Statistically sound experiments with OpenAirInterface Cloud-RAN prototypes

CLEEN 2016

Niccolò Iardella[1], Giovanni Stea[1], Antonio Virdis[1], Dario Sabella[2], Antonio Frangioni[1]

[1] University of Pisa, Italy; [2] Telecom Italia Lab, Turin, Italy

niccolo.iardella@for.unipi.it, giovanni.stea@unipi.it, a.virdis@iet.unipi.it, dario.sabella@telecomitalia.it, frangio@di.unipi.it

Abstract— Research on 4G/5G cellular networks is progressively shifting to paradigms that involve virtualization and cloud computing. Within this context, *prototyping* assumes a growing importance as a performance evaluation method, besides large-scale simulations, as it allows one to evaluate the computational requirements of the system. Both approaches share the need for a *structured* and statistically sound experiment management, with the goal of reducing errors in both planning and measurement collection. In this paper, we describe how we solve the problem with OpenAirInterface (OAI), an open-source system for prototyping 4/5G cellular networks. We show how to integrate a sound, validated software, namely ns2-measure, with OAI, so as to enable harvesting samples of arbitrary metrics in a structured way, and we describe scripts that allow structured experiment management, such as launching a parametric simulation campaign and harvesting its results in a plot-ready format. We complete the paper by demonstrating some advantages brought about by our modifications.

Keywords: LTE-A, Cloud-RAN, OpenAirInterface, performance evaluation, experimentation, ns2-measure

1 Introduction

Future 5G cellular networks will employ *virtualization* and *cloudification* of the Radio Access Network (RAN) [12], whereby the baseband processing is done on *virtual baseband units* (BBU) running on commodity hardware, leaving only antennas on site. On the other hand, software products, both commercial and open-source, are already available that emulate a software BBU compliant with the 3GPP standards. One such product OpenAirInterface (OAI), which runs an LTE protocol stack entirely implemented in software [2]. OAI also allows one to carry out experiments using hardware equipment and commercial terminals. The above two fact motivate a shift in the research paradigm, which is progressively based on *prototypes* of cellular networks. In fact, OAI has been and is being widely used in EU-funded and academic projects in the field of cellular networks. The Flex5GWare EU project [6], where the authors of this paper are involved, aims at building cost-effective hardware/software platforms for 5G so as to increase the hardware versatility and reconfigurability, increase capacity and decrease the overall energy consumption. Within it, one of the proof of concepts will consist in evaluating resource allocation algorithms in a Cloud-RAN environment, which will be realized running a customized version of the OpenAirInterface software on virtual machines.

This implies the need to get credible *performance metrics* out of the OAI software, for both the cell and the user, and at several levels: what is the cell MAC-level throughput, how user application-level throughput varies with the number of users or interfering eNBs, how much energy is consumed, etc. It goes without saying that the above activity must be done with a long-term perspective, so as to keep the software maintainable, and ensuring that rigorous, unbiased and statistically sound results are obtained. In this respect, it has already been observed in [7], [8] that an *unstructured* approach to experiment management is often a major source of bugs, and ultimately affects the credibility of the results.

Unfortunately, OAI offers little in the way of a structured experiment management, leaving the task almost entirely to the user. First of all, emulation scenarios are defined in non-parametric XML files. This requires a user to manually change the XML file so as to modify the parameters (e.g,. in order to vary the number of users), possibly in several parts simultaneously, which is error-prone. For instance, even generating a new replica of the same emulation scenario with a different random seed becomes non trivial. As far as measure gathering is concerned, OAI offers two basic ways: one is system logging printouts, which can be redirected to file and parsed (using standard tools such as `grep`). The other is a built-in dashboard, which shows the instantaneous situation at the physical level in terms of channel response and signal power. These tools, which were probably meant for different purposes – namely, logging/debugging for the first one, and debugging and providing a quick visual feedback regarding physical-layer parameters for the second, are not suited for a systematic performance evaluation. For instance, the throughput is computed having the *simulation duration* at the denominator, regardless of when the generator is actually started. This implies that – if generators are started at different times in the simulation – the throughput results are incorrect. Moreover, there is no way to define a *warm-up* phase, where samples are not collected. Finally, the overhead of writing on file the entire system log (of which just a minor portion may be of interest) is non negligible.

