Spring 2020 Semester Project :

Development of Simple Cloud Radio Application for 5G Platform

Project Supervisor : Florian Kaltenberger

Submitted By : Pavani Kolli

Table of Contents

Introduction

As we know that eurecom is currently deploying a 5G experimental platform based on the OpenAirInterface software and off-the-shelf hardware components. So this platform consists of a remote radio head (RRH) and antennas mounted on the top of Eurecom's building. The RRH is connected over fiber optics to Eurecom's server room that runs a RedHat Openshift cluster. All the radio and core networks functions run in a virtualized environment using Docker images. So the main goal of the project is to build simple programs that use the software defined radios (USRP) to do measurements. Later, need to build images containing the programs. Then based on that deploy the containers on the platform using Kubernetes/OpenShift.

Prerequisite Knowledge

Before starting the project we need to know about following things like:

1) The role of C- RAN in 5G
2) Open air Interface software
3) OpenShift
4) Kubernetes
5) Docker & Containers
6) OAI 5G cloud RAN platform based on docker

1. The role of C-RAN in 5G

The C-RAN also known as the centralized RAN, is cheaper for operators in terms of capex (capital expenditure) and opex (operation and maintenance expenditure). In C-RAN architecture, a pool of centralized baseband resources called BBUs (Baseband units) supplies digitized processed signals to remote simple cell-site elements called RRHs/RRUs (Remote Radio Heads/Units). The RRHs include analog components (PA, LNA) and typically bare-bone electronics (A/D conversion).

The BBU serves many RRHs and reduces or increase radio resource utilization as needed based on time-of-day demands and locations. The ability to manage resources across spatially distributed RRUs results in a net statistical gain, where a fixed set of resources does not need to be permanently allocated to a site.

In addition, the C-RAN architecture is better performing than distributed architectures because it offers better support for cooperative radio optimization techniques such as eICIC and CoMP. The C-RAN architecture also provides superior support for X2 handover because it occurs internally to the BBU and hence reduces the handover gap.

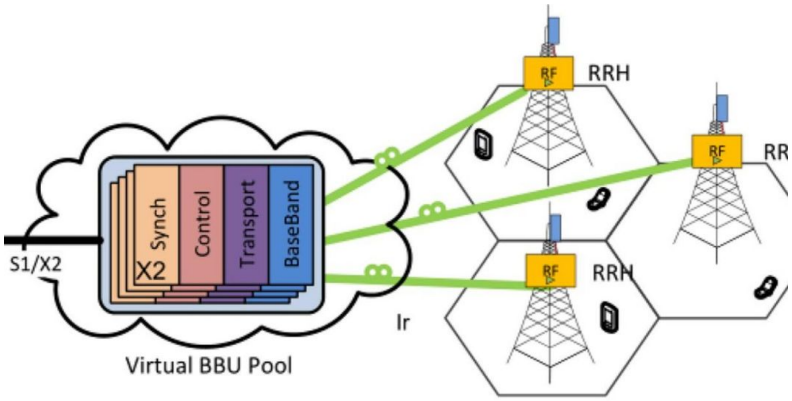


Fig . 1: C-RAN Architecture

Benefits

- ➢ The C-RAN can lower the total cost of ownership (TCO) and it can improve network performance.
- ➢ It is particularly beneficial in low-latency network scenarios.
- ➢ C-RAN include the ability to pool resources, reuse infrastructure, simplify network operations and managements, support multiple technologies, reduce energy consumption, and lower capex and opex.

2. Open air Interface software

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.

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. USRP) 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, 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.

3) OpenShift

OpenShift is a platform that allows you to run containerized applications and workloads and is powered by Kubernetes under the covers. Now the open source project that actually powers OpenShift is called OKD or Origin Community Distribution. OpenShift on the other hand, has multiple flavors and its an offering that comes with Red Hat Support regardless of where you choose to run your applications and workloads.

