POLYTECHNIC OF TURIN

Master's Degree in Electronic Engineering

Master's Degree Thesis

5G-Enabled UAV System: Implementation, Deployment and Analysis

Supervisors
Prof. Carla Fabiana CHIASSERINI
Dr. Corrado PULIGHEDDU

Candidate
Lorenzo BRAVI

OCT 2024

Abstract

With the growing adoption of Unmanned Aerial Vehicles (UAVs) across various industries, the need for reliable, real-time control of these systems has become more critical than ever. UAVs are now utilized in various use-cases ranging from civilian to military, many of which demand low-latency highly reliable communication between the drone and its control system. However, conventional communication networks like 4G LTE and earlier technologies often fall short of meeting the demanding needs of these applications.

In this context, a key enabling technology is edge computing: in the edge computing paradigm, computation and data storage are brought closer to the location where they are needed, typically just outside the network core. This has the main benefit of reducing the latency of communication towards the User Equipment (UE) compared to using cloud servers and allows for real-time processing. Concerning UAV operations, edge computing also enables offloading resource-intensive tasks, such as trajectory calculation and image processing, to local servers, thereby reducing the computational load on the UAV itself and extending its battery life.

This thesis seeks to demonstrate the deployment and performance of a 5G-enabled UAV system. The study aims to explore the challenges involved in its implementation and assess its capacity to deliver reliable and real-time control in UAV operations.

The first aspect that is investigated is the deployment of an OpenAirInterface 5G network leveraging Ettus USRP B210 software-defined radios. After the correct operation of the network is validated, the thesis moves on to examine a second critical component: the PX4 Autopilot framework, an open-source flight control software providing a simulation environment and integration with Robot Operating System 2 (ROS2).

Having laid the theoretical background, the complete deployment of the 5G-enabled UAV testbed is documented, detailing the development of the ROS2 UAV control application and proposing solutions to the issues that arise, especially regarding the ROS2 Middleware for communications. The experimental UAV testbed is then analyzed in terms of trajectory error, occupied network bandwidth and ROS2 application round-trip time (RTT). The study proposes a method to measure RTT based on talker/listener ROS2 nodes and researches the causes of the high variability of RTT over time by gathering and analyzing the behavior of RAN KPIs. Finally, trajectory error between the desired and actual flight paths is evaluated, in particular when varying angular velocity over the circular trajectory and control publishing frequency. The analysis identifies a critical angular velocity over which the UAV is not controlled effectively and suggests a value for the control publishing frequency.

Since the thesis makes use of a simulated UAV, future work could concentrate on repeating the performance evaluation with an actual UAV, as well as applying a URRLC network slice to the UE and employing more advanced edge-computing frameworks (such as Multi-access Edge-Computing) to further improve trajectory error.

Acknowledgements

The work of this thesis was performed at the Advanced Wireless Experience (AWE) lab at Politecnico di Torino. Many thanks to Prof. Carla Fabiana Chiasserini and Dr. Corrado Puligheddu for allowing me to work on this thesis under their supervision.

I would also like to express my deep gratitude to PhD student Yen-Chia Yu (Rex) for his unwavering support and altruistic attitude.
Finally, special thanks to my friend and fellow colleague Davide Boggio Marzet for his collaboration throughout the ups and downs of this journey.

Table of Contents

Chapter 1

Introduction

Fifth-generation mobile networks (5G) represent a significant leap in wireless communication technology. 5G offers higher data rates, lower latency, and higher reliability compared to its predecessors, making it ideal for applications that require real-time communication and fast data processing.
An example of such applications is the control Unmanned Aerial Vehicles (UAVs). A UAV, commonly referred to as a drone, is an aircraft that operates without a human pilot on board, via radio control or in a completely autonomous fashion using onboard computers and sensors. They are widely used for a range of applications, going from civilian to military uses, from agriculture and disaster response to search-and-rescue and mapping.
In this context, a key enabling technology is edge computing: in the edge computing paradigm, computation and data storage are brought closer to the location where they are needed, typically just outside the network core. This has the main benefit of reducing the latency of communication towards the User Equipment compared to using cloud servers and allows for real-time processing. Concerning UAV operations, edge computing enables offloading resource-intensive tasks, such as trajectory calculation and image processing, to local servers, thereby reducing the computational load on the UAV itself and extending its battery life.

1.1 Problem Statement and Research Objective

With the growing adoption of Unmanned Aerial Vehicles (UAVs) in various industries, the need for reliable, real-time control of these systems has become more critical than ever. UAVs are increasingly employed in applications ranging from aerial surveillance to logistics, most of which require seamless communication between the drone and the operator or control system. However, traditional communication technologies, including 4G LTE and earlier networks, struggle to meet the stringent requirements of these use cases, particularly in terms of latency and bandwidth.
While 4G technology offers moderate data throughput and latency, it lacks the ability to support ultra-reliable and low-latency communications (URLLC), which are of great aid in real-time UAV control. The introduction of 5G technology, with its promise of enhanced mobile broadband (eMBB) and URLLC, presents a potential solution to these issues.

This thesis seeks to demonstrate the deployment and performance of a 5G-enabled UAV system. The study aims to explore the challenges involved in its implementation and assess its capacity to deliver reliable and real-time control in UAV operations. By doing so, this research aims to contribute to the growing body of knowledge on leveraging 5G technology for UAV applications.

Chapter 2

5G System Overview

5G refers to the fifth generation of mobile telephony, a system defined and standardized by the *Third Generation Partnership Project* (3GPP).
3GPP is a consortium made up of a number of standards organizations: it produces complete specifications not only for the air interface (PHY+DATA layers), but also for all network interfaces crucial for the operation of mobile systems. This encompasses call and session control, mobility management, service provisioning, and other essential functionalities. Such comprehensive standards enable 3GPP networks to seamlessly operate across different vendors and operators, which aided in the global convergence towards 3GPP specifications.
3GPP was formed in 1998 and is best known for producing complete specifications of:

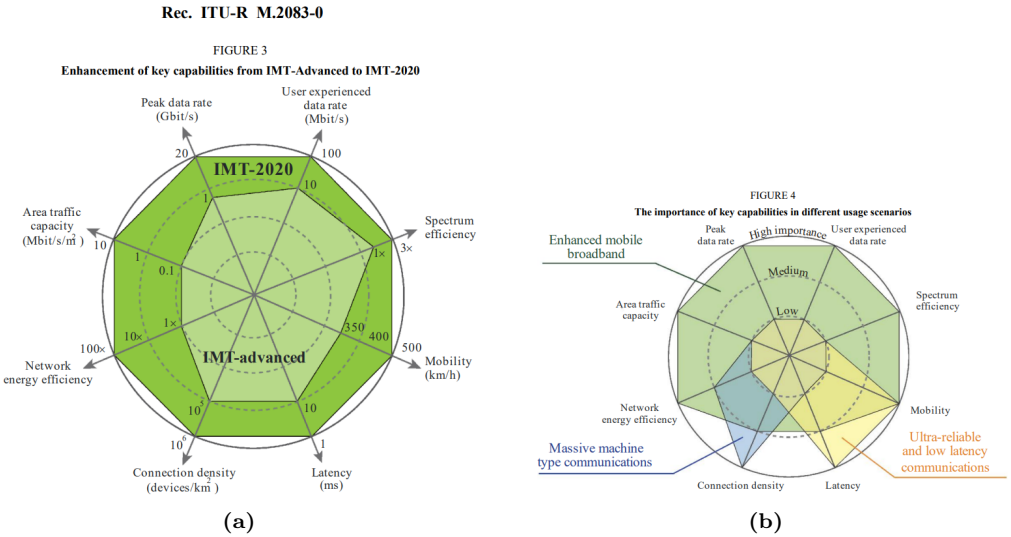
- 2G: GSM, GPRS, EDGE
- 3G: UMTS, HSPA, HSPA+
- 4G: LTE, LTE Advanced, LTE Advanced Pro
- 5G: NR, 5G Advanced

5G enhances 4G services across various dimensions:

- **Enhanced Mobile Broadband (eMBB)**: 5G specifies higher data rates, offering up to 50 Mbps for outdoor and 1 Gbps for indoor (5GLAN) in the downlink, with half of these values available for the uplink. Case studies, including aviation, demonstrate 5G's capability to deliver a bitrate of 1.2 Gbps to airborne flights.
- **Critical Communications (CC) and Ultra Reliable and Low Latency Communications (URLLC)**: In contexts requiring a highly reliable link, 5G can ensure a reliability of 99.9999% and end-to-end latency of 50 ms. Edge computing (describe in section 3.1) plays a vital role in achieving these requirements.
- **Massive Internet of Things (mIoT)** : 5G supports scenarios with very high traffic densities of devices

It is important to understand that the key advantage of 5G lies in its **flexibility**. Taking a glance at figures 2.1, it can be noted that 5G expands 4G capabilities by about and order of magnitude in each metric: a single user cannot experience all these improvements *simultaneously*, but the 5G cell is able to push the limits on certain KPIs while relaxing specifications on others.

In this regard, an important innovation lies in the ability to operate **network slicing**: the overall network can be split into a number of different sub-networks which serve different UEs with different KPIs.
5G's flexibility and adaptability open possibilities for different use cases, enabling expansion to new sets of markets, the **verticals**. Examples of verticals include automotive, media streaming and robotics. [1]



(a) **(b)**

Figure 2.1: (a) Enhancement of key capabilities from IMT-Advanced to IMT-2020, (b) Importance of key capabilities in different usage scenarios

2.1 System Architecture

As in previous generations, the 5G system (5GS) is composed of a radio access network (NG-RAN), a core network (5GC, or CN in general) and a variable number of user equipments (UEs).

The CN and the RAN are two essential components of a telecommunications system.
On one hand, the core network acts as the central hub, responsible for managing user connections, authenticating devices, routing data, and ensuring secure communication. It connects access networks like 4G, 5G, or Wi-Fi, to external systems such as the internet. The core network is also where essential **network functions** are performed, such as mobility management, billing, service orchestration and security.
On the other hand, the RAN connects end-user devices (UEs) to the core network through radio signals. It consists of base stations that manage the radio frequency spectrum, signal transmission, and eventually handovers when devices move between cell towers.

It is important to note that two deployment models are specified for 5G:

1. Non-Standalone (NSA) architecture: in this setup, the 5G RAN with its New Radio (NR) interface operates alongside the existing LTE and Evolved Packet Core (EPC) infrastructure (i.e., the 4G RAN and 4G Core). This allows the use of 5G NR technology without needing

to replace the entire network. While only 4G services are supported in this architecture, they benefit from the enhanced capabilities of the 5G NR, such as reduced latency and increased bandwidth

2. Standalone (SA) architecture: refers to a pure 5G network, where the NR interface connects directly to the 5GC. This unlocks all the advanced features of 5G technology

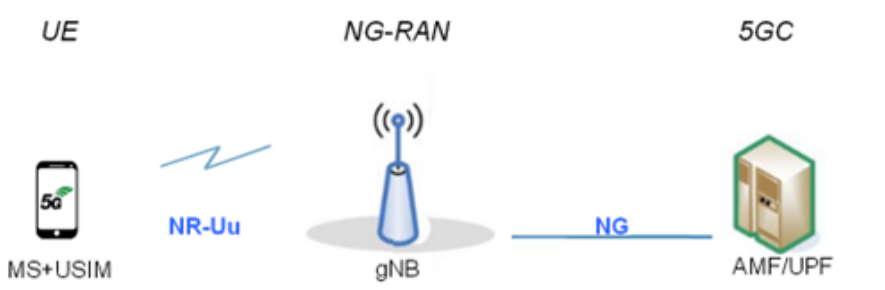


Figure 2.2: 5G system basic overview

As can be observed in figure 2.2, the NG-RAN is based on one or more instances of a gNB (the 5G base station), while UEs connect to the RAN via the radio interface NR-Uu.

Concerning the 5GC, its two most powerful innovations lie in **Network Function Virtualization** (NFV) and **Software-defined Networking** (SDN), which are the two enabling technologies for **network slicing**.

4G and previous generations of mobile telephony relied on proprietary, hardware-based appliances for each network function (NF). NFV enables these functions to be virtualized and deployed flexibly on general-purpose server. This approach brings greater agility, scalability and cost efficiency to network management.

SDN is a networking architecture that separates the control plane, which determines how data packets should be routed, from the data plane, which handles the actual forwarding of packets. This decoupling allows for centralized, software-based control of the network, providing a simpler, more flexible and cost-effective way to manage the network compared to the traditional approach.

