5G Testbed for MEC Implementation

Ans Fida, Haotian Deng, Mohammad Haseeb

Abstract

This project presents an implementation-based study of the concept of Mobile Edge Computing (MEC). We took a deploy-implement-experiment approach in this project. We first deploy a 4G LTE network testbed using the open-source software OpenAirInterface. Next, we implement a basic MEC controller that is intelligent enough to route UE traffic destined for MEC to MEC and all other traffic to the Internet via EPC. Finally, we compare and contrast the difference in latencies added by the OAI EPC core code. Our results show that accessing the MEC directly is 7-10 milliseconds faster on average as compared to accessing it via the EPC. This has been tested on our testbed and on a local area network. This project has been an effort towards paving the way for our eventual goal of implementing a *carrier independent MEC*.

1. Introduction

There is exciting research going on in both the telecommunication industry and the academia regarding the new generation mobile network: 5G. Many new technologies have been proposed, some of which will help achieve the 5G network we have defined while others will compliment user experience and lower the load on network providers. NFV, SDN, cRAN, mmWave etc. are all designed while keeping the 5G KPIs in mind. Along with these new solutions, the concept of Mobile Edge Computing (MEC) is another promising concept that will cater to the emerging delay-sensitive services, such as real-time virtual reality, safe autonomous driving, remote healthcare monitoring, IoT, etc.

MEC offers cloud-computing capabilities at the edge of the network to application developers and content providers. It allows the availability of the cloud servers inside or adjacent to the base stations. This ensures ultra-low latency and high bandwidth as well as real-time access to radio network information that can be leveraged by different applications to increase performance [1]. By providing low latency requirements to users and only selectively sending data over the core network to the remote cloud, MEC provides an attractive platform for users, application developers, and network operators alike. The European Telecoms Standards Institute (ETS) has the responsibility of coming up with standards for MEC and aiding its timely development.

The purpose of our work has been to verify the feasibility of MEC and an MEC controller, quantify its performance aspects, and find control-plane related information and functions that might be needed to build the next generation MEC i.e. a carrier independent MEC. However, before we could begin working on MEC, we had to have some infrastructure or a testbed on which to build and test our hypothesis. For this, we leveraged OpenAirInterface (OAI), which is an open-source software implementation of the 4G LTE network stack and entities. We explain OAI and our testbed setup in detail in section 2. We specifically looked into how a mobile network operator might implement the routing of user data at the eNodeB and what are the limitations of the different approaches. In addition to this, we quantified the performance gain of MEC over OAI implementation. We present our solution for packets routing at MEC controller and performance assessment in sections 3 and 4, respectively. We wrap our paper by mentioning some of the future directions for this work and finally, providing some concluding remarks on our work.

2. Testbed Setup

The concept of building a mobile network testbed for experimental purposes has gained momentum in recent years. Industry giants like Qualcomm and Deutsche Telekom have their own

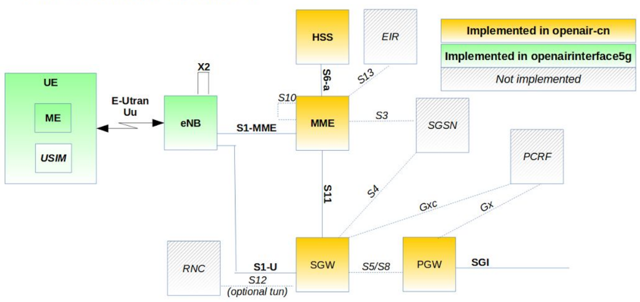


Figure 1: OAI Entities

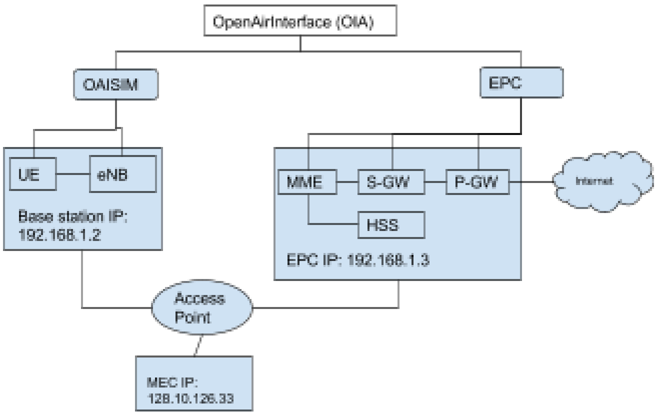


Figure 2: Testbed Architecture

testbed implementations [2]. There are also some publicly available testbeds such as 5G Berlin [3]. They offer many services related to 5G including 5G Playground, Open5GCore and OpenSDN Core. In academia, NYU Wireless Lab announced last year that they will be building "an advanced programmable platform to support the development of millimeter wave (mmWave) wireless communication" [4]. Prior to us, Professor Guan-Hua Tu at Michigan State University (MSU) had setup an OAI testbed and we got some initial help from their work.

OAI is an open-source software-based implementation of the 4G LTE network. Figure 1 shows the different 4G LTE network entities implemented in the OpenAirInterface software [5]. OAI can be used to build and customize an LTE base station (OAI eNB), a user equipment (OAI UE) and a core network (OAI EPC) on a machine running Linux. OAI provides separate modules for each of the above entities but also provides a UE simulator that emulates an actual UE. In this case, the eNB and the UE code are bundled together into the module called OAISIM. Although we originally wanted to use an actual UE along with a USRP, we could not immediately get hands on a programmable SIM card that is

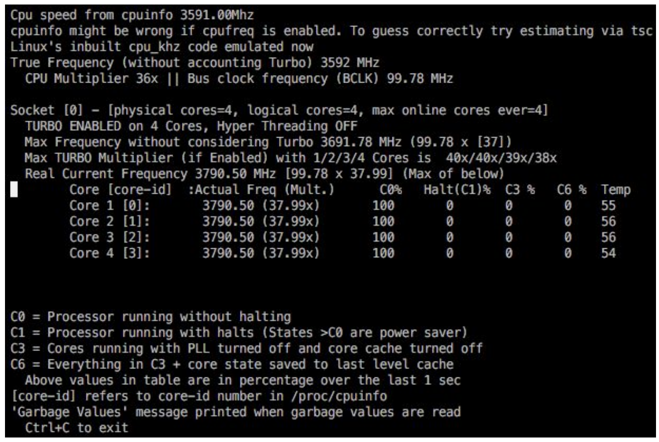


Figure 3: eNB Machine CPU configuration

required for setting up the UE and hence, decided to go with the UE simulator provided by OAI. The are multiple tutorials [6][7][8] from the OpenAirInterface community which help us a lot during our deployment.