One of the big advantages is being able to take advantage of public and private resources for running OpenShift. That includes bare metal or virtualized hardware, whether it's on-premises or on a cloud provider. On top of that we are going to have the operating system which is generally Red Hat Enterprise Linux, but you can also use CentOS when working with OKD. On top of that we are going to have Kubernetes. So we will have the Kubernetes layer. Finally this is where OpenShift really comes in. It's a layer that's built on top of Kubernetes, and makes working with it much easier. OpenShift takes a lot of the difficult tasks, like deploying applications and doing things like the day-to-day operations, easier by building a web console in the CLI, as well as a facade on top of Kubernetes, to make all of those tasks just a little bit more streamlined and easy to do.

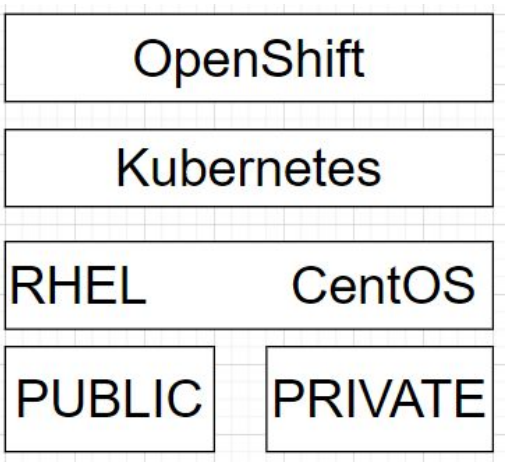


Fig.2: Architecture of installing with openshift looks like

4) Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

When you deploy Kubernetes, you get a cluster. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

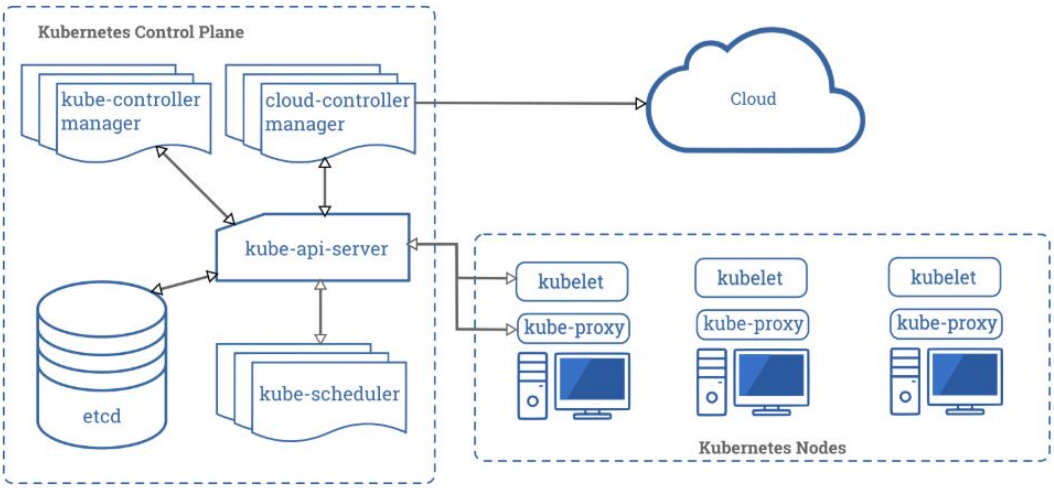


Fig.3: Diagram of a Kubernetes cluster with all the components tied together

5) Docker & Containers

Docker is a platform for developers and sysadmins to build, run, and share applications with containers. The use of containers to deploy applications is called containerization. Containers are not new, but their use for easily deploying applications is. Containerization is increasingly popular because containers are:

- Flexible: Even the most complex applications can be containerized.
- Lightweight: Containers leverage and share the host kernel, making them much more efficient in terms of system resources than virtual machines.
- Portable: You can build locally, deploy to the cloud, and run anywhere.
- Loosely coupled: Containers are highly self sufficient and encapsulated, allowing you to replace or upgrade one without disrupting others.
- Scalable: You can increase and automatically distribute container replicas across a datacenter.