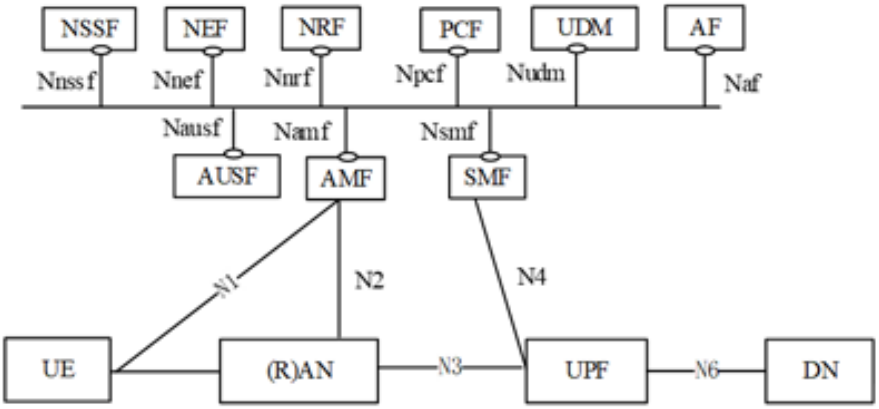


Figure 2.3: 5G system showing the CN functions

Figure 2.4 shows the 5GS with an emphasis on the main NFs of the CN. The upper part of figure 2.4 shows the NFs involved in the control plane, while the bottom part the NFs and elements pertaining to the user plane. The interaction between NFs is facilitated via APIs, where each NF offers services to other NFs (service-based architecture).
Some notable elements include:

- The already introduced UE, NG-RAN
- External Data Nework (ext-DN): refers to a network that is external to the 5G system, such as for example the public internet
- User Plane Function (UPF): manages data transfer towards/from the external DN and enforces QoS
- Access and Mobility management Function (AMF): handles UE registration, authentication, authorization and mobility inside the network (e.g. handovers)
- Session Management Function (SMF): handles the establishment, maintenance and termination of data sessions for UEs, including IP address assignment
- Policy Control Function (PCF): manages policies related to quality of service (QoS) and resource allocation.
- Network Slice Selection Function (NSSF): selects and manages network slices for different services and users

A more complete list of 5G NFs is provided in [2] (page 42).

2.2 RAN fundamentals

This section introduces a selection of concepts related to the 5G RAN that will prove useful later in the thesis.

2.2.1 OFDM

5G's modulation technique is the same as LTE, Orthogonal Frequency Division Multiplexing (OFDM). In short, OFDM transmits symbols concurrently across closely spaced channels, each characterized by a narrow bandwidth. This helps to mitigate multipath effects, as each subchannel will have almost flat frequency response. The waveforms utilized for symbol transmission on each subcarrier are orthogonal, meaning that the scalar product of two waveforms operating at different frequencies results in zero. [3]

2.2.2 Frequency Ranges and Numerology

5G utilizes two primary frequency ranges to achieve varying performance characteristics.
The first is the Sub-6 GHz range (FR1), which encompasses frequencies from 450 MHz to 6 GHz. This range includes traditional cellular bands and is characterized by broader coverage and better penetration through obstacles.
The second range is Millimeter Wave (FR2), covering frequencies from 24.25 GHz to 52.6 GHz. FR2 supports exceptionally high data rates and capacity but is limited by its shorter range and reduced penetration capabilities.

In 5G, numerology refers to the different configurations of subcarrier spacing and related parameters in the OFDM (Orthogonal Frequency Division Multiplexing) waveform, tailored to support a variety of use cases and frequency ranges. 5G uses scalable numerologies, defined by a parameter μ, which determines subcarrier spacing Δf.
The subcarrier spacing can be adjusted to accommodate different services and operating frequencies. For example, lower subcarrier spacings (like 15 kHz) are used for sub-6 GHz frequencies, as multipath effects are prevalent at these frequencies. Larger subcarrier spacings (like 120 kHz) are employed at higher frequencies, such as millimeter waves, to minimize latency (because symbol duration is shorter). [4]

Numerology (μ)	**Subcarrier Spacing (Δf)**
0	15 kHz
1	30 kHz
2	60 kHz
3	120 kHz
4	240 kHz

Table 2.1: 5G Numerologies and their corresponding subcarrier spacings

2.2.3 Frame Structure and PRBs

Downlink and uplink transmissions are organized into frames, each with a duration of $T_f = 10$ ms, which are divided into ten subframes, each lasting $T_{sf} = 1$ ms. Each subframe is further partitioned into a number of slots, where each slot comprises 14 consecutive OFDM symbols. The slot serves as the fundamental scheduling unit in the time domain. The duration of each slot is inversely proportional to the subcarrier spacing, given by the formula: *Slot length* $= \frac{1 \text{ ms}}{2^\mu}$. Consequently, the number of slots within a subframe is dependent on the value of μ.

In this context, a Physical Resource Block (PRB) is defined as a set of 12 consecutive OFDM subcarriers in the frequency domain. Therefore, the size of a PRB is dependent on the employed

numerology; PRBs serve as the fundamental unit for scheduling resources within the frequency domain.

Time Division Duplex (TDD) and Frequency Division Duplex (FDD) are two key duplexing methods used for managing uplink and downlink transmissions. TDD employs a single frequency band, alternating in the *time domain* between uplink and downlink transmissions. This allows for flexible adjustment of the uplink and downlink ratio based on traffic demands, making TDD suitable for asymmetric traffic patterns. In contrast, FDD uses separate frequency bands for uplink and downlink, providing simultaneous bidirectional communication.
The NG-RAN supports both TDD and FDD operations by providing the possibility to configure a slot in various ways: it may consist entirely of downlink transmissions, with all OFDM symbols allocated for downlink; entirely of uplink transmissions, with all OFDM symbols allocated for uplink; or a combination of both. [4]

2.2.4 Physical Channels and Signals

Physical channels and signals are essential for the transmission and reception of data across the network. Physical channels refer to the standardized pathways through which data is transmitted, different channels are allocated over different subcarriers and over different parts of the time slot. Simoultaneously, some sub-carriers are allocated to the transmission of signals, which are used for synchronization, channel estimation and aiding in the establishment of a reliable connection between the UE and gNB.

- **Downlink Physical Channels:**
  - **PDSCH:** Downlink Shared Channel
  - **PBCH:** Downlink Broadcast Channel
  - **PDCCH:** Downlink Control Channel
- **Downlink Physical Signals:**
  - Demodulation Reference Signals (DMRS)
  - Phase Tracking Reference Signals (PTRS)
  - Channel State Information Reference Signals (CSI-RS)
  - Synchronization Signals (PSS, SSS)
- **Uplink Physical Channels:**
  - **PUSCH:** Uplink Shared Channel
  - **PRACH:** Uplink Random Access Channel
  - **PUCCH:** Uplink Control Channel
- **Uplink Physical Signals:**
  - Demodulation Reference Signals (DMRS)
  - Phase Tracking Reference Signals (PTRS)
  - Sounding Reference Signals (SRS) [4]

2.3 Cloud Computing and Edge Computing

Cloud computing refers to the delivery of various services (such as data storage, processing power, and networking) over the internet ("the cloud"). It allows users to access and store data or run applications without the need for dedicated hardware or infrastructure. The core idea is resource pooling, where large-scale data centers handle computing tasks and storage, offering services like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Mobile cloud computing (MCC) extends these principles to mobile devices. Its focus is on enabling smartphones, tablets, and other portable devices to run more complex applications by offloading computational tasks to remote cloud servers. This approach helps conserve battery life and device resources, since the cloud handles tasks that would otherwise exhaust the mobile device's limited processing power and storage. MCC is particularly useful for applications that involve heavy data processing, like video streaming or real-time analytics, but faces challenges such as bandwidth limitations, latency issues, and service availability across various networks. [5]

Edge computing aims to solve these issues by shifting from cloud-centric models to more localized processing: cloud capabilities closer to the UE through the deployment of edge servers in mini clouds at the network's edge. This proximity significantly reduces latency and enhances security, enabling to meet the stringent quality of service (QoS) requirements of next-generation applications like augmented reality, virtual reality and robotics.
Nonetheless, edge computing comes with certain challenges that increase complexity. One major issue is its lower geographical availability, as applications are deployed on local servers, limiting their accessibility to users in other regions. This can lead to consistency issues between different geographical deployments, where data or service discrepancies arise due to the decentralized nature of edge computing.
Additionally, service migration becomes more complex as services move between different edge locations, requiring careful management to avoid disruptions.

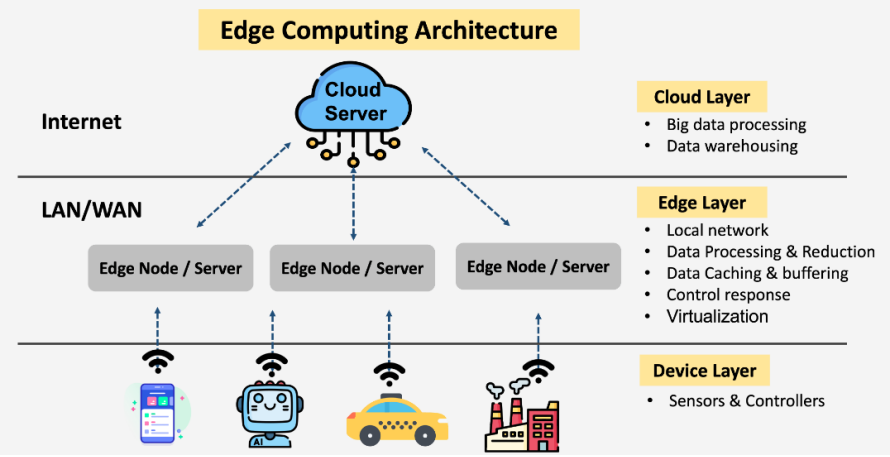


Figure 2.4: Edge computing infographic [6]

In the context of 5G, edge computing serves 5 main objectives: it improves data management by handling large volumes of real-time data locally, thereby reducing the burden on centralized data centers; it enhances QoS by meeting the low-latency and high-throughput demands of highly

interactive applications; it predicts network demand for optimal resource allocation, ensuring efficient use of limited network resources; it manages location awareness to provide personalized, context-based services; and it improves resource management by optimizing network resource utilization.
Various computational platforms support edge computing in 5G, including **multi-access edge computing** (MEC), which provides storage and computational capabilities at the edge, often co-located with base stations, and fog computing, which utilizes local hardware devices like routers and switches for computation. [7]

2.3.1 Multi-access Edge Computing

MEC is an evolution of traditional edge computing that is specifically designed for mobile networks, with a key difference being that it exploits information related to the radio link. While standard edge computing reduces latency by processing data closer to the user, MEC goes further by integrating with the mobile network infrastructure itself. This allows MEC to take advantage of real-time insights into the radio link, such as signal strength or bit error rates. This feature makes MEC ideal for applications like video streaming or augmented reality, where the ability to adapt to radio link conditions in real-time is particularly useful.

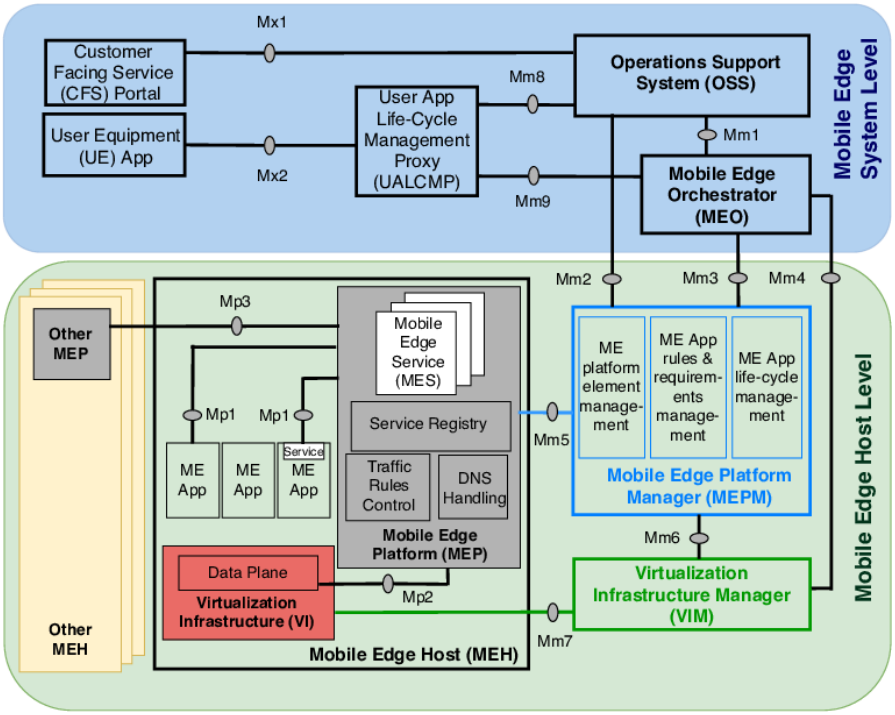


Figure 2.5: MEC System Reference Architecture [8]

The MEC system reference architecture (reported in figure 2.5) consists of two high-level entities:

- MEC host, which is composed of:
  - MEC platform, which provides essential functionalities to run MEC applications and provide MEC services to such applications.

Among MEC services, two critical ones are the Radio Network Information Service (RNIS) and the Location Service (LS).