In this paper we describe how to automate experiment management with OAI so as to make it faster, structured and less error prone. First of all, we show how to integrate an existing software, namely *ns2-measure* [7], into OAI. *ns2-measure* was originally developed for the ns2 simulator, and offers to researchers a framework for data collection and creation of statistically sound results. We describe the steps to compiling the two software together (something made slightly tricky by the fact that OAI is written in ANSI C, whereas ns2-measure is in C++), and the few, localized modifications required to OAI. This enables a user to gather a wide range of measures of interest in a seamless way, adding a negligible overhead to the OAI running time and memory consumption. Moreover, we describe intuitive, yet general scripts that can be used to generate parametric emulation scenarios and aggregate performance metrics across a set of parametric scenarios to facilitate producing output graphs and tables. As for parametric scenario generation, our script describe the set of parameters that should vary across the scenarios (therein including the initial seed for the random generators when independent replicas are required) at a high level, and the script generates the XML scenario files to run the OAI emulation and manages their execution. As for aggregation of performance metrics, we show scripts that allow to compute means and related confidence intervals, taking measures from *ns2-measure* outputs or OAI built-in logging facilities.

The rest of the paper is organized as follows: Section 2 reports background information on OAI. Section 3 describes the ns2-measure software. In Section 4 we describe our tools and explain how to integrate ns2-measure with OAI. We report some example evaluation results in Section 5, and we conclude the paper in Section 6.

2 OpenAirInterface

OpenAirInterface (OAI) is an open-source platform for wireless communication systems, developed at Eurecom's Mobile Communications Department. It allows one to prototype and experiment with LTE and LTE-Advanced (Rel-10) systems, so as to perform evaluation, validation and pre-deployment tests of protocol and algorithmic solutions. OAI allows one to experiment with link-level simulation, system emulation and real-time radio frequency experimentation. As such, it is widely used to setup Cloud-RAN and Virtual-RAN prototypes. It includes a 3GPP-compliant LTE protocol stack, namely the entire access stratum for both eNB and UE and a subset of the 3GPP LTE Evolved Packet Core protocols [2].

OAI can be used in two modes: the first one is a real-time mode, where it provides an open implementation of a 4G system interoperable with commercial terminals, so as to allow experimentation. This requires using a software-defined radio frontend (e.g. the Ettus USRP210 external boards [3]) for airtime transmission.

The second mode is an emulation mode, where software modules emulating eNBs and UEs communicate through an emulated physical channel. In the emulation mode, scenarios are completely repeatable since channel emulation is based on pseudo-random number generation. In emulation mode, OAI can emulate a complete LTE network [1], using the oaisim package. Several eNBs and UEs can be virtualized on the same machine or in different machines communicating over an Ethernet-based LAN. The PHY and the radio channels are either fully emulated (which is time-consuming) or approximated in a PHY abstraction mode, which is considerably faster. In both cases, emulation mode runs the entire protocol stack, using the same MAC code as the real-time mode. This way, the oaisim package can be used to alpha-test and validate new implementations or sample scenarios, dispensing with all the problems that airtime transmission on a SDR frontend may bring about. Since the same code is used in the emulation and the real-time mode, a developer can then switch seamlessly to the real-time environment.

OAI includes the *OAI Traffic Generator* (OTG), which can be mounted on top of the LTE stack and used to run an emulation with different loads [4]. The generator includes predefined traffic profiles, such as device-to-device, gaming, video streaming and full buffer, and can be customized using OAI scenario descriptors (OSDs).

OAI's structure reflects the one of the LTE protocol stack: every layer of the stack is composed of one or more modules, implemented by one or more C libraries. Every layer or module uses calls to interface functions of other modules to retrieve status information and to encapsule/decapsule data. For example, the MAC scheduling module gets called by the main MAC module at every subframe and is implemented in *eNB_scheduler.c* and *pre_processor.c* files [9]. Every application in the OAI suite (such as oaisim and the real time eNB) instantiates and initializes the stack layers and the other modules it needs: oaisim, for example, makes use of other modules for the emulation capabilities, the most notable being the OTG, the OAI Channel Generator (OCG) which emulates the radio channel and the OAI Mobility Generator (OMG) that emulates the movements of the nodes. After the *init phase*, the application enters in a *loop phase* where modules and layers execute their functions on a per-subframe basis; lastly, before exiting, the main process deallocates the layers and possibly executes termination operations (e.g., output of performance stats).