- Secure: Containers apply aggressive constraints and isolations to processes without any configuration required on the part of the user.

Images and containers

Fundamentally, a container is nothing but a running process, with some added encapsulation, features applied to it in order to keep it isolated from the host and from other containers. One of the most important aspects of container isolation is that each container interacts with its own private filesystem; this filesystem is provided by a Docker image. An image includes everything needed to run an application- the code or binary, runtimes, dependencies, and any other filesystem objects required.

Containers and Virtual Machines

A container runs natively on Linux and shares the kernel of the host machine with other containers. It runs a discrete process, talking no more memory than any other executable, making it lightweight.

By contrast, a virtual machine (VM) runs a full-blown "guest" operating system with virtual access to host resources through a hypervisor. In general, VMs incur a lot of overhead beyond what is being consumed by your application logics.

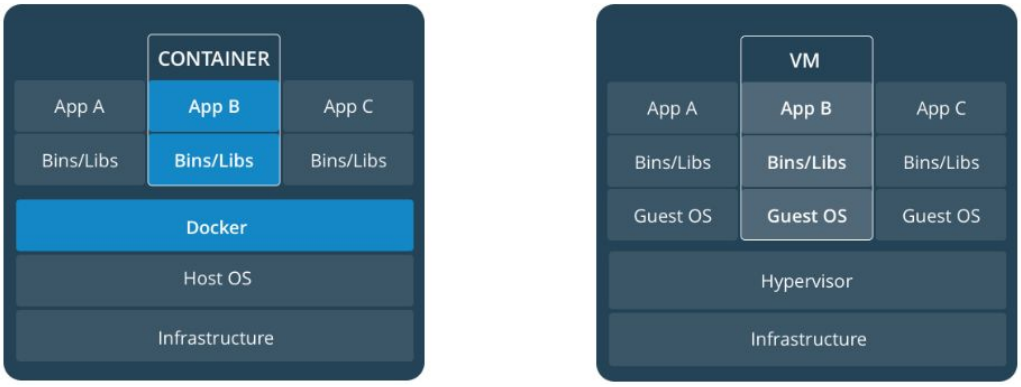


Fig. 4: Shows the difference between Containers and Virtual Machine

6) OAI 5G cloud RAN platform based on docker

The figure below shows the system architecture, where RRH and BBU modules of OAI were deployed on Docker containers. With the help of Dockerfile we can create Docker Images for BBU and RRU, with requirements based on OAI. BBU and RRH units for Kubernetes were made based on created Docker images. In order to deploy these units, StatefulSets were used. It's a Kubernetes controller used to scale a set of pods and to guarantee ordering and uniqueness of them. To establish host and pod communication, a Container Network Interface (CNI), was used to provide a layer 3 networking capability and associate a virtual router with each node. In order to

connect RRH with BBU, OAI requires some parameters: BBU, RRH, and EPC IP addresses and also SIM card information when OAISIM is used. So, to be able to dynamically scale the BBU and RRH, all needed information have to be stored on the ETCD, which is a distributed key-value store. When BBU or RRH run, it accesses the ETCD using the container name as a key taking all the values associated and storing its IP address. RRH run first and stores its IP address on ETCD, after, BBU can run and take this information. So the figure below shows a local cluster with three machines. One machine acts as Kubernetes master and holds all the BBU needed pods and a virtual machine for EPC, as depicted in figure below. The other two are worker nodes and will hold one RRH container each.