RNIS gives authorized applications access to real-time radio network data, including details about the UE and its associated radio access bearers (RABs). The Location Service (LS), on the other hand, provides precise location data for UEs, including geolocation and Cell ID information. LS can cater to specific needs, such as identifying the location of all UEs in a particular area or tracking the location of a specific UE

  - Virtualization infrastructure, which provides compute power, storage, and network resources needed to run the MEC applications, which are hosted on top of the infrastructure in the form of VMs or containers. An important fact is that the virtualization infrastructure includes a data plane that handles traffic routing between MEC applications, the 5G network, local networks and external networks

- MEC management, which comprises of:
  - MEC system-level management, responsible for supervising the entire MEC system via the MEC orchestrator, which is responsible for a number of tasks. Firstly, it maintains an overall view of deployed MEC Hosts and Services, available resources and topology. In addition, it handles tasks such as preparing the virtualization infrastructure, selecting appropriate MEC hosts for app instantiation, triggering app instantiation/termination and eventually execute app relocation from one MEC host to another
  - MEC host-level management, responsible for managing the functionality of a specific MEC host and the applications running on it. It includes the MEC platform manager, which manages the lifecycle of MEC applications, and the virtualization infrastructure manager, whose main task is to handle the allocation, management and release of virtualized resources

Reference points are standardized communication interfaces between network functions or entities. MEC reference points can be divided into three groups:

- Mp: MEC platform functionality reference points
- Mm: Management reference points
- Mx: Reference points connecting to external entities [8]

Chapter 3

OpenAirInterface 5G

In 2009, the research center EURECOM founded the OpenAirInterface (OAI) Software Alliance with the goal of developing software that would offer a standard-compliant implementation of 3GPP mobile network components.

OpenAirInterface5G (OAI5G) refers to OAI's implementation of 5G, which includes the RAN and CN as well as MEP, which is OAI's implementation of the MEC standard. The platform is designed to run on commercial off-the-shelf (COTS) hardware, making it accessible and cost-effective for a wide range of applications.

The framework itself is made up of several Linux packages that can be downloaded from the official website and installed on one or more systems. Additionally, it can be integrated with radio frequency hardware, such as Ettus USRP or other RF platforms customized by EURECOM. [9]

3.1 Deployment of OAI5G network with nr-UE

The first step in the exploration of OAI5G is to set up a standalone 5G network in its simplest form, in a wired configuration without MEC capabilities. The testbed is composed of:

- 2x laptops - running the OAI gNB and UE respectively (version 2.1.0).

Specification	**Details**
Hardware Model	MSI Co., Ltd. PL62 7RC
Memory	16.0 GiB
Processor	Intel® Core™ i7-7700HQ CPU @ 2.80GHz x 8
Graphics	NVIDIA GeForce MX150
Disk Capacity	2.0 TB SSD
OS Name	Ubuntu 22.04.4 LTS
OS Type	64-bit

Table 3.1: HW and SW details of the laptop running the gNB and CN

Specification	Details
Hardware Model	Dell Vostro 15 3578
Memory	8.0 GiB
Processor	Intel® Core™ i7-8550U CPU @ 1.80GHz x 8
Graphics	AMD Radeon 520
Disk Capacity	500 GB HD
OS Name	Ubuntu 22.04.4 LTS
OS Type	64-bit

Table 3.2: HW and SW details of the laptop running the UE

- 2x Ettus USRP B210 + Enclosure Kit - software defined radios for the transmission and reception of messages
- 2x SMA-SMA cable - to create a communication channel
- 2x 30 dB attenuator - placed in order to simulate channel attenuation

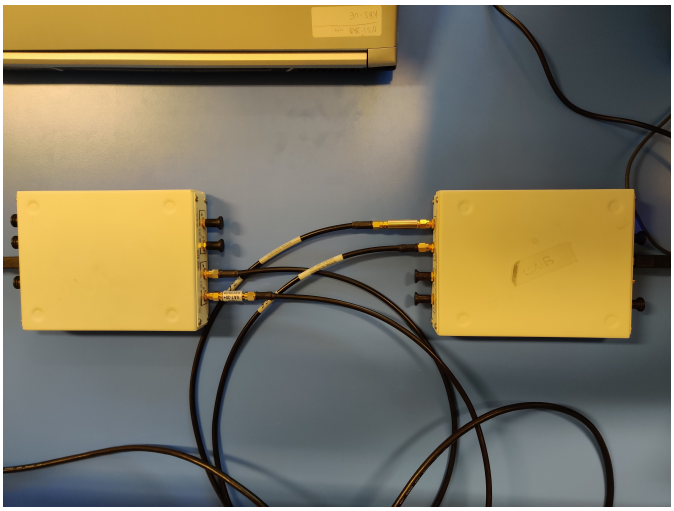


Figure 3.1: Wired connection of USRPs

The deployment of the network was achieved by following tutorial [10]. An important thing to note is that the system requirements to run the OAI gNB and UE are stringent:

- **Laptop/Desktop/Server for OAI CN5G and OAI gNB**
  - **Operating System**: Ubuntu 22.04 LTS
  - **CPU**: 8 cores x86_64 @ 3.5 GHz
  - **RAM**: 32 GB
- **Laptop for UE**
  - **Operating System**: Ubuntu 22.04 LTS
  - **CPU**: 8 cores x86_64 @ 3.5 GHz

– **RAM**: 8 GB

Both the gNB and UE laptops do not respect the system requirements!
Unfortunately, this was the most powerful hardware available for the thesis, so the rest of the research will have to accomodate for the shortage of compute resources.

Initially, file *gnb.sa.band78.fr1.106PRB.usrpb210.conf* was used as configuration for the gNB:

- Standalone version of 5G, exploiting New 5G Packet Core (NGCore)
- Band n78, consisting of frequencies from 3.3 to 3.8 GHz, TDD duplex mode
- Numerology I, which can be employed for frequencies below 6 GHz
- 106 Physical Resource Blocks (PRBs), each corresponding to 12 consecutive subcarriers in the frequency domain

In order to ensure the presence of the 5G network, a GNU-radio-based spectrum analyzer was built according to [11] and was later used to prove that the network was deployed in the correct frequency band. A screenshot is shown in figure 3.2.

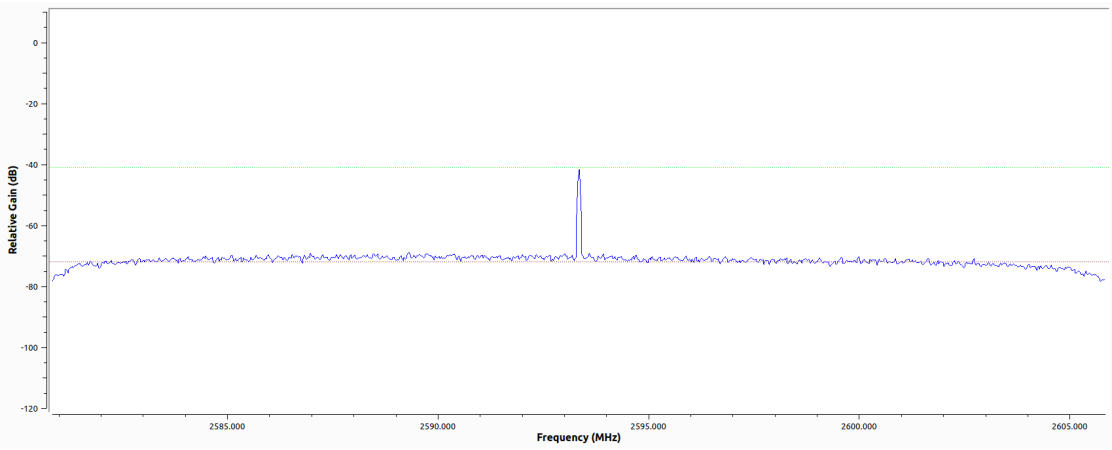


Figure 3.2: GNU radio based spectrum analyzer receiving signal in band n41

After ensuring that the UE could ping the external data network (ext-dn) (fig 3.3), a downlink throughput test was performed for bands n78 and n41 (.conf file was appropriately changed to *gnb.sa.band41.fr1.52PRB.usrpb210.conf*). The latter is again a TDD band, ranging from 2.5 to 2.7 GHz.

An iperf server was launched on the UE (`iperf -s -u -i 1 -B 10.0.0.3`), while Docker container oai-ext-dn was accessed in the CN (`docker exec -it oai-ext-dn /bin/bash`): after a successful `ping` towards the UE, a (downlink) throughput test was started by running the command `iperf -u -t 86400 -i 1 -fk -B 192.168.70.135 -b 100M -c 10.0.0.3`, which sets the maximum bandwidth for the test to 100 Mbps (never reached by the network).

```
phd@phd-dell:~$ ping 192.168.70.135 -I oaitun_ue1
PING 192.168.70.135 (192.168.70.135) from 10.0.0.4 oaitun_ue1: 56(84) bytes of data.
64 bytes from 192.168.70.135: icmp_seq=1 ttl=63 time=38.2 ms
64 bytes from 192.168.70.135: icmp_seq=2 ttl=63 time=36.9 ms
64 bytes from 192.168.70.135: icmp_seq=3 ttl=63 time=35.6 ms
64 bytes from 192.168.70.135: icmp_seq=4 ttl=63 time=39.0 ms
64 bytes from 192.168.70.135: icmp_seq=5 ttl=63 time=38.0 ms
64 bytes from 192.168.70.135: icmp_seq=6 ttl=63 time=36.9 ms
64 bytes from 192.168.70.135: icmp_seq=7 ttl=63 time=35.8 ms
^C
--- 192.168.70.135 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6007ms
rtt min/avg/max/mdev = 35.624/37.215/38.989/1.152 ms
```

Figure 3.3: Ping UE to ext-dn

Figure 3.4: Wireless connection of USRPs

The same tests were ran when the USRPs were connected in a wireless configuration by removing the 30 dB attenuators and substituting the SMA-SMA cables with antennas. The results are summarized in table 3.3.
The available throughput seems to be in the 10s of Mb/s in both cases, however the wired configuration (with 30 dB of signal attenuation) always performs better than the wireless counterpart. In addition, for the wired case, the higher band leads to higher throughput, contrary to the wired configuration, where the chosen frequency band does not affect throughput; this suggests that in this case the conditions of the testing environment might be the bottleneck.

The range of the OAI5G connection performed via USRP B210 seems to be very limited, as even walking a few meters in the laboratory frequently leads to crashes of the UE. The problem might be caused by the limited computing resources of the setup.

3.2 OAI MEP

OAI MEP was recently developed by EURECOM in adherence to the 3GPP standard *ETSI GS MEC 003 V3.1.1* and is tightly integrated with the OAI 5G network. At the time of writing, the OAI MEP remains a work in progress, currently providing only a registry and discovery environment for MEC services, along with its own implementation of the Radio Network

[Mb/s]	Wired	Wireless
n41	17.3	14.7
n78	20.2	14.6

Table 3.3: Average throughput over 20 seconds for different configurations [Mb/s]

Information Service (RNIS).
The currently developed architecture is shown in figure 3.5. Both Mp1 and Mp2 interfaces are implemented: MEC apps communicate with MEP via the Mp1 interface, whereas applications hosted at MEP communicate with the RAN and CN via the Mp2 interface.

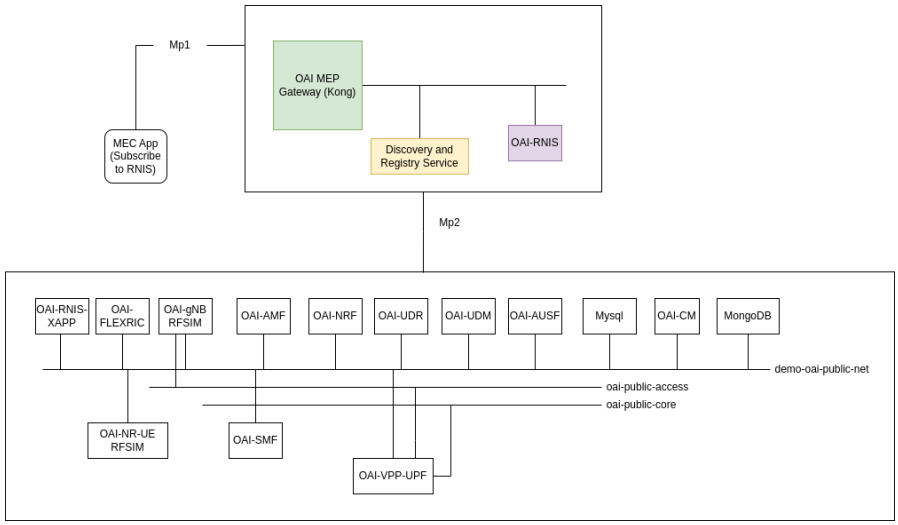


Figure 3.5: MEP architecture

Additionally, OAI also provides its own implementation of the RNIS, which in theory is able to expose radio link information to MEC apps. The service is composed of several modules (figure 3.6):

- **Northbound API**: An OpenAPI Standard interface that exposes the Radio Network Information API, allowing MEC Apps to subscribe to radio data via a RESTful interface
- **Notification Service**: Informs subscribed apps about changes in network data via an HTTP-based notification service
- **Data Convergence Service**: A repository for network information, which processes data and triggers notifications when changes occur
- **Core Network Wrapper Service**: Provides an abstraction layer for the Core Network. It is a consumer for all the events related to the CN exposed by the OAI-CM
- **KPIs-xApp Service**: Interfaces with the RAN xApp's (RNIS xApp) northbound API to collect user KPIs. xApps are specialized software applications that run on the RAN Intelligent Controller (RIC) designed to perform various radio network functions, analytics, and control tasks to optimize the performance of the RAN.