We now talk about the underlying hardware of our testbed and then give an in-depth explanation of the software side of the testbed. Note that we have three machines in our testbed setup: (1) eNB and UE machine with IP 192.168.1.2, (2) EPC machine with IP 192.168.1.3 containing the MME, HSS, S-GW and P-GW modules and (3) MEC machine with IP 128.10.126.33. We use the MSSN2 server to serve as our MEC for now. Figure 2 shows a high level overview of our testbed architecture.

2.1. Hardware Configurations

OAI provides numerous options when it comes to running the eNB and the UE. Among others, some of the options the user has are selecting various radio settings for the eNB, whether to use a physical UE with a USRP or using a built in simulator for the UE, etc. Since the eNB has to emulate many LTE network operations from the physical layer up to the application layer, the eNB module of OAI is a compute-intensive piece of software that requires fast hardware. For this reason, we had to turn off a number of CPU related checks on the eNB machine which would allow all cores on the machine to run at full capacity. We disabled all power management features (HyperThreading, CPU frequency control, C-States, P-States). Moreover, we disabled CPU Frequency scaling to enable all the

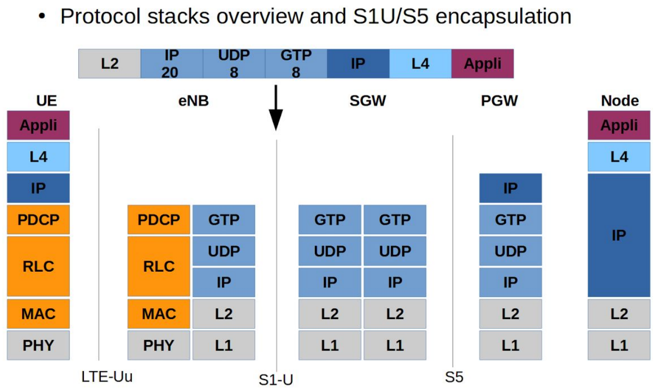


Figure 4: OAI Protocol stack overview

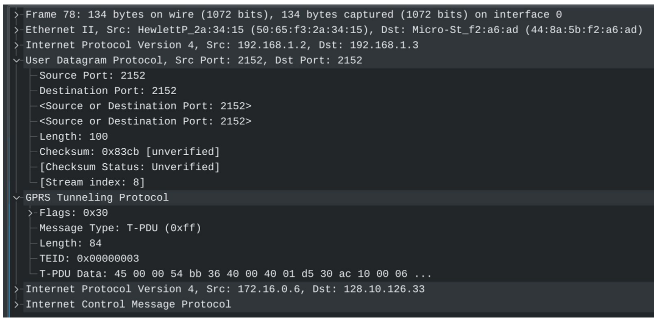


Figure 5: An example of ICMP packets transferred through S1-U link.

cores to function at their highest frequencies. We verified that all cores were operating at full capacity using an Ubuntu package called *i7z*. The output is shown in Figure 3. It shows that all four cores of our eNB machine are operating at their maximum frequencies and they are all running without halting. Additionally, it verifies that Hyperthreading has been turned off.

2.2. Software Configurations

OAI is primarily designed to be run on Ubuntu. Although the tutorial we were following mentioned to use Ubuntu 14.04 for eNB and Ubuntu 16.04 for EPC, we found that having different operating system versions on both the entities caused errors while exchanging initial SCTP set up messages between the two machines. We then switched two using the same Ubuntu versions (16.04) on both the machines. This solved the SCTP mismatch problem. As mentioned before, the eNB code simulates many RAN functions and therefore, requires ultra-low latencies. Consequently, the eNB code also requires the use of a low-latency kernel instead of the general kernel shipped in an Ubuntu distribution. The same holds true for the EPC code, however, that required a special kernel version made for the OAI EPC.

Having setup the hardware and the operating system at each host, the next step was to configure the actual OAI entities, namely the MME, HSS, Serving and Packet Gateway (SPGW) and the OAISIM (eNB and UE). OAI provides predefined configuration files for each of these entities and it is the developer's responsibility to correctly configure the IP addresses, port numbers, host names, and database entries for each entity's configuration file.

This completed our software configuration for the OAI and after this, the following interfaces were initialized between their respective entities: (1) S6-a between HSS and MME, (2) S11 between MME and SGW, (3) S5/S8 between SPGW, (4) S1-MME between eNB and MME and (5) S1-U between eNB and SGW. Moreover, the UE was successfully able to PING any host on the Internet using the virtual network interface provided by the OAISIM configuration. A Wireshark packet capture on the EPC machine verifies that the packets from the UE to any Internet host are indeed being routed through the EPC machine. Appendix A contains the screenshots of the whole flow from launching the HSS to the UE accessing the Internet through a PING command on the virtual network interface (having IP address 176.16.0.2).

3. MEC Controller

In order to support MEC server service along with the current mobile system so that computing power and media content required by latency sensitive applications can be brought from cloud to edge, eNodeB is expected to use an extra link to deliver specific user packets to MEC servers deployed at network edge directly instead of routing packets going through the core network and reaching the remote server finally. MEC controller plays the role of filtering out selected user packets and forwarding them to the the MEC server here. We explain how we implement our simplified MEC controller and the lesson we learned in this section.

As shown in the Figure 4 about the protocol stack overview, in the implementation of eNodeB side in the OpenAirInterface project, all user plane packets coming from user equipment will be decapsulated from physical layer till link layer and stopped before the IP layer. The eNodeB will then treat the user IP packets as an application layer packet and encapsulate it with a GTP (GPRS Tunnelling Protocol) header, a UDP header (to indicate the SGW port on EPC machine) and another IP header (to indicate the EPC machine IP address). The final new IP packets generated by eNodeB implementation are sent out to the core network through the virtual network interface oip1 which is created by OAI for internal transmission. The core network code of OAI will receive and handle this IP packets and extract the inside user IP packets and finally deliver it out to the Internet. Figure 5 shows an example of an ICMP (generated by ping command) packet originally from UE and go through the S1-U link to the S-GW. The inner user IP packets contains the ICMP packet and has the 172.16.0.6 (the virtual private IP address assigned to the UE simulator) as source IP address and 128.10.126.33 (MSSN2 as the MEC server) as the destination IP address. The outer IP packet header encapsulated by the eNodeB uses 192.168.1.2 (local IP address of eNodeB machine) as the source address and 192.168.1.3 (local IP address of EPC machine) as the destination address. The port number 2152 in the UDP header is designated for S1-U link between eNodeB and S-GW. The basic responsibility of MEC controller is filtering out such user IP packets targeted to the MEC server and reroute it to the MEC server directly by modifying the outer IP packet destination address to the MEC server address.