OAI software uses three methods for the output of performance metrics:

- a graphical dashboard that can be optionally shown while the system emulation or the eNB implementation runs, which shows received/sent signal power, channel impulse and frequency responses, constellation diagram and PUSCH/PDSCH throughput (see **Fig. 1**).
- A series of prints in the standard output logging of the system emulation, which appear when the traffic generator is enabled, and show traffic-related metrics (sent and received bytes, application level throughput, one-way delay and so on).
- One or more files with PHY level stats on HARQ processes and DLSCH/PDSCH throughput.

All these methods are useful to get a rough idea of how the system behaves, but none of them, taken alone, is sufficient to profile it completely and effectively: the graphical dashboard is shown in real-time, it leaves no logs and it is destroyed once OAI terminates. The traffic generator stats have the disadvantage of being written on standard output together with the entire OAI log, so they must be collected using `grep` or other text search tools, which can be more and more impractical as the number of simulation runs and input parameters increases. The same can be said of the output files for the PHY stats. Another limitation of the traffic generator stats is that throughput is calculated on the *entire* emulated time, taking no account of the initial warmup time in which the system is running but the generator is not.

In general, OAI is missing a structured, flexible and extendable system for the collecting and managing performance measures. Different metrics get collected and shown in different ways and the only way for the user to keep track of experiment results is to tailor custom scripts to launch OAI and extract the desired data from the existing outputs. The time interval of samples collection cannot be selected and this makes the analysis of dynamic scenarios difficult when not impossible and leads to warped measurements when warmup time is a critical factor.

Moreover, built-in metrics might not be sufficient for research purposes. For example, the performance evaluation of MAC scheduling algorithms would need to keep track of resource block allocation, which is not among the built-in metrics. In this case, the lack of an efficient and robust metrics collection framework makes custom metrics hard to implement and collect, and even when they are implemented they are bound to the same limits of the built-in ones.

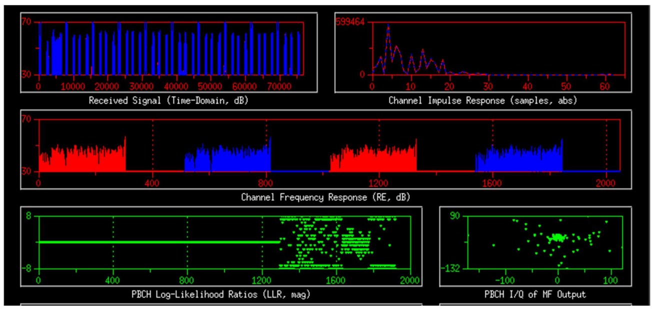
Received Signal (Time-Domain, dB)
Channel Impulse Response (samples, abs)
Channel Frequency Response (RE, dB)
PBCH Log-Likelihood Ratios (LLR, mag)
PBCH I/Q of MF Output

Fig. 1. Detail of the graphical dashboard showing in real-time the physical-level stats of a node in OAI system emulation.

3 Ns2-measure

The *ns2-measure* package [7] is a C++-based framework for collection of statistics. It was originally developed for the ns2 network simulator [10], offering an interface to TCL, its main configuration language. Its C++ API however, can be used for integration into any C/C++ code. The main goal of *ns2-measure* is to provide researchers a structured and ready-to-use tool for collection of statistically sound measurements. More in detail it can be used for both collecting samples of user-defined metrics during the simulation, and to estimate the average values or the probability density function (PDF) of the above samples. Metrics can be of three *types*, depending on how their samples are collected, as listed below:

- RATE, which are time-related and time-averaged, e.g. the throughput;
- CONTINUOUS, describing a continuous-time stochastic process (either discrete- or continuous-state), i.e. one whose trajectories are continuous in time. An example is the number of packets in the queue during the simulation, which is a discrete quantity (hence discrete-state) that varies at any time (hence continuous-time);
- DISCRETE, describing a discrete-time stochastic process, i.e. one whose trajectories are impulses. An example is the end-to-end delay of a flow, a continuous quantity measured at successive packet departure instants (hence discrete-time).