In other words, the Core Network Wrapper handles events from the CN, while the KPIs-xApp collects radio metrics from the O-RAN xApp. The collected data is sent to the Data Convergence Service, which aggregates the information and triggers events; lastly, this data is made accessible via the Northbound API or the Notification Service. [12]

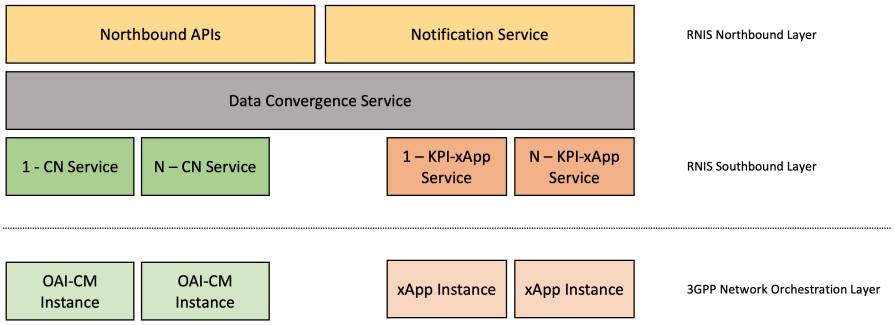


Figure 3.6: RNIS Architecture as of the time of writing

3.2.1 Deployment of OAI-MEP 5G network with RFsim

The goal of this section is to showcase how OAI-MEP can be exploited to run a simple MEC application consuming RAN and CN KPIs exposed via RNIS, all done in a **simulated environment** (rfsim).
Compared to previous deployments of OAI5G, where only the CN made use of Docker containers, this implementation is completely container-based and includes the *configuration manager*, a network function designed to manage OAI CN functions potentially through a GUI.

After following [12], a quick check revealed that all containers are in the right state, as can be observed in figure 3.7. It was then possible to successfully ping the ext-dn container from the UE (figure 3.8) and to run the example MEC app provided by the tutorial, obtaining RAN KPI readings.

3.2.2 Understanding the example MEC app

Listed below is the example MEC app provided by OAI:

content/chapters/3/docs/example–mec–app.py

```
# Copyright    2023 the OAI–RNIS Authors

# Licensed under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at

#   http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.
```

```
lorenzo@lorenzo-X555LD:~/Documents/thesis/OAI/blueprints/mep$ docker ps
CONTAINER ID   IMAGE                                      COMMAND                  CREATED       STATUS                            PORTS
NAMES
83534ca97a78   oaisoftwarealliance/oai-mep:latest         "oai_mep"                2 hours ago   Up 2 hours (healthy)
oai-mep-registry
7b798781bb05   kong:latest                                "/docker-entrypoint.…"   2 hours ago   Up 2 minutes (healthy)            8000/tcp, 8443-8444/
tcp, 0.0.0.0:32793->80/tcp, :::32793->80/tcp, 0.0.0.0:32792->8001/tcp, :::32792->8001/tcp
oai-mep-gateway
97b380cd30ac   postgres:9.6                               "docker-entrypoint.s…"   2 hours ago   Up 2 hours (healthy)              5432/tcp
oai-mep-gateway-db
052a0277a509   oaisoftwarealliance/oai-flexric:1.0        "python3 -u rnisxapp…"   2 hours ago   Up 2 minutes (healthy)            36421-36422/sctp
oai-rnis-xapp
5ee2eb4a35fa   oaisoftwarealliance/oai-flexric:1.0        "/usr/local/bin/near…"   2 hours ago   Restarting (139) 26 seconds ago
oai-flexric
3bdf0b0eab09   rabbitmq:3-management-alpine               "docker-entrypoint.s…"   2 hours ago   Up 2 hours (healthy)              4369/tcp, 5671/tcp,
15671/tcp, 15691-15692/tcp, 25672/tcp, 0.0.0.0:32769->5672/tcp, :::32769->5672/tcp, 0.0.0.0:32768->15672/tcp, :::32768->15672/tcp   rabbitmq-broker
4e9811411e29   oaisoftwarealliance/oai-gnb:mep-compatible  "/opt/oai-gnb/bin/en…"   2 hours ago   Up 2 hours (healthy)
rfsim5g-oai-gnb
c900bbb5d605   oaisoftwarealliance/oai-cm:latest           "oai_cm"                 2 hours ago   Up 2 hours (healthy)
oai-cm
8fe70533e3e9   mongo:latest                               "docker-entrypoint.s…"   2 hours ago   Up 2 hours (healthy)              27017/tcp
mongodb
a4e624d1ca7f   oaisoftwarealliance/oai-smf:v1.5.0         "python3 /openair-sm…"   2 hours ago   Up 2 hours (healthy)              80/tcp, 8080/tcp, 88
05/udp
oai-smf
42bcdb1674d4   oaisoftwarealliance/oai-amf:v1.5.0         "python3 /openair-am…"   2 hours ago   Up 2 hours (healthy)              80/tcp, 9090/tcp, 38
412/sctp
oai-amf
71347932d7d4   oaisoftwarealliance/oai-ausf:v1.5.0        "python3 /openair-au…"   2 hours ago   Up 2 hours (healthy)              80/tcp
oai-ausf
c4d73494addb   oaisoftwarealliance/oai-udm:v1.5.0         "python3 /openair-ud…"   2 hours ago   Up 2 hours (healthy)              80/tcp
oai-udm
eb2eef432b3e   oaisoftwarealliance/oai-udr:v1.5.0         "python3 /openair-ud…"   2 hours ago   Up 2 hours (healthy)              80/tcp
oai-udr
728773d283d1   oaisoftwarealliance/oai-upf-vpp:v1.5.0     "/openair-upf/bin/en…"   2 hours ago   Up 2 hours (healthy)              2152/udp, 8085/udp
vpp-upf
bd8ab1878128   mysql:8.0                                  "docker-entrypoint.s…"   2 hours ago   Up 2 hours (healthy)              3306/tcp, 33060/tcp
mysql
88b74077332a   oaisoftwarealliance/trf-gen-cn5g:latest    "/bin/bash -c ' ipta…"   2 hours ago   Up 2 hours (healthy)
oai-ext-dn
80a4197a437c   oaisoftwarealliance/oai-nrf:v1.5.0         "python3 /openair-nr…"   2 hours ago   Up 2 hours (healthy)              80/tcp, 9090/tcp
oai-nrf
```

Figure 3.7: Running containers after deployment of RNIS

```
root@e8c9a98b9a6b:/opt/oai-nr-ue# ping 192.168.73.135 -I oaitun_ue1
PING 192.168.73.135 (192.168.73.135) from 12.1.1.2 oaitun_ue1: 56(84) bytes of data.
64 bytes from 192.168.73.135: icmp_seq=1 ttl=63 time=95.0 ms
64 bytes from 192.168.73.135: icmp_seq=2 ttl=63 time=14.0 ms
64 bytes from 192.168.73.135: icmp_seq=3 ttl=63 time=16.4 ms
64 bytes from 192.168.73.135: icmp_seq=4 ttl=63 time=19.3 ms
64 bytes from 192.168.73.135: icmp_seq=5 ttl=63 time=21.4 ms
^C
--- 192.168.73.135 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 14.061/33.255/95.054/31.001 ms
root@e8c9a98b9a6b:/opt/oai-nr-ue#
```

Figure 3.8: Successful ping from UE to external data network (ext-dn)

```python
# Contact: netsoft@eurecom.fr

import flask
from flask import request, jsonify
import socket
import json
import requests

collected_data = {}

hostname=socket.gethostname()
ip_addr= "192.168.70.1" #socket.gethostbyname(hostname)
port = 30020

print(f"Dashboard URL:{ip_addr}:{port}/dashboard")

### create a subscription for receiving l2 measurements
sub_endpoint = "http://oai-mep.org/rnis/v2/subscriptions"
sub_body ={
   "callbackReference": f"http://{ip_addr}:{port}/subscriptions/l2meas-200",
   "filterCriteriaNrMrs": {},
   "subscriptionType": "NrMeasRepUeSubscription",
   "expiryDeadline": {
```

```
38      "nanoSeconds": 12133423,
39      "seconds": 123124234
40    }
41  }
42  requests.post(url =sub_endpoint, json=sub_body)
43
44  print("Subscribed to RNIS")
45
46  app = flask.Flask(__name__)
47
48
49  @app.route('/subscriptions/l2meas-200', methods=[ 'POST'])
50  def receive_notification():
51      if request.method == 'POST':
52          content = request.get_json(force=True)
53          print(content)
54          kpis = content["Report"]
55          for aid in content["AssociateId"]:
56              if aid not in collected_data:
57                  collected_data[aid] = {}
58              for kpi in kpis:
59                  collected_data[aid]["kpi"] = kpi
60      return "OK"
61
62  @app.route('/dashboard', methods=['GET'])
63  def dashboard():
64      return "<h1> Network Monitoring Dashboard</h1> " + json.dumps(collected_data)
65
66  app.config["DEBUG"] = False
67  app.run(ip_addr, port=port)
```

Let's break it down:

- **Lines 23-29**: host IP address and port are established and the dashboard URL is printed to console (the dashboard itself is defined in lines 62-64)
- **Lines 31-44**: creates a RNIS subscription so that the MEC app is able to receive L2 Measurements. `CallbackReference` specifies the URL where notifications should be sent: it is set to *http://ip_addr:port/subscriptions/l2meas-200* (the endpoint is defined in lines 49-60).

  The line `"SubscriptionType":"NrMeasRepUeSubscription"` means that the RNIS subscription is related to measurement reports (MeasRep) of the UE in the 5G-nr network. The measurements are not filtered and the deadline of the subscription is specified with the `seconds` and `nanoseconds` fields
- **Lines 46-60**: the Flask application is initialized and an endpoint is defined in order to receive the L2 measurements. If a `POST` request is received (meaning that RNIS has sent a measurement report to the MEC app), the method `receive_notification()` is called: the JSON data of the measurements is first printed to console and then parsed according to the different UEs (`aid` - supposedly standing for *AssociateId*) and the KPIs associated to the UE. The data is saved in a matrix structure in the variable `collected_data`
- **Lines 62-64**: defines the `dashboard` endpoint to display a network monitoring dashboard upon a receiving a `GET` request. Upon receiving such request, the method `dashboard()` is called and a simple HTML webpage is returned to the client making the GET request. In this way, users are able to visualize the network report data through their web browser

- **Lines 66-67**: the app is configured and finally ran on the specified IP address and port

Subsequently, some simple modifications were applied to the example MEC app in order to write incoming KPIs from RNIS onto a log file; in particular, method `receive_notification()` was modified as follows:

```
from datetime import datetime as dt
currenttime = dt.now().isoformat()

[...]

@app.route('/subscriptions/l2meas-200', methods=[ 'POST'])
def receive_notification():
    if request.method == 'POST':
        content = request.get_json(force=True)
        print(content)
        kpis = content["Report"]
        for aid in content["AssociateId"]:
            if aid not in collected_data:
                collected_data[aid] = {}
            for kpi in kpis:
                collected_data[aid]["kpi"] = kpi
        #log collected data
        with open("log_"+ currenttime + ".json", "a") as json_file:
            json.dump(content,json_file)
            json_file.write('\n')
    return "OK"
```

3.2.3 Deployment of OAI-MEP 5G network with nr-UE

The next step is to create the network with real equipment (USRPs) instead of a simulated environment. The hardware setup is the same as the one presented in section 3.1, with the gNB laptop also running the OAI MEP framework.

The procedure to achieve the deployment is not presented in the OAI MEP tutorial [12], however some clues on how to do this are found in the *docker-compose-ran.yaml* files used in the deployment of the RAN containers, including the gNB and UE. The main modifications operated to such files are:

- flag `USE_B2XX` was set to 'yes', in order to allow execution on USRP B210 instead of simulated environment
- The serial of the USRP was obtained through `sudo uhd_usrp_probe` and substituted in entry `SDR_ADDRS` of the .yaml file
- USB access was granted to the container by mounting the volume `- /dev/bus/usb/:/dev/bus/usb/`. An **important remark** is that it seems that the containers are able to access the USB port of the host computer only if docker is used from command line while **Docker Desktop is closed**. Otherwise, the containers are unable to recognize the USRP.
- flag `USE_VOLUMED_CONF` was set to 'yes' for the gNB, so that the container may use an arbitrary .conf file passed from the host. For the UE, the flag was set to 'no', as the configuration can be passed directly through the fields already present in the .yaml file (*IMSI, KEY, OPC...*)

- The gNB .conf file was moved to the *blueprints/mep/docker-compose* folder and mounted as volume in the gNB/UE docker container (`- ./gnb-n78.conf:/opt/oai-gnb/etc/mounted.conf`). In particular, the gNB used the file *gnb.sa.band78.fr1.106PRB.usrpb210.conf* (appropriately renamed to *gnb-n78.conf*)