3.1. IPTables

Our first approach is utilizing iptables command on the Linux machine which host eNodeB to redirect those specific user traffic packets. The linux tool iptables command is an administration tool for IPv4/IPv6 packet filtering and NAT (Network Address Translation). The original expectation is using the u32 module in iptables to match the packets with specific pattern and apply routing rules for those packets. However, it turns out that all redirect rules specified in iptables applied to packets which go through oip1 interface will be overwritten by the eNodeB implementation. An interesting finding is that we are capable to drop those packets locally but any redirection fails to take effect. Such observation makes us believe the eNodeB side implementation in OAI overwrite over any iptables rules for packets go through the oip1 interface.

Our second approach is to bring another machine as the physical MEC controller entity to the table. Our target is using this physical machine to separate packets coming from eNodeB and redirecting them to either MEC server or the EPC machine. By modifying the configuration of the OAISIM, we are able to specify the IP address used for S1-MME interface to the MEC controller machine and the registrations procedure between eNodeB and the core network succeed after we use iptables to route all incoming packets from OAISIM machine to EPC server at the MEC controller. However all the following user traffic still goes to the EPC machine directly. The reason is the eNodeB side get the IP address used for S1-U interface to S-GW through the inner communication with MME. Such behaviour cannot be manipulated simply through changing configurations at eNodeB side. Based on this understanding we also tried to modify the IP address for S11 interface between MME and S-GW to the MEC controller machine. Unfortunately such changes break the implementation of the openEPC as the EPC server are no longer reachable to the MEC controller machine.

3.2. Routing Code

As the previous approach of using iptables command did not fulfill our expectation, we implemented our own filtering and routing functionality at the GTP layer at the eNodeB side implementation. The basic idea is straightforward: in the GTP implementation, we locate the range of bytes from the GTP packet payload which representing the destination address of the user IP

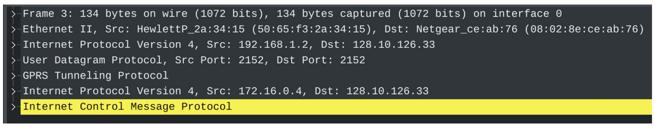


Figure 6: An example of ICMP packet received at MEC server directly from eNodeB.

packets by calculating the proper offset. This offset is fixed as the header length for both GTP headers (8 bytes) and the user IP packets header (20 bytes) implemented in the OAISIM is fixed. So this payload range for the user IP packet destination address is from the 25th byte to the 28th byte. After filtering out user IP packets with destination address to the MEC server, we send it over UDP message by assigning the destination address to MEC server instead of S-GW.

Our patch code are applied to the gtpv1u_send_udp_msg() function in the openairinterface5g/operair3/GTPV1-U model. After applying such modification, we managed to redirect all user packets with MEC server as destination to MEC server directly without going through the EPC first.

Figure 6 shows an example of ICMP packet received at the MEC server which is sent out directly from eNodeB. The outer IP packet has the address of MEC server as destination address instead of the S-GW address like other normal user traffic.

4. Preliminary Assessment and Validation

We try to conduct the preliminary study about the latency improvement for routing traffic to edge server without going through the core network based on the the testbed upon OpenAirInterface implementation. The results we presented here could only show the extra latency caused by the openEPC implementation and the value could be even larger in reality.

Following are the detail of the preliminary study procedure. We keep sending ping requests from the user device to the MEC server for 60 seconds and record both timestamps of ICMP request and response packets using Wireshark to calculate the RTT. We compare the latency

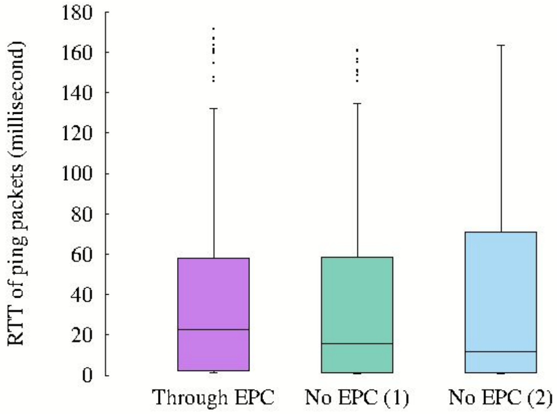


Figure 7: Comparison of latency performance between three cases using RTT of ping request and response messages.

performance between traffic which go to MEC server directly with traffic which go through the core network before reaching the MEC server (case "***Through EPC***") by enable/disable our MEC controller routing function.

There is an issue need to be mentioned here, the IP packets which contains the ICMP message received at MEC server directly from eNodeB also contains extra headers including one GTP header, one UDP header and another IP header unlike those IP packets received from core network after header decapsulation. So the ping message from user equipment without going through core network cannot get proper response, instead a "destination unreachable" response is received from the MEC server (case "***No EPC 1***"). The source IP address of such response messages actually is the address of the MEC server, so it still can be used to calculate the RTT although we cannot quantify the processing difference between such response with a proper ping reply. In addition to this, we also implement a simplified server at the the MEC machine with Python socket programming. Once a IP packet contains GTP, UDP, IP and ping request inside is received, the server will send the same IP packets back to the user equipment immediately as a response (case "***No EPC 2***"). The wireshark running at UE side can also capture such response and calculate the RTT.

We compare the latency of these three cases. Figure 7 shows the performance gain of the two "No EPC" cases compared with the "Through

EPC" case, both "No EPC" cases get smaller median RTT values with 7 and 10 milliseconds respectively while the "No EPC 2" case has 10 millisecond larger for the upper quartile.

This preliminary study validates that routing traffic to the edge server directly without going through the core network could help to improve the latency. Considering in our testbed, both eNodeB and EPC are deployed under the same local area network, the network delay would be ignorable and most of the saving should be caused by the EPC processing time depends on the OpenAirInterface core network implementation.