The framework also offers the user support for independent replicas, which are used to obtain statistically sound results (e.g., with associated confidence intervals).

Each metric (of any type) can be defined for more than one entity at a time within the system. This allows a user to obtain both system-wide and per-entity statistics. For example, when simulating an LTE network one might be interested in both a cell-level and a per-UE throughput, and both can be defined and sampled simultaneously. Data collection can be enabled and disabled *dynamically* at runtime by flipping the *collect* variable. This allows the user some (very much needed) freedom: for instance, she can define a warm-up time wherein statistics are not collected, or she can measure the throughput of intermittent applications in a meaningful way, by turning on throughput sample collection only when a burst of activity occurs.

The core element of *ns2-measure* is the *Stat* C++ class. It is responsible for creating data structures for each metric at the beginning, collecting samples while the emulation runs and producing output at the end. It also keeps a reference to the elapsed emulated time, to tag time-related metrics, such as the RATE ones. These operations are made available via three main C++ functions. The *Stat::command* instantiates the data structures for the user-defined metrics, activates and deactivates the collection and manages the output file. The available metrics are configured via file and are included into the system during compilation. The above data structures are implemented in the *Sample* class, which stores the measured samples for each entity, keeping track of their total, maximum and minimum values.

The *Stat::put* function is used to insert data collection probes within the code. This function takes as a parameter the name of the considered metric, the ID of the entity for which the sample is collected and the measured value, which will be stored in the appropriate instance of the *Sample* class, possibly updating the max and min values.

The *Stat::print* function is used to finalize an experiment. More in detail, it computes and stores to a file the estimated mean value of each metric. Files will also contain the run-id of the experiment, which can be used when multiple replications of the same scenario are run, e.g. to aggregate metrics across the various replications.

The output of experiments performed on complex and possibly large system can grow quite big in some cases. For this reason results are stored into *binary* files, thus reducing the occupancy with respect to text files. If needed, the results can be converted to human-readable text format using external tools that come together with the *ns2-measure* framework.

4 Contributions and integration

In this section we first explain how to integrate *ns2-measure* with OAI, and then show scripts that automate experiment management, so as to facilitate running entire simulation campaigns.

4.1 Integrating ns2-measure

As outlined before, the core of *ns2-measure* is the *Stat* class, which collects the raw samples from user-defined probes. It is a static C++ class which uses the method *Stat::command* to implement the TCL interface and interpret the commands specified in **Table 1**. The other main method is *Stat::put*, which implements the collecting probe and accepts as input parameters the name and type of the metric and the value of the sample. The metrics' names are defined in two headers, *metrics.h* (which contains macros with names to use when calling the *put* method) and *metric_names.h* (which contains human-readable names to be used when saving the output).

Since the *Stat* class has been developed with ns2 in mind, a certain effort of adaptation must be spent to port it on other simulators. In particular we need to work: a) on code interoperability so that the *Stat* methods can be used in the new environment; b) on the method that the *Stat* collecting probe must use to read the simulated time when acquiring samples, and c) since build automation tools (such as `CMake`) are likely to be used, we need to make them aware of the new code. The following passages describe the specific interventions on OAI. However, they are general enough to apply to other C++-based simulation softwares.

Code wrapping - OAI is mainly implemented in C and makes no use of the TCL language, so the static class must be modified so as to allow its methods to be called from the C code. To achieve this, we implemented a wrapping library which contains one C function per TCL command: for example *stat_cmd_add()* calls *Stat::command*, thus emulating the "`add`" TCL command and so on. Similarly, the *stat_put()* function wraps the *Stat::put* method (see **Fig. 2**).

Simulated-time reading – On collecting a sample, the *Stat* class calls an ns2 method to read the current value of the simulated time. In OAI these calls must be replaced by a read to the *time_ms* variable, which is updated at every new subframe.