The finalized `docker-compose-ran.yaml` file entries for the gNB and UE are respectively listed below:

content/chapters/3/docs/docker–compose–ran–USRP.yaml

```
    oai-gnb:
        image: oaisoftwarealliance/oai-gnb:mep-compatible
        privileged: true
        container_name: oai-gnb-usrp
        environment:
            RFSIMULATOR: server
            USE_SA_TDD_MONO: 'yes'
            GNB_NAME: gnb
            USE_B2XX: 'yes'  #only needed when using B210
            USE_VOLUMED_CONF: 'yes' #only needed when mounting the configuration
    file
            TAC: 1
            MCC: '208'
            MNC: '99'
            MNC_LENGTH: 2
            NSSAI_SST: 1
            AMF_IP_ADDRESS: 192.168.70.132
            GNB_NGA_IF_NAME: demo-oai
            GNB_NGA_IP_ADDRESS: 192.168.70.160
            GNB_NGU_IF_NAME: cn5g-access
            GNB_NGU_IP_ADDRESS: 192.168.72.160
            SDR_ADDRS: serial=314BCFF #substituted serial of USRP
            USE_ADDITIONAL_OPTIONS: --sa -E --continuous-tx --log_config.
    global_log_options level,nocolor,time,line_num,function #(B210)
        volumes:
            - shared_lib:/usr/local/lib/flexric/
            - ./conf/flexric.conf:/usr/local/etc/flexric/flexric.conf
            -  /dev/bus/usb/:/dev/bus/usb/ #(B210)
            - ./gnb-n78.conf:/opt/oai-gnb/etc/mounted.conf #(B210) config file
    should be present
        networks:
            public_net:
                ipv4_address: 192.168.70.160
            public_net_access:
                ipv4_address: 192.168.72.160
        healthcheck:
            test: /bin/bash -c "pgrep nr-softmodem"
            interval: 10s
            timeout: 5s
            retries: 5
```

content/chapters/3/docs/docker–compose–ran–realUE.yaml

```
    oai-nr-ue:
        image: oaisoftwarealliance/oai-nr-ue:2023.w06
        privileged: true
        container_name: oai-nr-ue
        environment:
            USE_B2XX: 'yes'  #only needed when using B210
            USE_VOLUMED_CONF: 'no' #only needed when mounting the configuration
    file
```

```
            SDR_ADDRS: serial=314BC7D
            RFSIMULATOR: 192.168.70.160
            FULL_IMSI: '208990100001120'
            FULL_KEY: 'fec86ba6eb707ed08905757b1bb44b8f'
            OPC: 'C42449363BBAD02B66D16BC975D77CC1'
            DNN: oai
            NSSAI_SST: 1
            USE_ADDITIONAL_OPTIONS: -E --sa -r 106 --band 78 --numerology 1 -C
    3319680000 --uicc0.imsi 208990100001120 --ue-fo-compensation --log_config.
    global_log_options level,nocolor,time
        volumes:
            - shared_lib:/usr/local/lib/flexric/
            - ./conf/flexric.conf:/usr/local/etc/flexric/flexric.conf
            -  /dev/bus/usb/:/dev/bus/usb/ #(B210)
            - ./nrue.uicc.conf:/opt/oai-nr-ue/etc/mounted.conf #(B210) config file
     should be present
        networks:
            public_net:
                ipv4_address: 192.168.70.161
        healthcheck:
            test: /bin/bash -c "pgrep nr-uesoftmodem"
            interval: 10s
            timeout: 5s
            retries: 5
```

Unfortunately, a fully working OAI5G-MEP deployment with nrUE was never achieved. The nrUE would partially connect to the gNB and RAN KPIs would be correctly measured, but the UE and gNB could not ping each other as in the RFSim deployment.

For this reason, the remainder of the thesis will make use of the MEP-less 5G deployment achieved in section 3.1.

3.2.4 Evaluation of MEP Testbed: Radio Link Measurements

Even though the nrUE and gNB could not ping each other, a simple experiment was conducted: radio link KPI measurements were taken while progressively increasing the distance between the USRPs until the two disconnected.
The listing below reports an example of one measurement of one of the KPIs:

```
{
  "AssociateId": [
    "12.1.1.7"
  ],
  "CellId": 0,
  "Report": {
    "rsrp": {
      "kpi": "rsrp",
      "source": "RAN",
      "timestamp": 1712674276022358,
      "unit": "dBm",
      "value": -86,
      "labels": {
```

```
            "amf_ue_ngap_id": 6
        }
        [...]
    },
    "TimeStamp": 1712674276.0269651
}
```

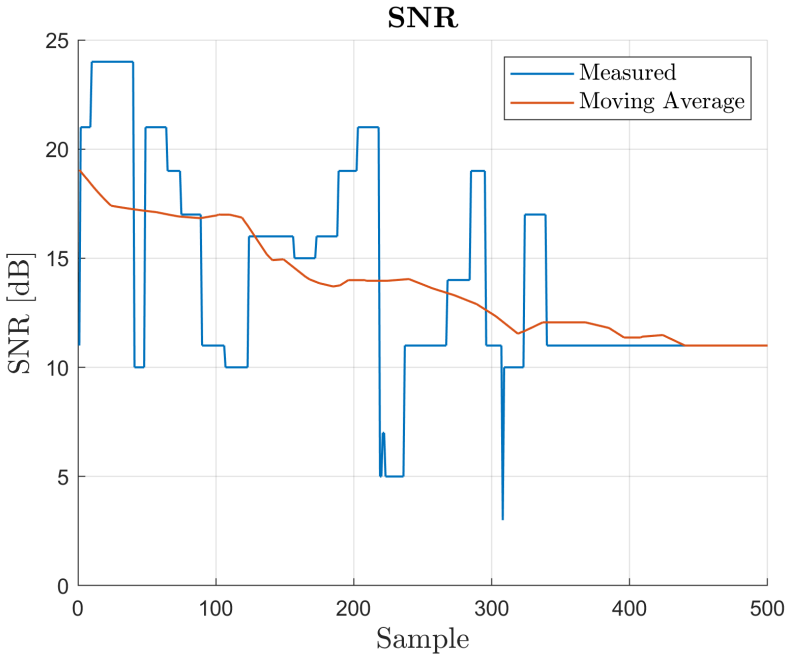


Figure 3.9: SNR measurements

Figure 3.9 shows the obtained measurements for the signal-to-noise ratio: even though there is a high volatility in the value, it can be observed that on average the SNR lowers when the user moves further away from the gNB, which is expected. After around 350 samples were recorded, the measurement stopped fluctuating and became constant independently of how much further away the UE travelled, meaning that signal was lost.
Figure 3.10 shows a measurement of the Reference Signal Received Power (RSRP), which refers to the received power of the 5G Reference Signals (in dBm) withing the frequency bandwidth. Following information given in table 3.11, the RSRP starts out already in Mid Cell conditions, plummets at around sample 300 to around $-140dBm$ and subsequently recovers at around $-115dBm$ (connection was lost in this state).
Figure 3.12 shows the Block Error Rate, which ideally is kept at around 10 % thanks to the Channel Quality Indicator (CQI) parameter: the UE sends feedback to the gNB about the quality of the perceived channel (measured using the reference signals), suggesting a transport format. Unfortunately, both in simulation and with the testbed, CQI is always reported as 0: this would suggest that the UE is out of range, but this is clearly not the case given the successful real-time measurement of other radio parameters.
Additionally, it was observed that the Modulation and Coding Scheme (MCS) in DL changes from 9 to 8 as the UE is moved further away, meaning that a more robust scheme was adopted in response to the changes in the channel. The MCS in UL remains constantly at 6.

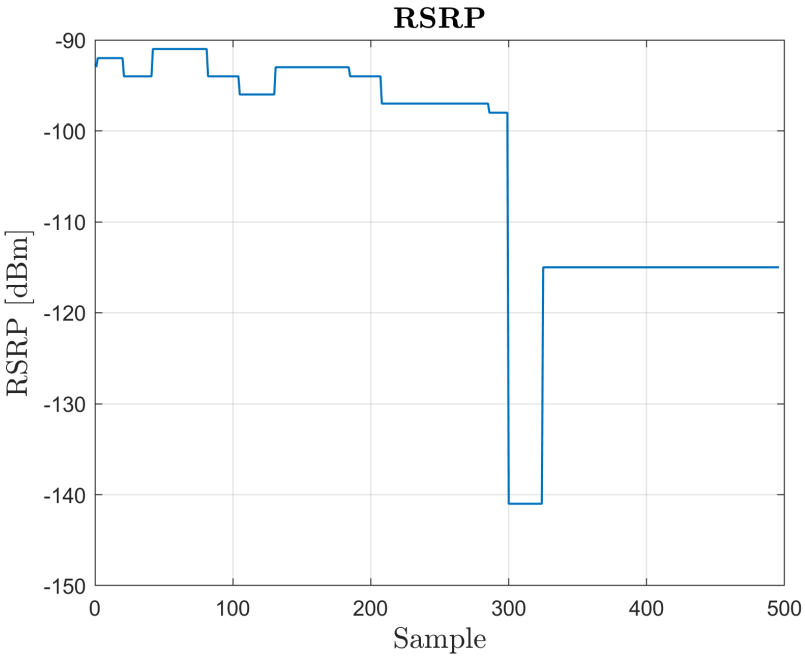


Figure 3.10: RSRP measurements

		RSRP (dBm)	RSRQ (dB)	SINR (dB)
RF Conditions	Excellent	>=-80	>=-10	>=20
	Good	-80 to -90	-10 to -15	13 to 20
	Mid Cell	-90 to -100	-15 to -20	0 to 13
	Cell Edge	<=-100	<-20	<=0

Figure 3.11: [13]

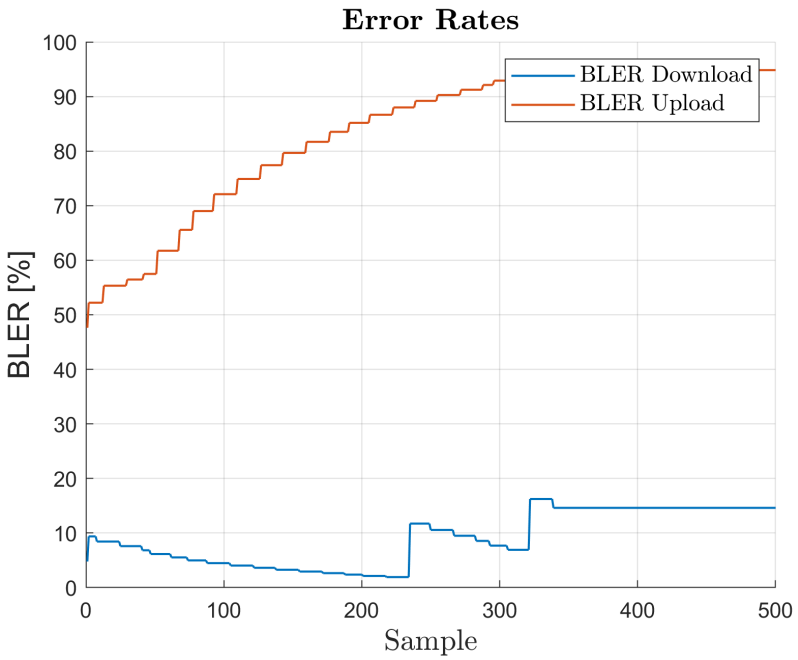


Figure 3.12: BLER measurements in DL and UL

Chapter 4

PX4 Autopilot

PX4 is an *open-source* flight control software designed for drones and other unmanned vehicles. It offers a versatile toolkit for drone developers, enabling them to collaborate and share technologies to develop customized solutions for various drone applications. [14]

PX4 boasts several features [15]:

- numerous vehicle types including multicopters, fixed-wing aircraft (planes), helicopters, airships, rovers, boats, submersibles
- high modularity and compatibility with different sensors, flight-controllers and payloads
- powerful flight modes included with safety features
- deep integration with robotic APIs, such as ROS2 and MAVSDK

4.1 System Architecture

PX4 is composed of two primary layers.
The first layer, the *flight stack*, functions as an estimation and flight control system. The second layer, the *middleware*, acts as a general robotics layer that supports various types of autonomous robots, facilitating internal and external communications as well as hardware integration. [16]

Figure 4.1 presents a typical high-level configuration for a PX4 system, comprising of a few hardware and software components [17]:

- **Flight controller**: The central on-board hardware device responsible for controlling the drone's flight by processing sensor data and sending commands to the motors, according to the PX4 flight stack.
- **Companion (or mission) computer**: It works on-board alongside the flight controller to handle advanced tasks such as image processing, navigation, and autonomous decision-making, providing advanced features such as object avoidance and collision prevention. It usually runs on Linux.
- **Ground station**: The computer used by the operator to monitor and control the drone. It typically runs a ground station software (such as QGroundControl) and eventually some robotics software such as ROS/ROS2/MAVSDK.

- **Motors and sensors**
- **Payload**: Any additional equipment carried by the drone for specific missions, such as cameras, delivery packages, or scientific instruments.
- **Application(s)**: Software that defines the specific tasks and behaviors the drone will perform
- **Middleware**: As already explained, software layer that facilitates communication between the flight controller, companion computer, and other subsystems, ensuring seamless integration and operation.
- **Telemetry Radio / LTE**: represents the means of communication between the ground station and the mission computer. Citing from [16]:
  > PX4 does not deliver software specifically for LTE and/or cloud integration (this requires custom development).