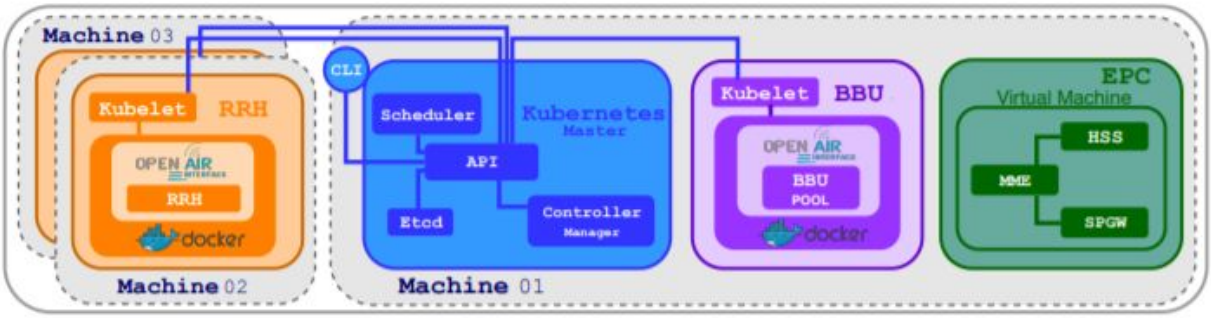


Fig.5: Represents OAI 5G cloud RAN platform based on docker

Methadology

Now coming back to the goal of the project that is to build program that use the software define radio (USRP) to do measurements using Docker, and also to build images containing the program. So with the help of above knowledge now i have to build a docker images using Dockerfile.

1) First step is to build a Dockerfile

Dockerfile

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

Instructions

- ARG - This instruction allows you to define variables that can be passed at build-time. You can also set a default value.
- FROM - The base image for building a new image. This instruction must be the first non-comment instruction in the Dockerfile. The only exception from this rule is when you want to use a variable in the FROM argument. In this case, FROM can be preceded by one or more ARG instructions.

- RUN - The commands specified in this instruction will be executed during the build process. Each RUN instruction creates a new layer on top of the current image.
- CMD - The command CMD, similarly to RUN, can be used for executing a specific command. However, unlike RUN it is not executed during the build, but when a container is instantiated using the image being built. It should be considered as an initial, default command that is executed (i.e. run) with the image-based creation of containers.
- The WORKDIR directive is used to specify where to execute the CMD-defined command.

Dockerfile with unmodifed rx_ascii_art_dft

```
ARG centos
FROM centos:7   ### here centos:7 is the base image

ARG GIT_TAG=latest

RUN yum install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm \
   yum install ncurses-devel ncurses \
   git make cmake gcc gcc-c++ autoconf automake bc bison flex libtool patch devtoolset-7 \
      atlas-devel \
      blas \
      blas-devel \
      boost \
      boost-devel \
      bzip2-devel \
      check \
      check-devel \
      doxygen \
      elfutils-libelf-devel \
      gflags-devel \
      glog-devel \
      gmp-devel \
      gnutls-devel \
      guile-devel \
      kernel-devel \
      kernel-headers \
      lapack \
      lapack-devel \
```

```
    libasan \
    libconfig-devel \
    libevent-devel \
    libffi-devel \
    libgcrypt-devel \
    libidn-devel \
    libidn2-devel  \
    libtool \
    libusb-devel \
    libusb1-devel \
    libusbx-devel \
    libboost-all-dev \
    libxml2-devel \
    libXpm-devel \
    libxslt-devel \
    libyaml-devel \
    lksctp-tools \
    lksctp-tools-devel \
    lz4-devel \
    mariadb-devel \
    nettle-devel \
    openssh-clients \
    openssh-server \
    openssl \
    openssl-devel \
    pkgconfig \
    psmisc \
    python-pip \
    python-mako \
    python-docutils \
    python2 \
    python2-docutils \
    python2-requests \
    unzip \
    vim-common \
    xforms-devel \
    xz-devel \
    zlib-devel \
```

```
RUN yum clean all && rm -rf /var/cache/yum

## uhd installation process starts
RUN set -xe; \
   cd /usr/local/src; \
   git clone https://github.com/EttusResearch/uhd.git;
WORKDIR /usr/local/src/uhd/
RUN git checkout $UHD_TAG; \
   cd host; \
   mkdir build; \
   cd build; \
   cmake ../; \
   make; \
   make install; \
   ldconfig; \
   echo "export LD_LIBRARY_PATH=/usr/local/lib64" >> ~/.bashrc;
RUN cd /usr/local/lib64/uhd/utils; \
   uhd_images_downloader;

### executing the spectrum analysis example in the container
WORKDIR /usr/local/src/uhd/host/build/examples
RUN echo $PWD
CMD ["./rx_ascii_art_dft","--freq","3.64e9","--rate","122.88e6","--gain","20","--bw","80e6","--ref-lvl","-30"]
```