5. Future Work

Our main task and purpose in this project was to setup the infrastructure (testbed) on which we implement and test our initial MEC controller. However, some problems like UE mobility, security, privacy and billing are still open and need to be solved. Our next step, therefore, is to design and implement a separate SDN controller that would assist and orchestrate the flow of traffic between the UE, eNB and EPC based on control plane information from the EPC and eNB.

Some of the basic requirements we define for this SDN controller are that it must be extremely lightweight, secure and must only implement the crucial control functions and nothing extra. Deciding on what control plane functions to implement is another open question. The reason for having such strict requirements on the controller is so that we can still ensure and provide the ultra-low latency and high availability benefits promised by MEC while still be able to enhance and provide a new carrier independent MEC.

6. Conclusion

In this project, we first setup an open-source 4G LTE network testbed called OpenAirInterface (OAI). We then implemented and proved the feasibility of the concept of an MEC controller. Furthermore, we measured and assessed the latency of accessing the MEC directly and of accessing the MEC after going through the EPC. Our results show that the former approach gains 7-10 milliseconds improvement of latency on average based on our OpenAirInterface testbed with both eNodeB and EPC deployed in local area network.

Although much work is still to be done, this work would be the stepping stone for our final goal with this project i.e. implementing and studying the possibility of a *carrier independent MEC*.

REFERENCES

[1] http://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing

[2] https://www.qualcomm.com/news/releases/2017/02/21/qualcomm-showcases-5g-leadership-its-first-3gpp-based-5g-new-radio

[3] http://www.5g-berlin.org/482f765f1380172e

[4] https://campustechnology.com/articles/2016/01/20/nyu-wireless-to-build-test-bed-for-5g.aspx?admgarea=news

[5] http://www.openairinterface.org/docs/workshop/3_OAI_Workshop_20170427/training/CN_training_user_plane.pdf

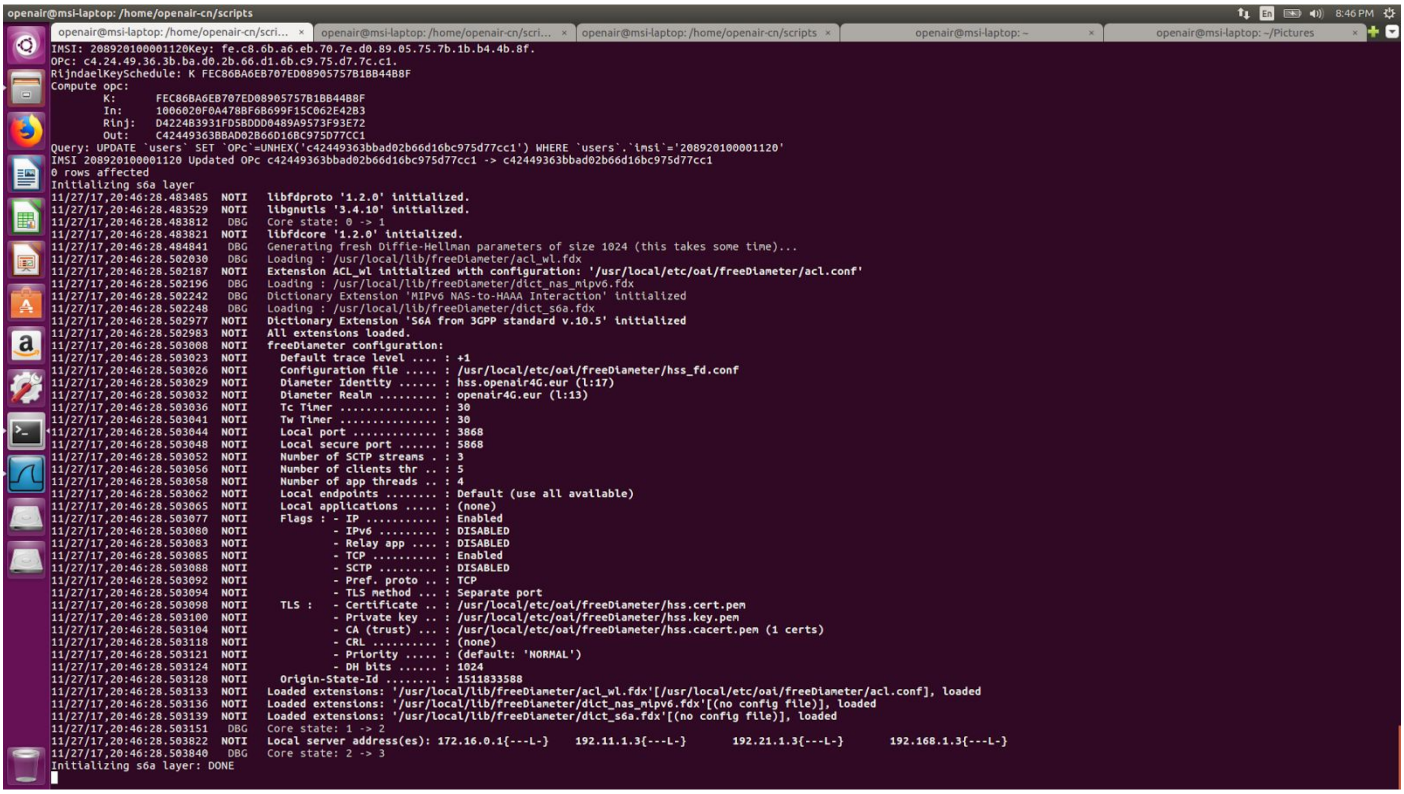
[6] How to connect OAI eNB (USRP B210) with COTS UE. https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/HowToConnectCOTSUEwithOAIeNBNew

[7] How to connect with OAI EPC (on different machines). https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/T/howtoconnectoaisimwithoaiepc