  Since the thesis involves the usage of 5G instead of LTE, it can be deduced that some custom setup will be needed
- **Drivers**: low-level software for the actuators and sensors

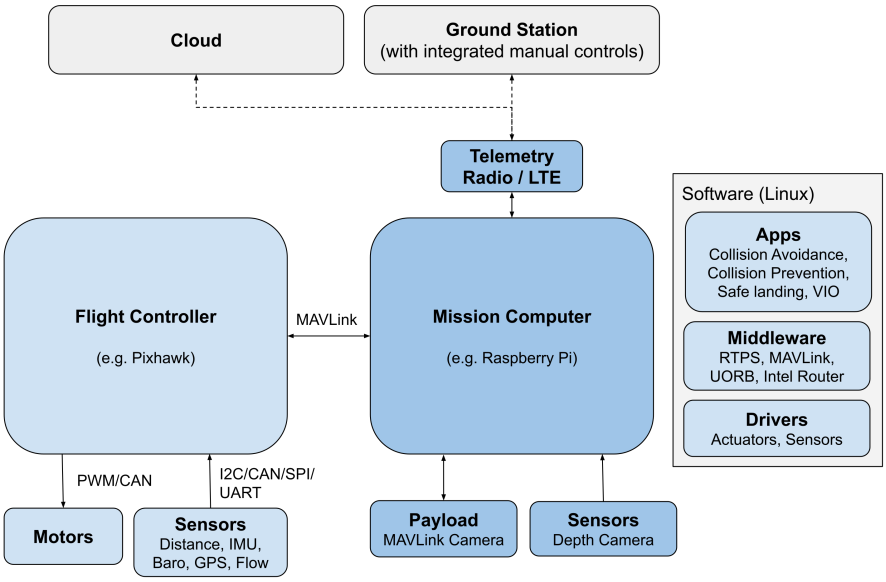


Figure 4.1: PX4 flight control system with companion computer

4.2 Software Architecture

The system design adheres to the *reactive manifesto* [18], striving to be:

1. **Responsive**: A responsive system ensures timely responses, which are crucial for usability and utility. Responsiveness helps in quickly identifying and addressing issues, delivering consistent response times and maintaining reliable service quality. Such reliability

and consistency simplifies error handling, enhances user confidence and promotes further interaction.

2. **Resilient**: Resilience ensures that a system remains responsive even when failures occur. Achieving resilience involves replication, containment, isolation, and delegation. Failures are confined to individual components, preventing them from affecting the entire system, while recovery is handled by external components

3. **Elastic**: An elastic system maintains responsiveness under fluctuating workloads. It adapts to changes in input rates by adjusting resource allocation. Reactive systems support scaling by monitoring performance metrics, achieving cost-effective elasticity on standard hardware and software.

4. **Message-Driven**: Reactive Systems utilize *asynchronous message-passing* to create boundaries between components, ensuring loose coupling, isolation and location transparency.

In PX4, this translates in having a modular codebase, using asynchronous messages for communication and making the system resilient to varying workloads. In this regard, the modules exploit a publish-subscribe message bus named **uORB** for communication.

4.2.1 Flight Stack

> The flight stack is a collection of guidance, navigation and control algorithms for autonomous drones. It includes controllers for fixed-wing, multirotor and VTOL airframes as well as estimators for attitude and position. [16]

The diagram in figure 4.2 shows an overview of the flight stack.

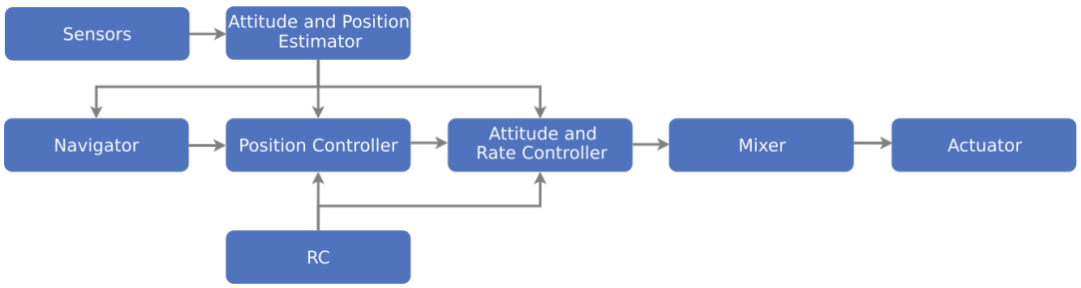


Figure 4.2: Overview of the PX4 flight stack

A central component is the **controller**, which takes as input a setpoint and the estimated state of the drone and outputs a correction in order to reach the input setpoint. There are controllers for position, attitude and rate.
The **mixer** finally translates the corrections into individual commands for the motors.
In this context, the estimated state of the vehicle (to be fed to the controllers) is provided by **estimators**, which make use of one or multiple sensor measurements. On the other hand, the setpoints for the controller are sent either by the on-board navigator or via an **RC link**.
For what concerns the thesis, it is important to note that the setpoints for the controllers need to be sent from the ground station via the **5G link** to the on-board flight controller module.

4.2.2 Middleware

In general, a **middleware** acts as a bridge between different software components or systems, enabling them to communicate and share data seamlessly. It abstracts the complexities of underlying communication protocols and provides a standardized way for applications to interact.

The middleware of PX4 mainly comprises device drivers for embedded sensors, mechanisms for communication with external entities such as companion computers and ground stations, and the uORB publish-subscribe message bus.
Additionally, it features a **simulation layer** that enables PX4 flight code to operate on a desktop operating system, allowing it to control a computer-modeled vehicle within a simulated environment.

uORB Messaging

> The uORB (micro Object Request Broker) is an asynchronous publish() / subscribe() messaging API used for inter-thread/inter-process communication. [19]

Each of the available uORB topics corresponds to a different type of message that may be sent. In total, there are more than 100 topic definitions in the PX4-Autopilot repository, and new ones may be defined by the user if necessary.
An example of a topic is `sensor_combined`, which carries the synchronized sensor readings of the complete system. Here is the message definition for such topic:

```
# Sensor readings in SI-unit form.
# These fields are scaled and offset-compensated where possible and do not
# change with board revisions and sensor updates.

uint64 timestamp                                # time since system start (microseconds)

int32 RELATIVE_TIMESTAMP_INVALID = 2147483647   # (0x7fffffff) If one of the relative timestamps is set to this value, it means the associated sensor values are invalid

# gyro timstamp is equal to the timestamp of the message
float32[3] gyro_rad                             # average angular rate measured in the FRD body frame XYZ-axis in rad/s over the last gyro sampling period
uint32 gyro_integral_dt                         # gyro measurement sampling period in microseconds

int32 accelerometer_timestamp_relative  # timestamp + accelerometer_timestamp_relative = Accelerometer timestamp
float32[3] accelerometer_m_s2           # average value acceleration measured in the FRD body frame XYZ-axis in m/s^2 over the last accelerometer sampling period
uint32 accelerometer_integral_dt        # accelerometer measurement sampling period in microseconds

uint8 CLIPPING_X = 1
uint8 CLIPPING_Y = 2
uint8 CLIPPING_Z = 4

```

```
21 uint8 accelerometer_clipping    # bitfield indicating if there was any
       accelerometer clipping (per axis) during the integration time frame
22 uint8 gyro_clipping             # bitfield indicating if there was any gyro
       clipping (per axis) during the integration time frame
23
24 uint8 accel_calibration_count   # Calibration changed counter. Monotonically
       increases whenever accelermeter calibration changes.
25 uint8 gyro_calibration_count    # Calibration changed counter. Monotonically
       increases whenever rate gyro calibration changes.
```

The different variables declared carry information from the gyroscope and the accelerometer; sometimes the reading also includes a relative timestamp, which is taken with respect to the timestamp declared at the start of the message.

4.2.3 Update Rates

Different components in the PX4 Autopilot system operate at various update rates, depending on their requirements. Modules within the system rely on data updates from sensors or other sources and their update speeds are often determined by the drivers managing the sensors.
For instance, Inertial Measurement Unit (IMU) drivers sample raw data from sensors at a high rate, typically 1 kHz, and after processing (such as integrating the data) they publish it at 250 Hz. This high-frequency update is necessary for accurate motion tracking. On the other hand, certain system components, like the navigator, don't require such frequent updates because their tasks, such as route calculation or position estimation, don't need rapid changes. As a result, they run at a considerably slower update rate to optimize system performance and resource usage.

4.2.4 Runtime Environment

PX4 operates on multiple operating systems that support a POSIX-API, including Linux, macOS, NuttX, and QuRT. It also requires real-time scheduling capabilities.
Inter-module communication utilizes a shared memory approach through uORB. The entire PX4 middleware functions within a single address space, meaning all modules can access the same portion of memory.
On Linux or macOS, PX4 operates within a single process, with each module running in its own thread. [16]

4.3 Simulation Environments

Simulators enable PX4 flight code to operate a computer-generated vehicle within a virtual environment. You can control this simulated vehicle in the same way as you would a real one, using tools like QGroundControl, an offboard API or a radio controller/gamepad.
Simulating flight is a fast and safe method for testing changes to PX4 code before conducting real-world flights. It's also useful if you do not have a physical vehicle to work with yet, which unfortunately is the case of the thesis.

PX4 offers both Software In the Loop (SITL) and Hardware In the Loop (HITL) simulations. In SITL, the flight software runs on your computer, either locally or on another networked machine. HITL simulation uses a real flight controller board with specialized simulation firmware. [20]

Two examples of available simulators are:

1. **Gazebo**: it is actively supported by the PX4 development team, it offers enhanced rendering, physics and sensor modeling capabilities compared to its predecessor Gazebo Classic.

   As a robust 3D simulation platform, it is ideal for testing scenarios like object avoidance and computer vision applications. Gazebo is also well-suited for multi-vehicle simulations and integrates well with ROS.

   The vehicles supported in Gazebo include quadcopters, standard VTOLs and planes

2. **jMAVSim**: it was previously part of the PX4 development toolchain but was removed in favour of Gazebo. It is a lightweight, fast simulator specifically designed for simple drone simulations. It is easier to set up and consumes fewer resources than Gazebo, making it suitable for testing basic flight control, position estimation, and autopilot tuning

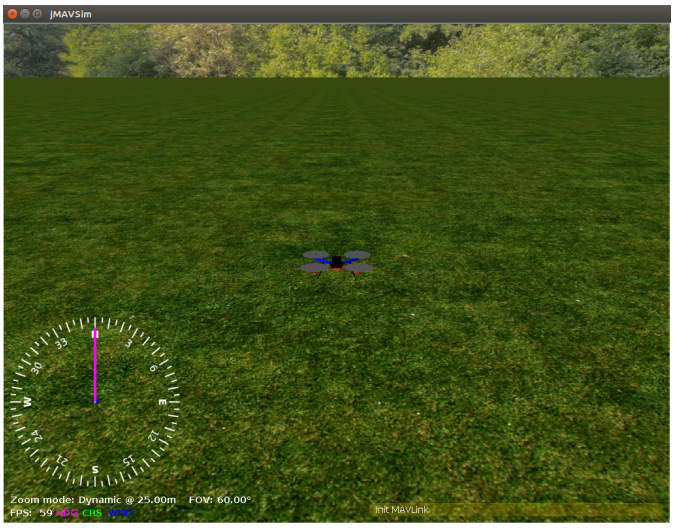


Figure 4.3: Screenshot of a running instance of jMAVSim

4.4 Flight Modes

Flight modes in multicopters are designed to simplify manual flying, automate common actions such as takeoff and landing, enable autonomous missions, or hand over control to external systems. This section provides an overview of flight modes specific to multicopters (and helicopters), as this corresponds to the type of rotorcraft typically used for UAVs.

Flight modes fall into two main categories: manual and autonomous. Manual modes offer varying degrees of autopilot support when controlling the vehicle via RC sticks or a joystick. Autonomous modes, on the other hand, allow the autopilot to fully manage the vehicle's movements and are most in the interest of the thesis.
Among the autonomous flight modes, Offboard Control is the one adopted for the thesis and will be discussed in section 4.4.1.
Pilots can switch between these modes using remote control switches or through a ground control station. Some flight modes require specific pre-flight conditions, such as GPS lock or sensor

availability. PX4 ensures that these conditions are met before allowing a transition to certain flight modes. [21]

4.4.1 Offboard Control Mode

Offboard mode allows external systems, such as companion computers, to control a vehicle by providing setpoints for position, velocity, acceleration, attitude, attitude rates, or thrust/torque. These setpoints can be communicated via ROS2 (adopted for the thesis) or MAVLink.

PX4 requires a continuous "proof of life" signal from the external controller at a minimum rate of 2 Hz, using MAVLink setpoint messages or ROS 2 *OffboardControlMode* messages. PX4 will enable offboard control only after receiving this signal consistently for over one second. If the signal drops below 2 Hz, PX4 will exit offboard mode and initiate a failsafe action, which depends on the RC control availability and settings defined by the parameter `COM_OBL_RC_ACT`.
For ROS2, the external controller proves its presence via *OffboardControlMode* messages, while setpoints are published to topics like *TrajectorySetpoint*.

In addition to providing heartbeat functionality, *OffboardControlMode* plays two crucial roles. Firstly, it determines the appropriate level within the PX4 control architecture where offboard setpoints should be integrated and disables any bypassed controllers. Secondly, it specifies the required estimates, such as position or velocity, and identifies which setpoint messages are valid for the situation.
Taking a glance at the message definition for *OffBoardControlMode*:

```
# Off-board control mode

uint64 timestamp        # time since system start (microseconds)

bool position
bool velocity
bool acceleration
bool attitude
bool body_rate
bool thrust_and_torque
bool direct_actuator
```

The fields are ordered such that position has the highest precedence, followed by velocity, acceleration, and other subsequent fields. The first field with a non-zero value determines the required valid estimate and the appropriate setpoint message for using offboard mode.
For instance, if the acceleration field is the first to have a non-zero value, PX4 will require a valid velocity estimate from sensor measurements and the setpoint must be provided using the *TrajectorySetpoint* message. Examples of other setpoint messages include *VehicleAttitudeSetpoint*, *VehicleRatesSetpoint* and *VehicleTorqueSetpoint*.

