "S-RU and S-DU Test Support," 2021. [Online]. Available: https://scf.io/en/documents/228_S-RU_and_S-DU_Test_Support.php

[17] P. Hintjens. (2011) 0mq - the guide. [Online]. Available: http://zguide.zeromq.org/page:all

[18] J. Vitek and T. Kalibera, "R-3 - repeatability, reproducibility and rigor," *ACM SIGPLAN NOTICES*, vol. 47, no. 4, S, pp. 30–36, APR 2012.

[19] J. K. Chaudhary, A. Kumar, J. Bartelt, and G. Fettweis, "C-ran employing xran functional split: Complexity analysis for 5g nr remote radio unit," in *2019 European Conference on Networks and Communications (EuCNC)*, 2019, pp. 580–585.

Author Biography

Biographies

Cristian C. Erazo-Agredo was born in Colombia, in 1990. He received the degree in systems engineering from the Universidad del Cauca, Popayán, in 2017, and the M.Sc. degree from the Center for Research and Advanced Studies, National Polytechnic Institute of Mexico, where he is currently pursuing the Ph.D. degree in engineering and computational technologies. His research interests include computational intelligence, multiobjective optimization, and network virtualization.

Luis Diez received the M.Sc. and Ph.D. degrees from the University of Cantabria in 2013 and 2018, respectively, where he is currently an Assistant Professor with the Communications Engineering Department. He has been involved in different international and industrial research projects. As for teaching, he has supervised 15 B.Sc. and M.Sc. thesis, and he teaches in courses related to cellular networks, network dimensioning, and service management. His research focuses on future network architectures, resource management in wireless heterogeneous networks, and IoT solutions and services. He has published more than 40 scientific and technical papers in those areas. He has served as TPC member and reviewer in a number of international conferences and journals.

Ramón Agüero Calvo (Senior Member, IEEE) received the M.Sc. degree (First Class Hons.) in telecommunications engineering and the Ph.D. degree (Hons.) from the University of Cantabria in 2001 and 2008, respectively, where he is currently a Full Professor with the Communications Engineering Department. Since 2016, he has been the Head of the IT Area (Deputy CIO) with the University of Cantabria. He has supervised five Ph.D.s and more than 70 B.Sc. and M.Sc. thesis. He is the main instructor in courses dealing with networks, and traffic modeling, both at B.Sc. and M.Sc. levels. His research focuses on future network architectures, especially regarding the (wireless) access part of the network and its management. He is also interested on multihop (mesh) networks, and network coding. He has published more than 200 scientific papers in such areas. He is a regular TPC member and reviewer on various related conferences and journals. He serves in the Editorial Board of IEEE COMMUNICATION LETTERS (Senior Editor since 2019), IEEE OPEN ACCESS JOURNAL OF THE COMMUNICATIONS SOCIETY, Wireless Networks (Springer), and Mobile Information Systems (Hindawi).

Mario Garza-Fabre received the M.Sc. and Ph.D. degrees in computer science from the Center for Research and Advanced Studies (Cinvestav), Mexico, in 2009 and 2014, respectively. From 2015 to 2018, he worked as a Research Associate/Fellow with the University of Manchester and Liverpool John Moores University, U.K. In 2018, he joined Cinvestav, where he is currently an Associate Professor of Optimization and Computational Intelligence. His main research interests involve the analysis and design of (meta-)heuristic optimization techniques, as well as their application to problems from areas such as bioinformatics, data mining/machine learning, and transportation.

Javier Rubio-Loyola is a research scientist at the Center for Research and Advanced Studies (Cinvestav), Mexico. He holds an Engineering degree in Communications and Electronics and an MSc degree in Digital Systems, both from Instituto Politécnico Nacional of México and a PhD in Telecommunications from Universitat Politècnica de Catalunya in Barcelona.

He has participated in several Spanish and IST-European research projects mainly in the network management area. His research interests focus on network management, network functions virtualization and software defined networks (NFV/SDN) and applied artificial intelligence.

Author Photo

Author's Photo

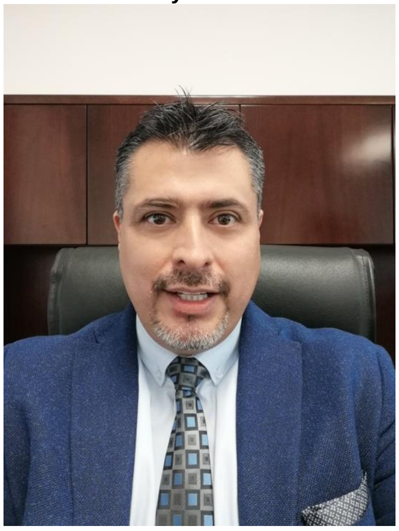
Cristian C. Erazo-Agredo

Luis Diez

Ramón Agüero Calvo

Mario Garza-Fabre

Javier Rubio-Loyola



Declaration of Interest Statement

The authors declare no conflicts of interests.