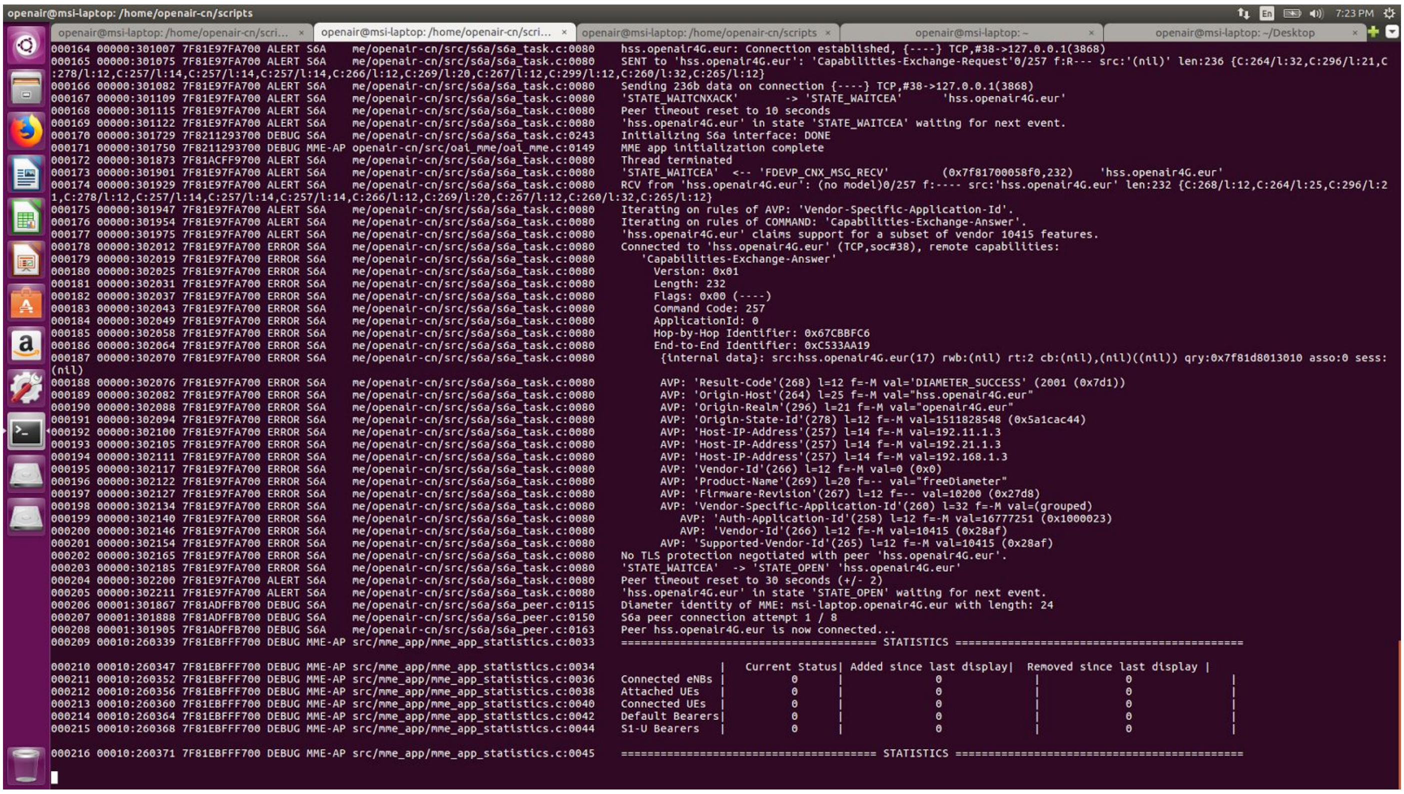
[8] OpenAirInterface System Emulation (aka oaisim). https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/OpenAirLTEEmulation