Build tools – Since OAI uses `CMake` [11] as an automation tool for building, we added as a libraries the *Stat* class and the wrapping library, and we added these libraries to the OAI System Emulation target. To speed up testing, we added the ability to activate or deactivate the *ns2-measure* functionality at run time via configuration file, without having to recompile the target.

Table 1. Main commands of ns2-measure TCL interface.

`$ stat file <filename>`	Specify the output file
`$ stat on`	Enable sample collection
`$ stat off`	Disable sample collection
`$ stat print`	Print stats on output file

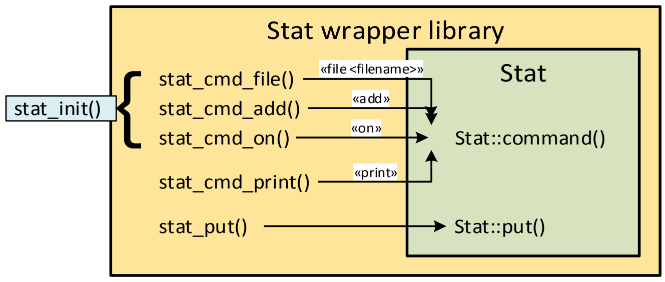

Fig. 2. The wrapping library contains one wrapper function per TCL command and a wrapper function for the probe. An initialization procedure is defined for the sake of convenience.

In order to use the new code, the OAI code must be modified in at least two points, namely the initialization phase and the termination phase, plus all the points where we want to put a probe in the loop phase. In the initialization phase we need to call the `file` command, specifying the name of the output file (again, the file name can be specified through a configuration file), and the `add` command, once per metric. In this phase one may want to activate measure collection using the `on` command (alternatively, this can be deferred to the time when the traffic is actually started). For the sake of convenience, all these operations are gathered in a *stat_init()* function. In the termination phase we need to call the `print` command so that the output metrics are calculated and the output is printed of the specified file. In the loop phase, probes are added where required. The procedure to add a new metric is thus quite straightforward, and consists of the following steps:

- define its name in the *ns2-measure* files *metrics.h* and *metrics_names.h*,
- add a call to *stat_add()* inside the *stat_init()* function (*stat_init.c*),
- add the probe using *stat_put()* wherever samples are to be collected, including the *stat.h* header. For example, if we need the number of resource blocks allocated by the scheduler, we need to insert a *stat_put()* call inside the *pre_processor.c* file.

Note that the target code must be recompiled *only* when new probes and/or new metrics are inserted, while *ns2-measure* can be (de)activated via configuration file.

4.2 Experiment management automation

The method used by OAI to define scenarios is XML files, the so-called *OAI Scenario Descriptors* (OSDs). These allow a very fine-grained customization of the emulation scenario, editing parameters such as the transmission power of the eNB antennas, the mobility model for the nodes and the profile of the traffic flows. However, OSDs do not support *variable parameters*, so when running an emulation campaign a different OSD must be prepared for each combination of parameter values.

To fill this gap we implemented an automatization script package: the main script takes as input a configuration file where parameter values or ranges of values are specified; then, for every combination of parameter values it calls another script which generates a specific XML descriptor, and launches the OAI system emulation using that descriptor; lastly, another script parses the results from different runs and gathers them in a CSV file. For example, if we want to try the same scenario with 1, 2 or 3 UEs, we specify the parameter *numUEs* as {1, 2, 3} and the script will generate three different XML descriptors, launch OAI three times, and merge the three sets of results in a CSV file. This process is shown in **Fig. 3**.

5 Experimental results

The purpose of this section is twofold. On one hand, we show that the integration of ns2-measure framework has a negligible impact on OAI performance. On the other hand, we exemplify the benefits that our framework brings to the user by showing that different (and unbiased) throughput results are obtained by allowing sample collection to start with the traffic generation (instead of at time zero), and by showing that comparing different metrics allows a user to get an immediate insight on the behavior of the system.

To assess the impact of the new code on performance, we evaluate the execution time and memory occupancy as a function of the traffic rate, both with and without *ns2-measure* samples collection activated.