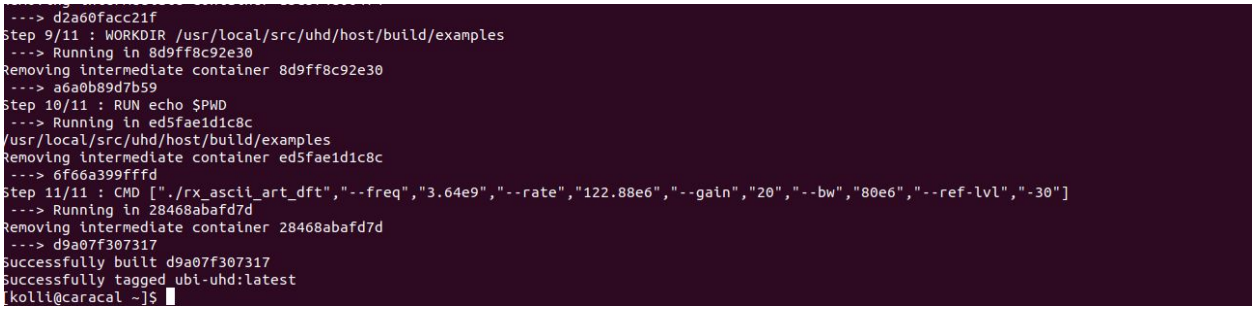
2) Building the Image

The next step is to build the image. To do so run the following command from below inorder to build image using the Dockerfile.

```
sudo docker image build -t ubi-uhd:latest /home/kolli
```

From the above command “-t” represents tag, “ubi-uhd” is the repository name, “latest” is the tag and “/home/kolli” is the path of the Dockerfile.

The output of the build process will look something like this:

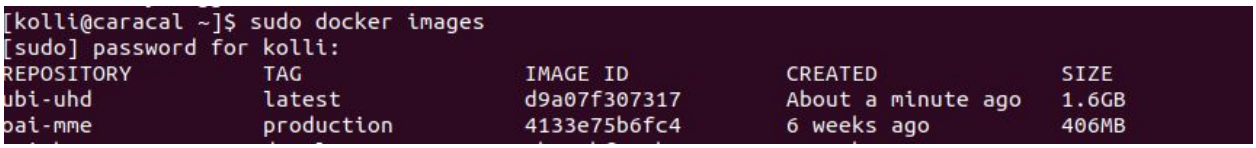


If we observe the execution procedure then we will find that it will execute step by step process. And once the image is build then it will show successfully built with the image id.

When the build process is completed the new image will be listed in the image list:

Sudo docker images

On executing the above command we will find the image we have built which is d9a07f307317 and also we can see the image with repository name ubi-uhd and tag latest.