APPENDIX A

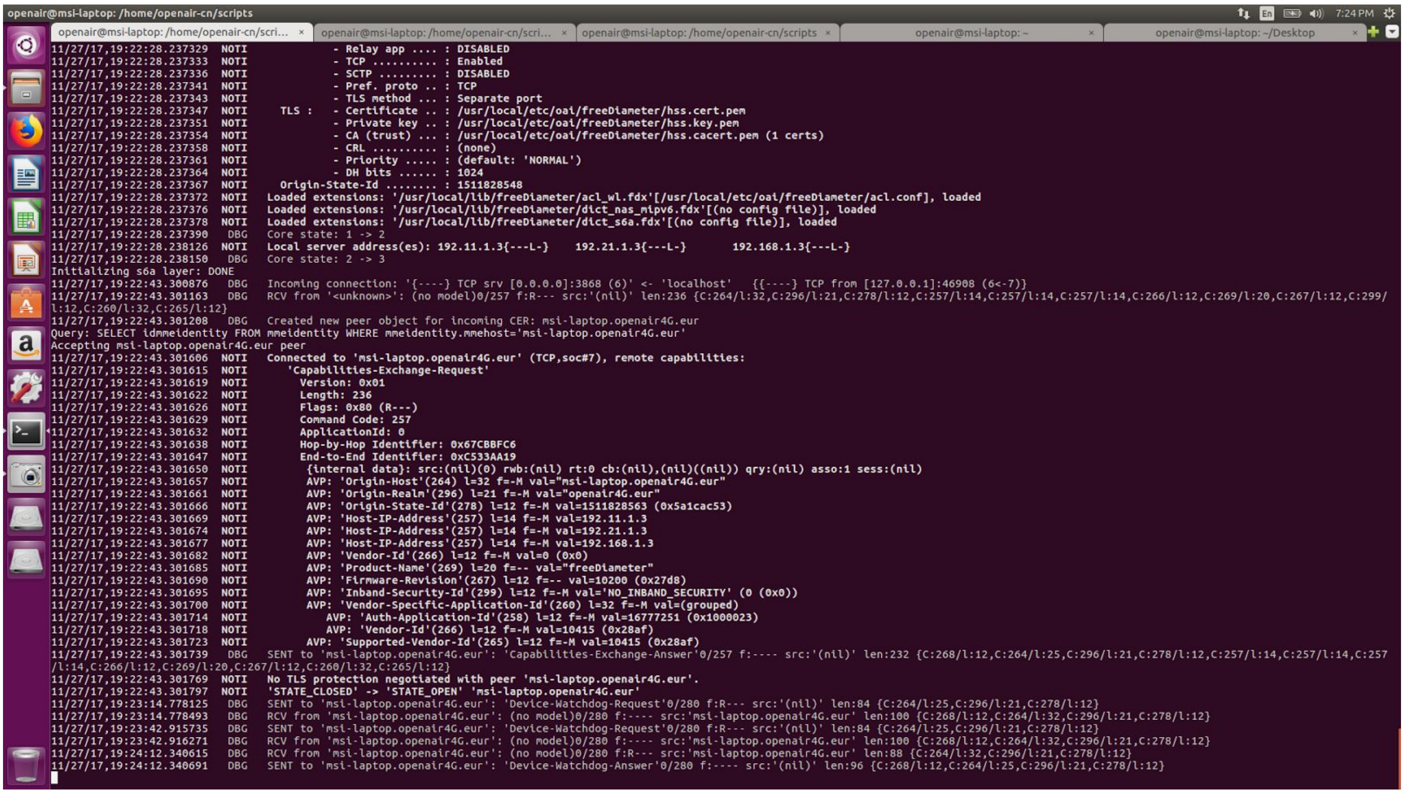
- HSS Launch



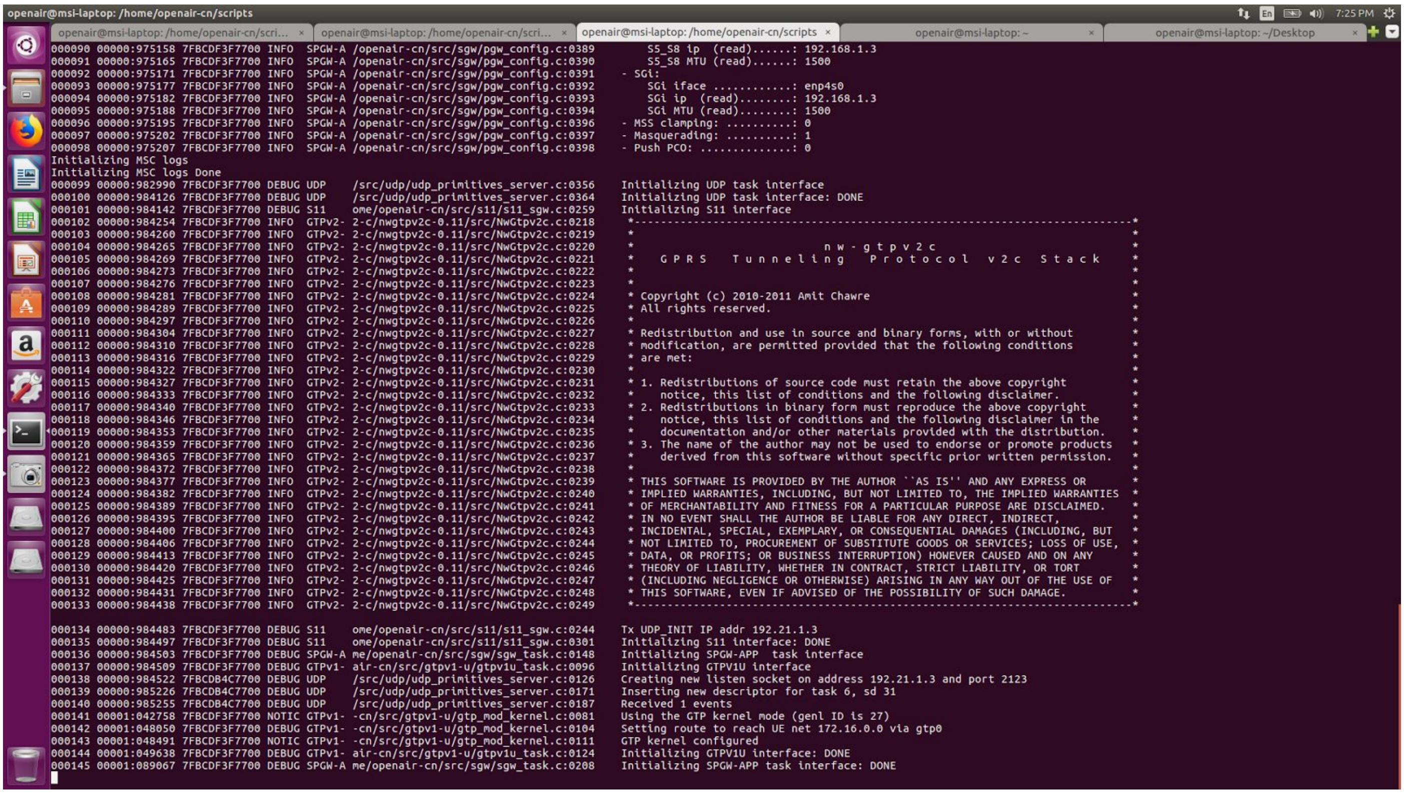
- MME Launch



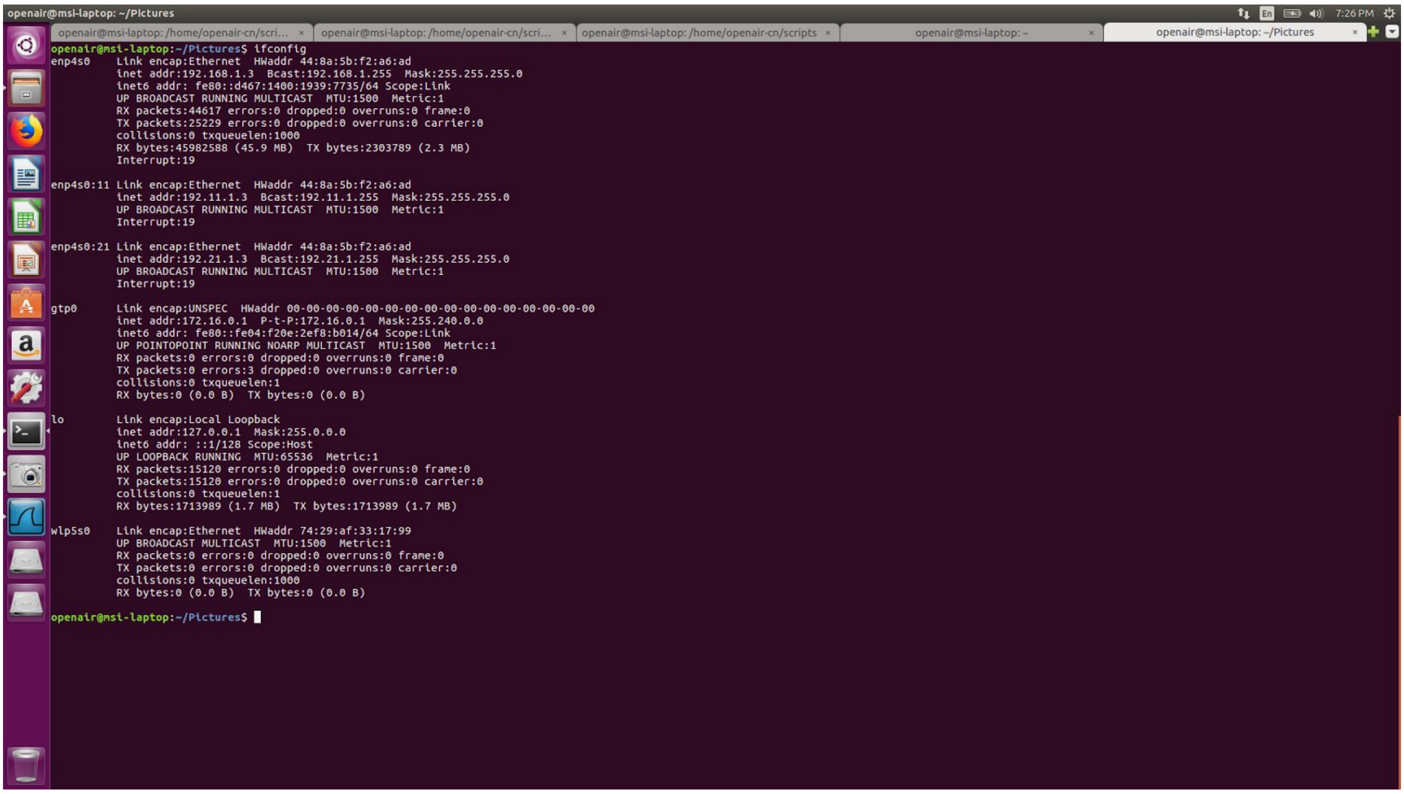