We run OAI System Emulation (oaisim) on a machine with an AMD FX 8350 4 GHz CPU, 8 GB RAM, running Xubuntu 14.04.2, emulating an eNB sending downlink traffic to a UE, using increasing traffic rates. The main emulation parameters are summarized in **Table 2**.

A note on traffic generation: OAI allows one to specify the size of generated traffic packets at the *application* level. OAI appends 55 bytes of TCP/IP headers and OTG metadata [9] to each packet. If we specify a packet size of 100 B and an inter-packet time of 1 ms, we obtain a data rate of 100×8=800 kbits/s at the application level, or (100+55)×8=1240 kbits/s at the IP level. OAI statistics refer to *IP-level* traffic.

We added 12 custom metrics, which get collected on a per-subframe basis. This adds 12 function calls to the *init* phase and 12×1000=12000 function calls per second to the *loop* phase. Each configuration/scenario is run three times with three different seeds, for a total of nine runs per configuration. To evaluate the execution time and the memory occupancy (more specifically the *maximum resident set size*, i.e., the maximum amount of memory the process allocates during its execution) we use the `/usr/bin/time` command [5].

Fig. 4 shows the results for the execution time: the introduction of *ns2-measure* samples collection introduces minimal to null overhead. Also memory occupancy is unchanged, being about 800000 kB for every run.

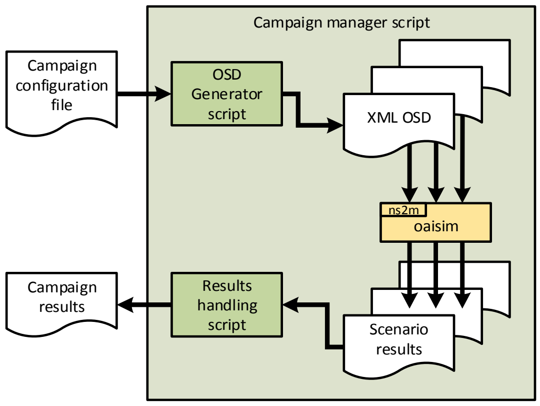

Fig. 3. Automated campaign management using handling scripts.

Table 2. Main parameters for the performance evaluation campaign.

Parameter	**Value**
Emulated time	20000 TTIs (20 s)
# eNBs	1

# UEs	1
Mobility and position	Static - eNB and UE are 200 m apart
Traffic type	CBR: 800, 1600, 2400, 3200 kbits/s *at the application level*
# ns2-measure metrics	12

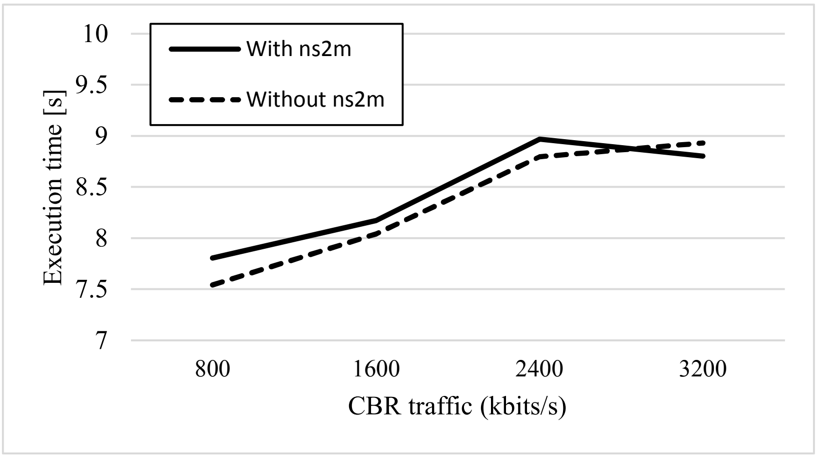

Fig. 4. Execution time of OAI system emulation, determined with /usr/bin/time, emulating 20 seconds of DL CBR traffic from an eNB to a UE.

We now show that our solution eliminates biases in throughput measurement. Since the traffic generator needs the underlying protocol stack and an active radio bearer to work, it needs to wait for the initialization of the stack and the establishment of the RRC connection. The OAI in-code documentation fixes the minimum starting time at 310 ms [14], and we chose a starting time of 500 ms in our experiments.