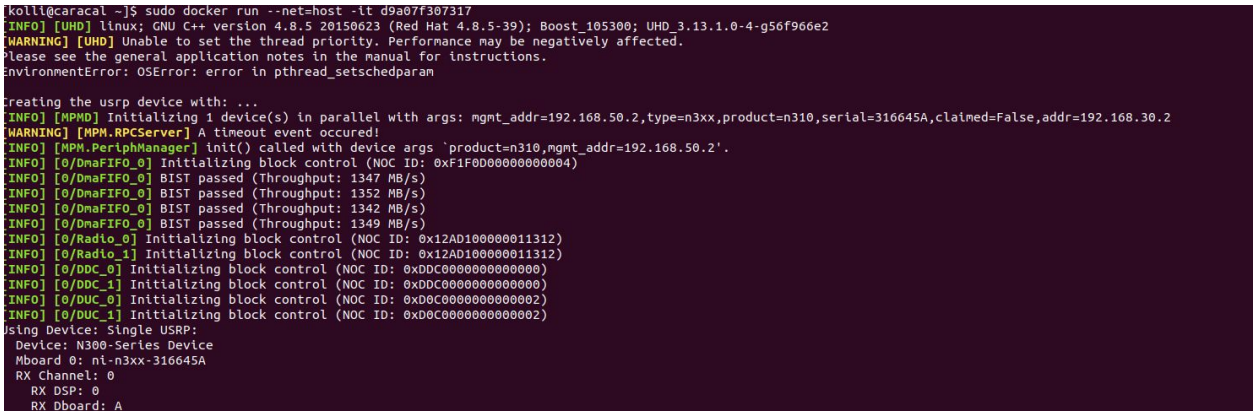
3) Running a Container

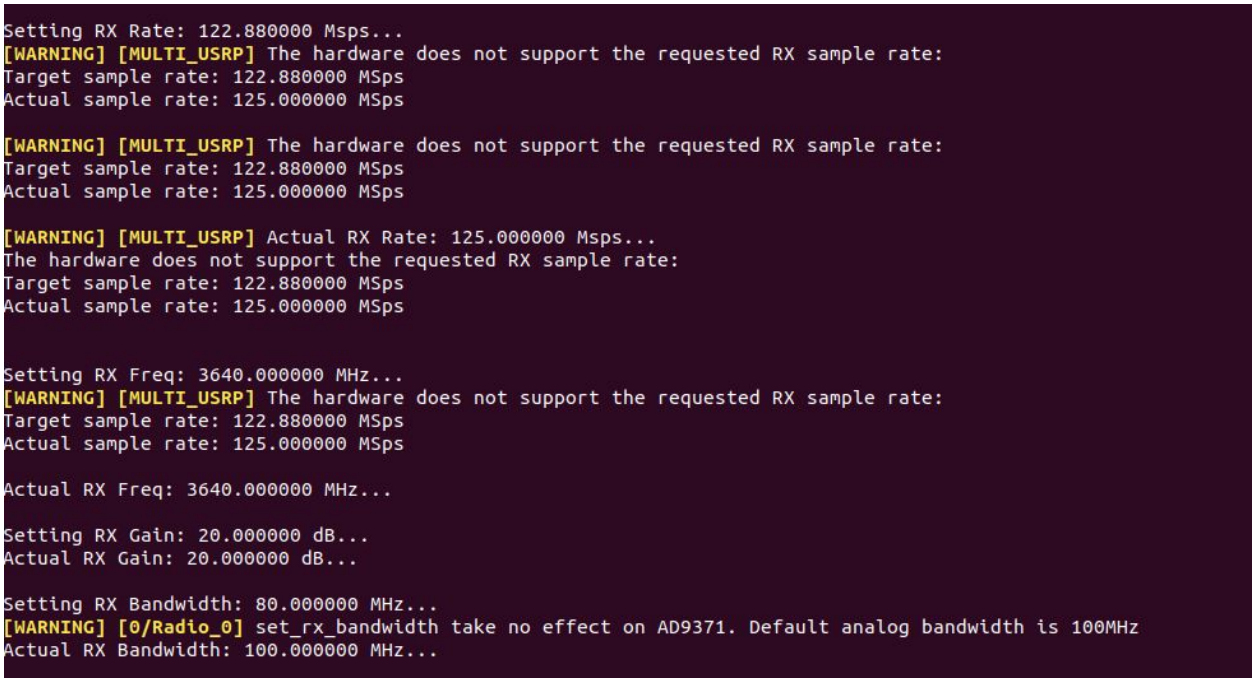
Now that the image is created and next thing is that we have to run a container from it by running:

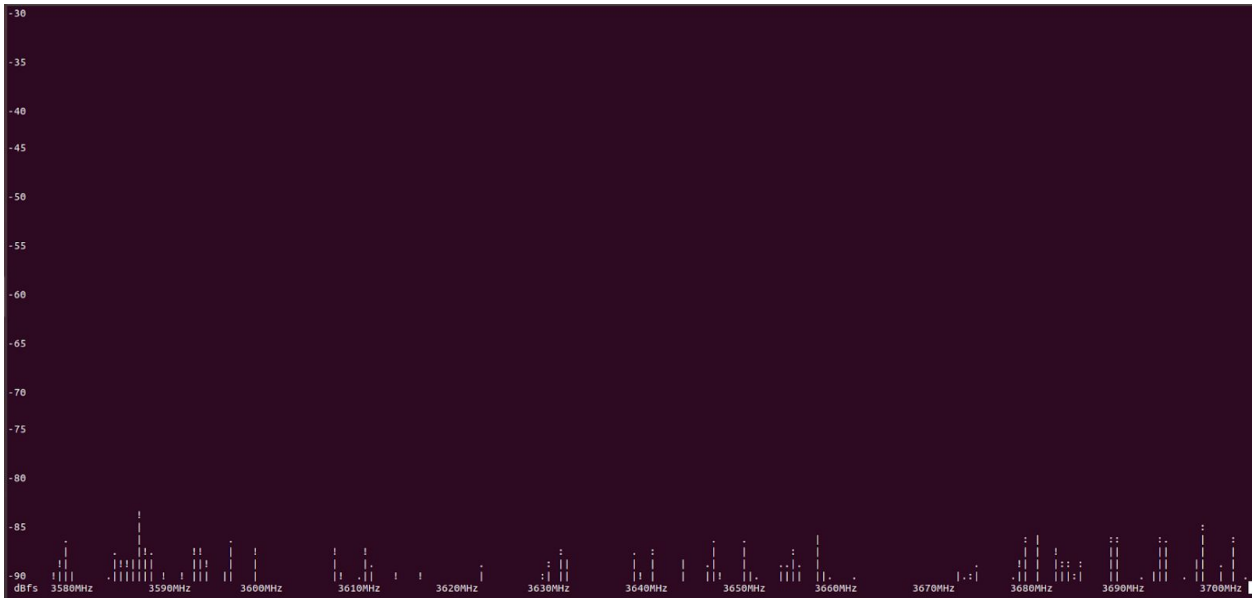
sudo docker run --net=host -it d9a07f307317

When you use --net=host it tells the container to use the hosts in the network stack. “-it” is short for “--interactive + --tty “ when we do docker run with this command then it would take us straight inside of the container

Below is the output we get when we run the container for unmodified rx_ascii_art_dft.cpp







Dockerfile with Modified rx_ascii_art_dft.cpp

```
FROM centos:7

ARG GIT_TAG=latest


RUN yum install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm \
    yum install ncurses-devel ncurses \
      git make cmake gcc gcc-c++ autoconf automake bc bison flex libtool patch devtoolset-7 \
        atlas-devel \
        blas \
        blas-devel \
        boost \
        boost-devel \
        bzip2-devel \
        check \
        check-devel \
        doxygen \
        elfutils-libelf-devel \
        gflags-devel \
        glog-devel \
        gmp-devel \
        gnutls-devel \
```

```
        guile-devel \
        kernel-devel \
        kernel-headers \
        lapack \
        lapack-devel \
        libasan \
        libconfig-devel \
        libevent-devel \
        libffi-devel \
        libgcrypt-devel \
        libidn-devel \
        libidn2-devel  \
        libtool \
        libusb-devel \
        libusb1-devel \
        libusbx-devel \
        libxml2-devel \
        libXpm-devel \
        libxslt-devel \
        libyaml-devel \
        lksctp-tools \
        lksctp-tools-devel \
        lz4-devel \
        mariadb-devel \
        nettle-devel \
        openssh-clients \
        openssh-server \
        openssl \
        openssl-devel \
        pkgconfig \
        psmisc \
        python-pip \
        python-mako \
        python2 \
        python-docutils \
        python2-docutils \
        python2-requests \
        unzip \
        vim-common \
        xforms-devel \
        xz-devel \
        zlib-devel \
RUN yum clean all && rm -rf /var/cache/yum
```

```
RUN set -xe; \
    cd /usr/local/src; \
    git config --global http.postBuffer 524288000; \
    git clone https://github.com/Kolli-1970/uhd.git;
WORKDIR /usr/local/src/uhd/
RUN git checkout UHD-3.13; \
    cd host; \
    mkdir build; \
    cd build; \
    cmake ../; \
    make; \
    make install; \
    ldconfig; \
    echo "export LD_LIBRARY_PATH=/usr/local/lib64" >> ~/.bashrc;
RUN cd /usr/local/lib64/uhd/utils; \
    uhd_images_downloader;
WORKDIR /usr/local/src/uhd/host/build/examples
RUN echo $PWD
CMD
["./rx_ascii_art_dft","--freq","3.64e9","--rate","122.88e6","--gain","20","--bw","80e6","--ref-lvl","-30"]
```