- HSS and MME Connection Setup



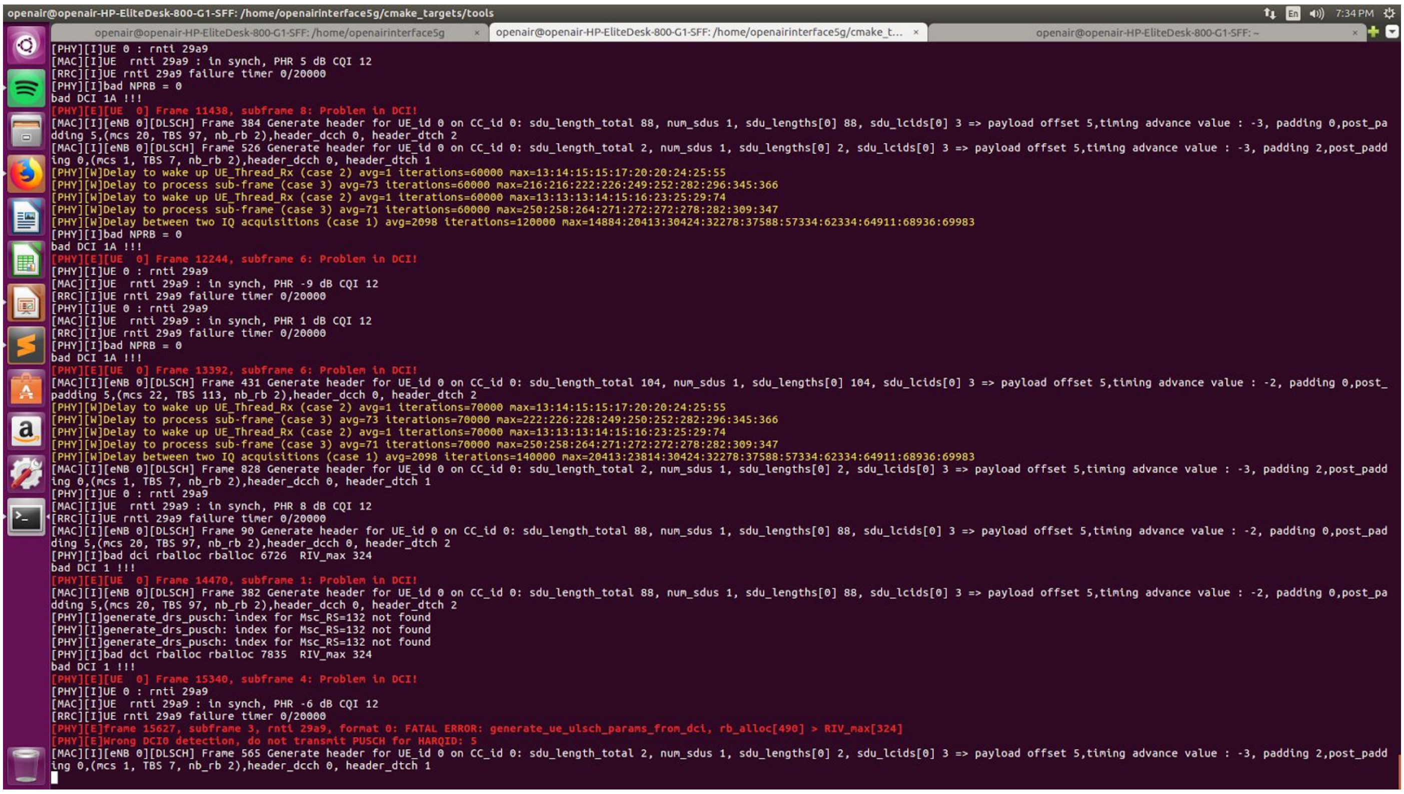
- SPGW Launch



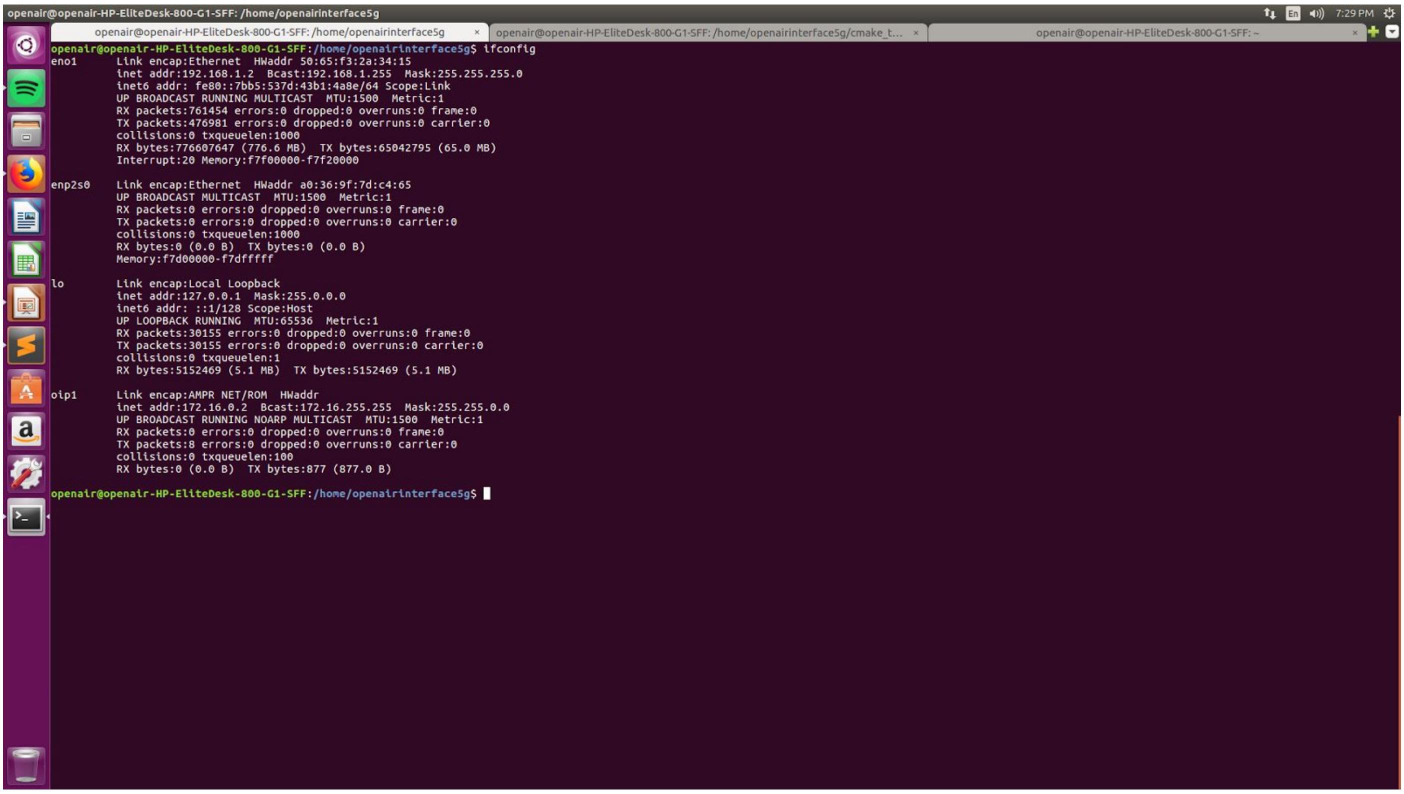
- EPC Network Interfaces