The native stat collection uses the entire emulation time to calculate traffic throughput and other rates, without considering the traffic starting time. Conversely, in our experiments, *ns2-measure* started collecting samples when the traffic started. In **Fig. 5** we show the IP-level throughput of the UE, as measured by the native traffic generator and by *ns2-measure*. As expected, the values reported by *ns2-measure* are slightly higher, since the measurement interval is 500ms shorter (as it should be).

This very experiment can also be used to show another benefit of using a flexible metric collection: a sub-linear behavior can be observed in the throughput curve, which suggests that the network approaches saturation as the offered load increases. This claim can be easily verified by collecting the number of resource blocks (RBs) allocated to the UE by the eNB scheduler (25 being the maximum number of RBs for the specific configuration). The number of RBs is shown on the right vertical axis, and clearly shows that the knee in the throughput is due to resource depletion. The same metric can also be used to infer the energy consumed by the eNB, according to well-established models of energy consumption ([13]), e.g. to evaluate the energy efficiency of the scheduling algorithm in use.

Moreover, while the statistics offered by OTG are calculated above the LTE protocol stack (i.e., at the IP level), with ns2-measure we can probe all the layers, e.g. to assess the overhead introduced by each of them. **Fig. 6** shows the throughput measured at different layers. As we expect, the closer we get to the physical layer the higher the throughput is, as more headers are added to the application payload.

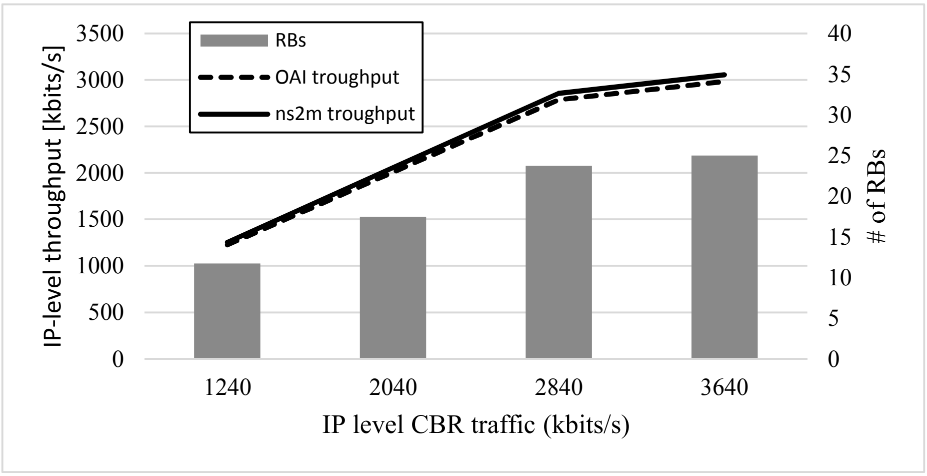

Fig. 5. IP-level throughput between an eNB and a UE as measured by OAI traffic generator and *ns2-measure* (left vertical axis); number of RBs allocated to the UE (right vertical axis).

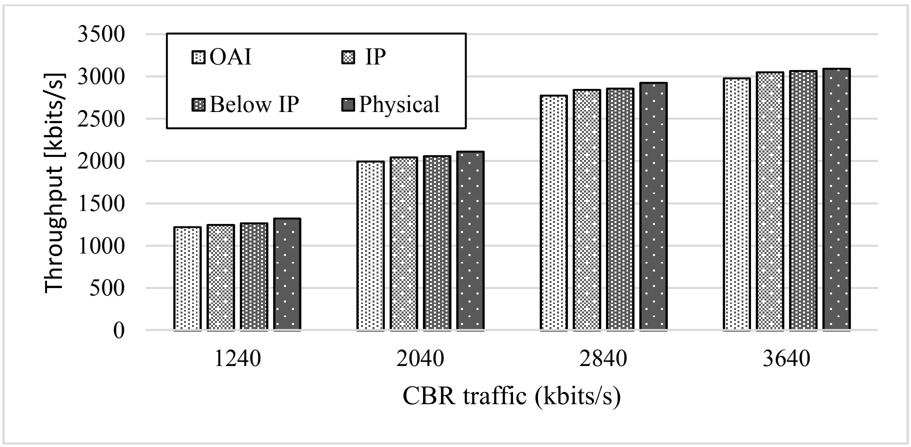

Fig. 6. Data throughput between an eNB and a UE, using different profiles of DL CBR traffic, as measured by ns2-measure at different layers of the LTE protocol stack.