Zooming in on the *TrajectorySetpoint* message, it supports the following input combinations:

1. Position Setpoint: When the position is specified (i.e., not NaN), non-NaN values for velocity and acceleration are used as feedforward terms for the inner loop controllers

2. Velocity Setpoint: If velocity is specified (i.e., not NaN) and position is set to NaN, non-NaN values for acceleration serve as feedforward terms for the inner loop controllers

3. Acceleration Setpoint: When acceleration is provided (i.e., not NaN) and both position and velocity are set to NaN

All values are expressed in the NED (North, East, Down) coordinate system, with units of meters (m) for position, meters per second (m/s) for velocity, and meters per second squared (m/s²) for acceleration. [22]

4.5 PX4-ROS2 Integration

ROS2 is a versatile robotics library that can be used with the PX4 Autopilot, enabling advanced drone applications. ROS2 has a large and active developer community, providing solutions to common robotics challenges, and allows easy access to Linux-based software libraries, such as those used for computer vision.
A key advantage of ROS2 is its deep integration with PX4, enabling the creation of custom flight modes that function seamlessly alongside internal PX4 modes. It also allows high-rate, low-latency reading and writing to uORB topics, making it ideal for control and communication tasks.

The connection between ROS2 and PX4 is facilitated by middleware using the XRCE-DDS protocol. This middleware converts PX4's uORB messages into ROS2 messages, allowing direct interaction between the two systems. The same uORB message definitions are used in ROS2 applications to serialize transmitted messages and deserialize received messages for seamless communication. [23]

4.5.1 Architecture

The ROS2 application pipeline for PX4 is simplified thanks to the use of uXRCE-DDS (e**X**tremely **R**esource **C**onstrained **E**nvironments - **D**ata **D**istribution **S**ervice), a middleware layer that manages communication between PX4 and ROS2. In fact, PX4 uses uORB for messaging to convey commands and communicate information, whereas ROS2's default middleware for communication is DDS (Data Distribution Service).

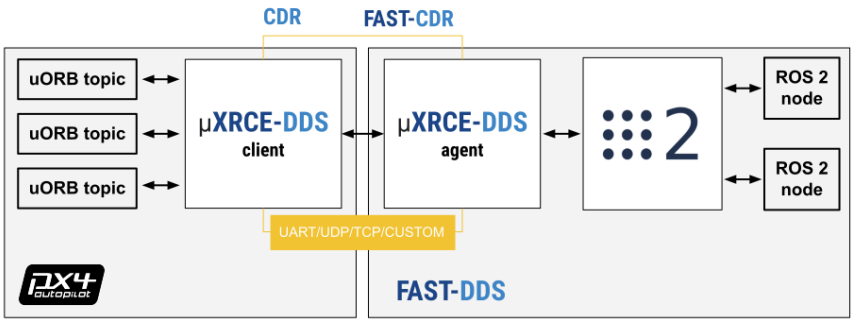


Figure 4.4: Architecture of the ROS2 integration with PX4

Figure 4.4 shows the architecture for the integration of ROS2 over PX4.

The uXRCE-DDS middleware operates through a client on PX4 and an agent on a companion computer, the two communicate via serial, UDP or TCP links. The agent serves as an intermediary, allowing the PX4 client to publish and subscribe to global DDS data.
The uxrce_dds_client is automatically generated and embedded in the PX4 firmware at build time. It leverages uORB message definitions found in PX4-Autopilot/msg to generate the necessary code for transmitting messages in ROS2.

When building ROS2 applications, it is essential to use the same message definitions that were employed to create the uXRCE-DDS client in PX4. These definitions can be obtained by cloning the px4_msgs interface package into the ROS2 workspace, with each branch corresponding to the appropriate PX4 release.
The micro XRCE-DDS agent, which runs on the companion computer, is independent of the client-side code. It can be compiled as part of a ROS build or installed separately. While the PX4 firmware includes the uXRCE-DDS client by default, it must be manually started, except in simulation environments where it starts automatically.

To control a vehicle using ROS2 applications, these applications subscribe to telemetry topics from PX4 and publish to topics that trigger actions. For example, the *VehicleGlobalPosition* topic provides the vehicle's global position, while *VehicleCommand* allows issuing commands like takeoff or landing.
When working with ROS2, it's crucial to ensure the correct Quality of Service (QoS) settings for subscribers. ROS2's default QoS settings are incompatible with PX4, so using the predefined QoS profile for sensor data is necessary. However, when publishing from ROS2 to PX4, no additional settings are required since PX4's defaults align with ROS2's.

There are also differences in frame conventions between ROS and PX4. ROS uses the FLU (Forward, Left, Up) frame, while PX4 follows the FRD (Forward, Right, Down) or NED (North, East, Down) convention. Converting between these frames requires specific rotations. Similarly, vectors like those in *TrajectorySetpoint* and *VehicleThrustSetpoint* messages must be transformed before being sent to PX4. To simplify these conversions, the *PX4/px4_ros_com* library includes shared utilities, such as *frame_transforms*, to handle frame transformations between ROS2 and PX4. [24]

4.5.2 ROS2 Middleware (RMW) - DDS

ROS2 utilizes Data Distribution Service (DDS) as its default middleware, in particular fast-DDS.
DDS is an end-to-end middleware that facilitates communication between distributed systems. It operates on a **publish-subscribe model**: the latter is a messaging pattern which involves publishers organizing messages into categories that subscribers can receive based on their interests. This differs from traditional messaging patterns, where publishers send messages directly to specific subscribers. [25] DDS offers a robust set of features, including a request-response transport and a *distributed discovery system* between DDS programs, making it highly fault-tolerant and flexible.

An important fact is that DDS operates on **UDP by default**. This independence requires DDS to handle the **reliability** of the transport. DDS offers extensive configurability through **Quality of Service (QoS) parameters**, enabling fine-tuning of communication characteristics. For instance, users can optimize for low latency by configuring DDS to prioritize speed over reliability, akin to a UDP protocol. Conversely, in scenarios requiring TCP-like behavior with

enhanced tolerance for prolonged connection losses, DDS allows adjustments to QoS parameters to accommodate these needs effectively.

Concerning ROS2, the decision to adopt DDS over building a new middleware from scratch was influenced by several factors: DDS's comprehensive documentation, recommended use cases, and standardized API simplify development and maintenance, reducing the amount of code to manage. Moreover, DDS's extensive track record in critical applications, coupled with its flexibility and reliability, aligns well with the requirements of ROS2, ensuring efficient and robust communication across robotic systems. [26]

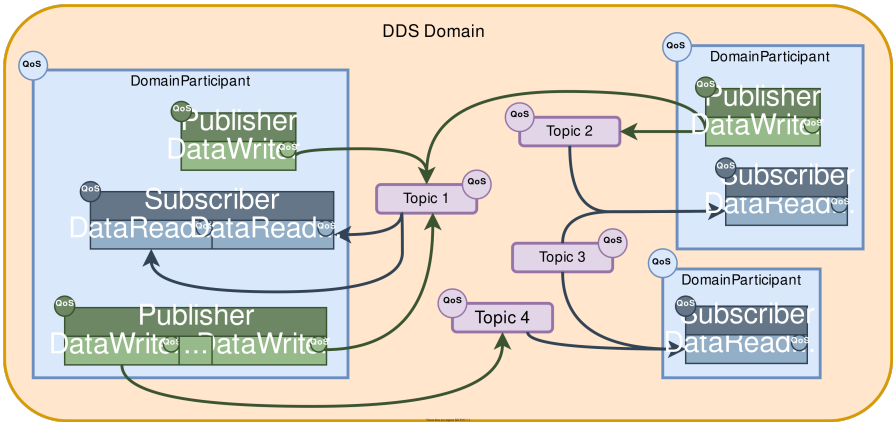


Figure 4.5: Main DDS entities and their interaction [27]

The main entities in DDS are displayed in figure 4.5:

- **DomainParticipant**: This is the entry point for the DDS domain. It represents the participation of an application in a DDS domain and is responsible for creating and managing other DDS entities.
- **Publisher**: A Publisher is responsible for managing DataWriters. It acts as a factory for DataWriters and controls the flow of data from the application to the DDS domain.
- **DataWriter**: This entity is used by the application to publish data. It writes data samples to a specific Topic, which are then distributed to interested subscribers.
- **Subscriber**: A Subscriber manages DataReaders. It acts as a factory for DataReaders and controls the flow of data from the DDS domain to the application.
- **DataReader**: This entity is used by the application to receive data. It reads data samples from a specific Topic, which are published by DataWriters.
- **Topic**: A Topic defines the type of data being published and subscribed to. It includes a name and a data type, which describes the structure of the data.
- **QoS (Quality of Service) Policies**: These policies control various aspects of data distribution, such as reliability, durability, and latency. They ensure that data is delivered according to the specified requirements.
- **Domain**: A logical partition within which DDS entities operate. It helps in organizing and isolating different sets of DDS entities. [27]

4.5.3 Discovery of nodes

DDS uses a distributed discovery system by default. This means that each node (or participant) in the network can discover other nodes without needing a central server. This is achieved through a process called Simple Discovery Protocol (SDP), where nodes periodically broadcast their presence and listen for other nodes' broadcasts.
In particular, for fast-DDS the discovery process consists of two key phases:

1. Participant Discovery Phase (PDP): During this phase, DomainParticipants announce their existence to one another. Each DomainParticipant periodically sends out announcement messages that include their IP address and port information, indicating where they can receive metadata and user data. DomainParticipants in the same DDS Domain match and connect based on these announcements. By default, these messages are **multicast** using well-known addresses and ports. Additionally, the frequency of these announcements can be customized in the discovery settings

2. Endpoint Discovery Phase (EDP): In this phase, DataWriters and DataReaders within the DomainParticipants recognize each other. DomainParticipants exchange information about their respective DataWriters and DataReaders, including details such as the topic and data type. A match occurs only if both the topic and data type align between the two endpoints. Once a match is made, the DataWriter and DataReader can begin transmitting and receiving user data [28]

4.6 Related Works

This section exposes a review of recent research papers that focused on the deployment of 5G-enabled UAVs and the respective performance analyses.

5G-enabled UAVs for energy-efficient opportunistic networking. [29]
The article examines how 5G-enabled UAVs can be leveraged to create opportunistic networks, enhancing network resource management, lowering energy consumption and increasing operational efficiency.
The proposed framework incorporates 5G-enabled drones along with edge command and control software to develop energy-efficient network topologies. Consequently, UAVs perform edge computing for optimized data collection and processing. This advancement utilizes cutting-edge Artificial Intelligence (AI) algorithms to enhance UAV networking capabilities while reducing energy use.

The employed 5G network is provided by Amarisoft whereas the UAV is controlled by a human operator using a ground control station. The UAVs and the related software is provided by Pixhawk 4 (PX4), as in this thesis.

Empirical results demonstrate substantial improvements in network performance with 5G technology compared to older 2.4 GHz systems. Communication failure rates dropped by 50%, from 12% to 6%. Average round-trip times decreased by 58.3%, from 120 ms to 50 ms (with peaks up to 75 ms). Data transmission rates surged from 1 Gbps to 5 Gbps, marking a 400% increase. These findings underscore the significant impact of 5G technology on UAV operations.

5G-Enabled UAVs with Command and Control Software Component at the Edge for Supporting Energy Efficient Opportunistic Networks. [30]

In the paper, the experimental 5G infrastructure for UAV trials was set up using the 5GENESIS facility in Athens, equipped with Amarisoft Core Network and PCIe SDR boards for 5G RAN. A field trial was conducted using a 5G-enabled UAV prototype, which offloaded its flight controller to the edge of the 5G network; the UAV was controlled via a joystick paired with a ground control software. This setup allowed low-latency communication and energy-efficient operation.
The results demonstrated successful UAV control over 5G with a low message loss rate of 6%. The round-trip time for 5G was approximately 50 ms, outperforming the conventional 2.4 GHz link, which averaged 120 ms. The additional 5G equipment on the UAV did not significantly affect energy efficiency due to its lightweight design.
In essence, the trial validated the feasibility of controlling UAVs over 5G and showed potential for future enhancements in energy efficiency through AI-driven optimizations at the network edge.

Chapter 5

UAV Testbed Deployment

In order to test the ability of the OAI 5G network to control a UAV, in principle the components are:

- An **application** that flies the drone on a certain trajectory by transmitting trajectory setpoints
- A **simulator** or a real UAV receiving the trajectory setpoints
- A **5G network** through which the communication happens

So the idea is to instantiate a ROS2 node at the edge of the network (the gNB for example) which communicates through the 5G network towards the simulator (or eventually the real UAV). This chapter presents the steps that led to the deployment of the UAV testbed, with particular regard to the issues encountered during the latter and how to solve them:

1. Development of PX4 offboard control application
2. ROS2 middleware issues
3. UAV simulator issues
4. Computer network topology issues

5.1 PX4/ROS2 Offboard Control Application

The first step in order to test the ability of the deployed 5G network to control a UAV is to develop a suitable benchmark application.