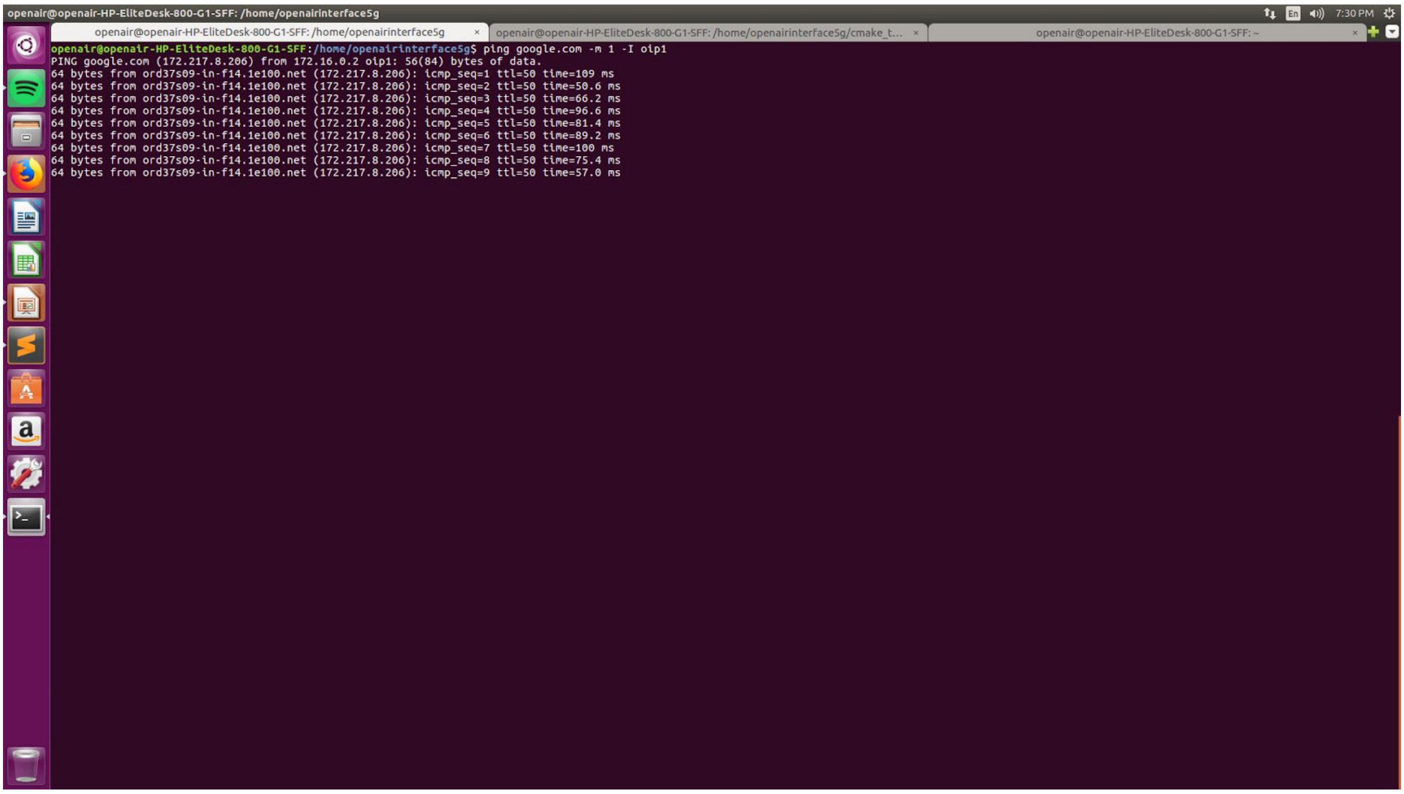
- OAISIM Launch



- OAISIM Machine Network Interfaces



- UE Internet Access



- EPC PING Packets Capture