Below is the output we get when we run the container for modified rx_ascii_art_dft.cpp

One set has values for 512 bins. Centre bin will be the center freq we pass as parameter i.e 3.64e9. Left bins will be -40 MHz and right bin will be +40 MHz.. So basically first value is the db for 3.64e9 - 40MHz and the last value is 3.64e9 +40MHz. Each bin will correspond to 40 MHZ/512 =78125 Hz… So second bin will 3.64e9 - 40e6 + 78125Hz and so on …

Changes made in /uhd/host/examples/rx_ascii_art_dft

```
    //calculate the dft and create the ascii art frame
    ascii_art_dft::log_pwr_dft_type lpdft(
        ascii_art_dft::log_pwr_dft(&buff.front(), num_rx_samps)
    );

    const size_t num_bins = lpdft.size() - 1 + lpdft.size() % 2; // make it odd
    ascii_art_dft::log_pwr_dft_type dft(num_bins);
    for (size_t n = 0; n < num_bins; n++) {
    dft[n] = lpdft[(n + num_bins / 2) % num_bins];
    printf("%f,",dft[n]);
    }
    printf("\n");

    /*std::string frame = ascii_art_dft::dft_to_plot(
        lpdft, COLS, LINES,
        usrp->get_rx_rate(),
        usrp->get_rx_freq(),
        dyn_rng, ref_lvl
    );

    //curses screen handling: clear and print frame
    clear();
    printw("%s", frame.c_str());

    //curses key handling: no timeout, any key to exit
    timeout(0);
    int ch = getch();
    if (ch != KEY_RESIZE and ch != ERR) break;*/
}

//---------------------------------------------------------------
//-- Cleanup
//---------------------------------------------------------------
```

```
rx_stream->issue_stream_cmd(uhd::stream_cmd_t::STREAM_MODE_STOP_CONTINUOUS);
    endwin(); //curses done

    //finished
    std::cout << std::endl << "Done!" << std::endl << std::endl;

    return EXIT_SUCCESS;
}
```

References

[1]https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html-single/getting_started_with_containers/index?extIdCarryOver=true&sc_cid=701f2000001OH7EAAW#using_red_hat_universal_base_images_standard_minimal_and_runtimes

[2]https://github.com/OPENAIRINTERFACE/openair-k8s/blob/master/images/oai-build-base/Dockerfile.rhel7

[3]https://www.howtoforge.com/tutorial/how-to-create-docker-images-with-dockerfile

[4]https://files.ettus.com/manual/page_build_guide.html

[5]https://arxiv.org/pdf/2001.08992v1.pdf

[6]https://dev.to/johnny_pravi/oais-core-network-based-on-containers---3jkd

[7]https://www.sharetechnote.com/html/SDR_OpenAirInterface.html

[8]https://kubernetes.io/docs/concepts/