6 Conclusions

This paper presented a set of tools to automate experiment management with a C-RAN prototype realized through OpenAirInterface. These tools allow a user to create a whole simulation campaign, i.e., to launch (possibly several replicas of) scenario where parameters vary, and to harvest the results obtained in the above campaign in a plot-friendly way. Having these tools spares a user time-consuming and error-prone tasks, which can be automated, thus enhancing the credibility of her simulations and increasing her productivity.

As a companion and complementary contribution, we integrated a structured and validated measuring framework, namely *ns2-measure*, into OAI. This allows one to define metrics in an easy way, and enable/disable measure gathering dynamically. On one hand, this speeds up debugging, since it allows a user to analyze the reasons of unexpected behaviors in the system by cross-checking different related metrics. On the other hand, this presents the user with a simple unified approach to harvesting measures, thus facilitating experimenting in the large (e.g., in teamwork).

7 Acknowledgements

The subject matter of this paper includes description of results of a joint research project carried out by Telecom Italia and the University of Pisa. Telecom Italia reserves all proprietary rights in any process, procedure, algorithm, article of manufacture, or other result of said project herein described.

This work was partially supported by the European Commission in the framework of the H2020-ICT-2014-2 project Flex5Gware (Grant agreement no.671563).

8 References

[1] R. Wang *et al.* "OpenAirInterface - An effective emulation platform for LTE and LTE-Advanced", Proc. of ICUFN 2014, Shanghai, China: IEEE, 2014, pp. 127–132.

[2] OpenAirInterface website, Url: http://www.openairinterface.org. (accessed January 2016)

[3] Ettus Research USRP B200/B210 Bus Series, Url: http://www.ettus.com/content/files/b200-b210_spec_sheet.pdf (accessed January 2016).

[4] A. Hafsaoui, N. Nikaein, W. Lusheng, "OpenAirInterface Traffic Generator (OTG): A Realistic Traffic Generation Tool for Emerging Application Scenarios", Proc. of MASCOTS 2012, pp.492-494, 7-9 Aug. 2012

[5] M. Kerrisk., time(1) - Linux manual page. url: http://man7.org/linux/man-pages/man1/time.1.html. (accessed January 2016)

[6] Flex5Gware website: http://www.flex5gware.eu (accessed Jan 2016)

[7] C. Cicconetti, E. Mingozzi, and G. Stea 2006. An integrated framework for enabling effective data collection and statistical analysis with ns-2. In Pro. WNS2'06, Pisa, Italy, October 10, 2006.

[8] L.F. Perrone, C. Cicconetti, G. Stea and B. Ward, "On the Automation of Computer Network Simulators", in Proc. of SIMUTOOLS 2009, Rome, 3-5 March 2009.

[9] A. Virdis, N. Iardella, G. Stea, D. Sabella, "Performance analysis of OpenAirInterface system emulation", in Proc. of PMECT 2015, Rome, Italy, August 26, 2015.

[10] The Network Simulator - ns-2, Url: http://www.isi.edu/nsnam/ns/ (Accessed Jan 2016).

[11] CMake, Url: https://cmake.org/ (Accessed Jan 2016).

[12] "C-RAN: The Road Toward Green RAN", China Mobile Research Institute (2011), Beijing, China, Oct. 2011, Tech Rep.

[13] D. Migliorini, G. Stea, M. Caretti, D. Sabella, " Power-aware allocation of MBSFN subframes using Discontinuous Cell Transmission in LTE systems", CLEEN 2013, Las Vegas, USA, Sep. 2 2013

[14] Gitlab OpenAirInterface repository, Url: https://gitlab.eurecom.fr/oai/openairinterface5g/blob/master/targets/SIMU/EXAMPLES/OSD/WEBXML/template_0.xml (accessed January 2016).