Following the foundations from the example given in the PX4 ROS2 offboard control guide [31], a simple application that flies the drone in a circle was developed. A circular path was chosen as it tests the ability of the UAV to maintain a consistent turn radius and handle *continuous* yaw adjustments; if the turn radius varies at any point, it means that the UAV is suffering from **trajectory error**. However, a limitation of the circular trajectory is that it does not test for abrupt changes in direction.

The ROS2 offboard control node receives `vehicle_local_position` messages from Gazebo simulator and outputs `trajectory_setpoint` messages at a certain publishing frequency (determined

by the user): such messages carry *velocity* setpoints for the UAV, which are used on-board in order to drive the actuators.

The final version of the code for the controller is reported in appendix A.

The core parts of the application include:

- Timer Callback - executes periodically to
  1. Call methods to publish offboard control mode and trajectory setpoint messages
  2. Arm the drone after the first 10 cycles (a steady stream of messages needs to be received by the on-board computer before offboard control can be switched on)
  3. (Eventually) change control frequency after a certain time inverval has elapsed: this was useful when running experiments that sweep control publishing frequency values
  4. Manage log files according to control frequency being used, temporarily stop logging when loss of connection is detected
- Vehicle position subscription callback - updates drone position upon reception of `vehicle_local_position` messages and logs it onto the current log file
- `publish_trajectory_setpoint()` method - publishes **velocity** setpoints for a (anti-clockwise) circular path using a discrete-time PID control law:

$$\begin{cases} x_d(t) = Rcos(\omega t) \quad [m] \\ y_d(t) = Rsin(\omega t) \quad [m] \\ z_d(t) = -h \quad [m] \quad \text{(FRD reference frame)} \end{cases} \implies \begin{cases} \dot{x_d}(t) = -R\omega sin(\omega t) \quad [m/s] \\ \dot{y_d}(t) = R\omega cos(\omega t) \quad [m/s] \\ \dot{z_d}(t) = 0 \quad [m/s] \end{cases}$$

Velocity setpoints in discrete-time with feedback to correct errors:

$$\begin{cases} v_x[k] = -R\omega \sin(\frac{\omega k}{f_{pub}}) + \boldsymbol{K_P} e_x[k] + \boldsymbol{K_I} \frac{\sum_{i=0}^{k} e_x[k]}{f_{pub}} + \boldsymbol{K_D}(e_x[k] - e_x[k-1]) f_{pub} \quad [m/s] \\ v_y[k] = R\omega \cos(\frac{\omega k}{f_{pub}}) + \boldsymbol{K_P} e_y[k] + \boldsymbol{K_I} \frac{\sum_{k=0}^{k} e_y[k]}{f_p ub} + \boldsymbol{K_D}(e_y[k] - e_y[k-1]) f_{pub} \quad [m/s] \\ v_z[k] = 2(-h - z) \quad [m/s] \end{cases}$$

Since the thesis' focus is on the telecommunication aspects of flying the drone, no advanced techniques for PID tuning were used. The PID was manually tuned to achieve trajectory tracking at steady state at $\omega = \frac{2\pi}{10}$ [rad/s] with a control publishing frequency of 10 Hz: $K_p = 2.5, K_i = 1.7, K_d = 0.8$

5.2 ROS2 Middleware Troubleshooting

As of now, **OpenAirInterface 5G does not support UDP multicast**. However, as previously explained, multicast is crucial when ROS2 nodes need to discover each other, since the RMW FastDDS strongly leverages it!
Therefore, an important aspect in ensuring successful communication between the UAV simulator and the offboard controller lies in the choice of the middleware for communications. The adopted solution leverages **Eclipse Zenoh**, which is described in short in the next subsection.

5.2.1 Zenoh

Eclipse Zenoh is a robust data management framework that integrates data in motion, data at rest, and computations. Data in motion refers to information actively being transferred between systems, applications, or devices, often requiring real-time processing. On the other hand, data at rest is information stored on various media, such as databases or hard drives, and remains static until accessed or modified. Zenoh combines traditional publish/subscribe mechanisms with geo-distributed storage and query capabilities, ensuring efficient data handling. In short, the main components of Zenoh include:

- **Zenoh Router (zenohd)**: Routes data between nodes and manages storage and queries
- **Publishers and Subscribers**: Handle data dissemination and reception, akin to DDS DataWriters and DataReaders
- **Storages**: Provide persistent data storage.
- **Queries**: Enable data retrieval in a geographically transparent way, regardless of where data is actually stored
- **Computations**: Allow distributed data processing

This architecture is designed for high efficiency and scalability, making it ideal for edge computing and IoT applications. [32]

5.2.2 Zenoh bridge for ROS2 over DDS

As already mentioned, a solution to the RMW issue leverages **Eclipse Zenoh** middleware, which supports TCP connections between nodes. However, even though Zenoh has been officially adopted as an alternative RMW since September 2023 [33], the integration is still a work in progress: the only feasible path is to use a Zenoh **bridge for ROS2 over DDS**. [34]

The bridge serves as a connector between ROS2 and Zenoh, facilitating communication between the two frameworks. It achieves this by leveraging CycloneDDS to discover DDS entities declared by ROS2 applications and creating corresponding **mirrored entities** in Zenoh: for instance, if a ROS2 application declares a DDS writer, the bridge generates a corresponding DDS reader, which listens for the data to be sent and passes it to a Zenoh publisher. Conversely, if a ROS2 application declares a DDS reader, the bridge generates a DDS writer which receives data from a Zenoh subscriber.

The bridge discovers ROS2 Nodes running on the same Domain ID via UDP multicast, as per DDS specification. Meanwhile, the two bridges may connect through TCP in a peer-to-peer manner (more complicated topologies involving Zenoh routers and clients are also possible). [35]

In order for this to work, the bridge also maps DDS topics read and written by the ROS2 applications to Zenoh resources, ensuring proper declarations for communication between ROS2 and Zenoh. [36]

The commands involved for the setup of the bridge are listed in section 5.3.

5.3 Network topology of UAV Testbed

The obvious next step is to deploy and test the program in different configurations, with the final goal being to run the offboard controller leveraging the OAI 5G network. In this context, different deployments of the application were performed:

1. **Simulator and ROS node running on the same machine** - easiest deployment, can be performed by following steps in tutorial [37] while replacing the code ran by the example with the previously described benchmark application
2. **Simulator on UE, ROS node on gNB** - deployment *failed* due to UE laptop not being able to handle simultaneously running OAI-UE software and Gazebo simulator
3. **Simulator on third computer connected to UE via ethernet, ROS node on gNB** - final setup, investigated in the following subsection

5.3.1 Finalized UAV Testbed

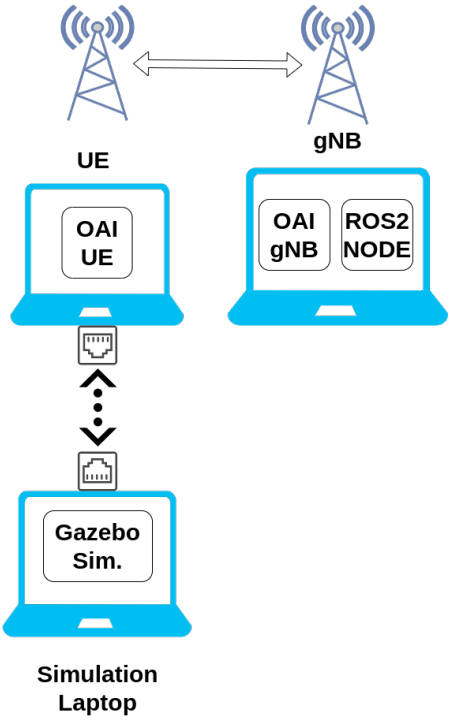


Figure 5.1: Network topology of the UAV testbed

Fig. 5.1 shows the network topology and the programs being run by each entity, whereas fig. 5.2 represents the system from the point of view of the middleware: the bridges are running on the gNB and UE and are connected in a peer-to-peer topology.

In summary, the commands to be run on each computer involve:

- **gNB**:

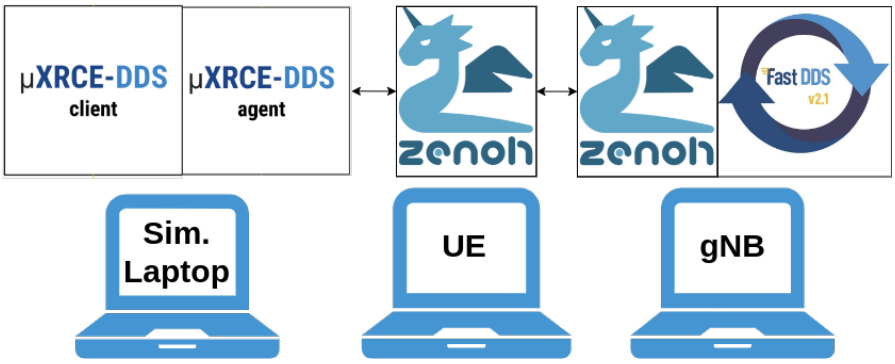


Figure 5.2: Middlewares for communication in UAV testbed

1. Perform system time synchronization using Network Time Protocol (NTP):

```
sudo service ntp stop
sudo ntpd -gq
sudo service ntp start
```

   This step is executed on all laptops, as ROS2 messages are lost when the system clocks are out of sync

2. Deploy OAI5G core network and gNB (version 2.1.0) in band 78 with configuration file *gnb.sa.band78.fr1.106PRB.usrpb210.conf* (described in section 3.1)
3. Configure correct routing from gNB towards the UE via the oai-upf

```
sudo ip route add 10.0.0.0/26 via 192.168.70.134 dev demo-oai
```

4. Source the ROS2 environment (including the PX4_ros_com repo *local_setup.bash* file) and run the Zenoh/DDS bridge as a peer looking to connect to the UE

```
cd ~/Documents/px4-ROS2_ws/;
source /opt/ros/humble/setup.bash;
source install/local_setup.bash;
./zenoh-bridge-ros2dds -m peer -e tcp/<UE IP>:12345
```

5. Source the ROS2 environment (as in previous step) and proceed to run the offboard control ROS2 node

```
ros2 run px4_ros_com offboard_control
```

- **UE**:
  1. Perform system time synchronization using NTP as for the gNB
  2. Deploy OAI5G UE (version 2.1.0) and make sure it properly connects to the gNB

3. Configure routing towards gNB

```
sudo ip route add 192.168.70.129 dev oaitun_ue1
```

4. Source the ROS2 environment and run the Zenoh/DDS bridge as a peer listening on a designated port

```
./zenoh-bridge-ros2dds -m peer -l tcp/0.0.0.0:12345
```

5. Check that the ROS2 node topics are available at the UE

- **Simulator Laptop**:

1. Perform system time synchronization using NTP as for the UE and gNB
2. Run μXRCE Agent, which will be discovered by the Zenoh bridge running on the UE

```
MicroXRCEAgent udp4 -p 8888
```

3. Navigate to the PX4 Autopilot folder (version 1.15.0, git hash ee2a8c9bda06425c4c78e72455059692309431b1) and run Gazebo simulator (automatically also runs μXRCE Client and connects to the agent) with the default UAV model (X500 Quadrotor)

```
make px4_sitl gz_x500
```

Theoretically, the commands listed above should be enough. However, initially the ROS2 messages did not seem to reach the μXRCE Client and therefore the simulator, even though messages from the simulator were correctly reaching the ROS2 offboard controller on the gNB. After a considerable effort, it seems that the cause was that

⚠ **uXRCE-DDS Agent was not subscribing to the topics published by the ROS2 node on the gNB. The way around this is to create subscriptions manually at the simulator side via *ros2 topic echo* commands**

3. Therefore, 3 additional commands were executed:

```
ros2 topic echo /fmu/in/offboard_control_mode
ros2 topic echo /fmu/in/vehicle_command
ros2 topic echo /fmu/in/trajectory_setpoint
```

Figure 5.3: Picture of the experimental apparatus

Chapter 6

UAV Testbed Analysis

This chapter deals with the measurements and experiments performed in order to characterize the deployed testbed, especially in terms of:

- **ROS2 topic throughput** - to quantify the networking load on the system
- **ROS2 application round-trip time** - crucial parameter in determining the responsiveness and alertness of the UAV
- **Trajectory Error** - determines how well the UAV is able to follow the predefined circular trajectory. This parameter was studied as a function of the angular velocity of the UAV and publishing frequency of the control

6.1 ROS2 topics throughput

First of all, it is important to measure the throughput of the ROS2 topics involved, in order to quantify the network load being exerted. This can be done via the `ros2 topic bw` command:

- `vehicle_local_position` - uplink
- `trajectory_setpoint` - downlink
- `offboard_control_mode` - downlink
- 'talker' topic - downlink
- 'listener' topic - uplink

Figure 6.1 shows the occupied bandwidth by each PX4-ROS2 topic as function of publishing frequency of the control, which is in general in the range of 10s of kB/s. As expected, the bandwidth occupied by `trajectory_setpoint` and `offboard_control_mode` messages linearly increases with control publishing frequency, doubling whenever the publishing frequency is doubled.
Concerning `vehicle_local_position` topic, its message frequency is not affected by the control publishing frequency, therefore the value remains constant at around 20 kB/s.
Lastly, the modified talker node (explained in section 6.2.1) occupies a bandwidth of 290 B/s when recording RTT one time per second, while the listener 320 B/s.

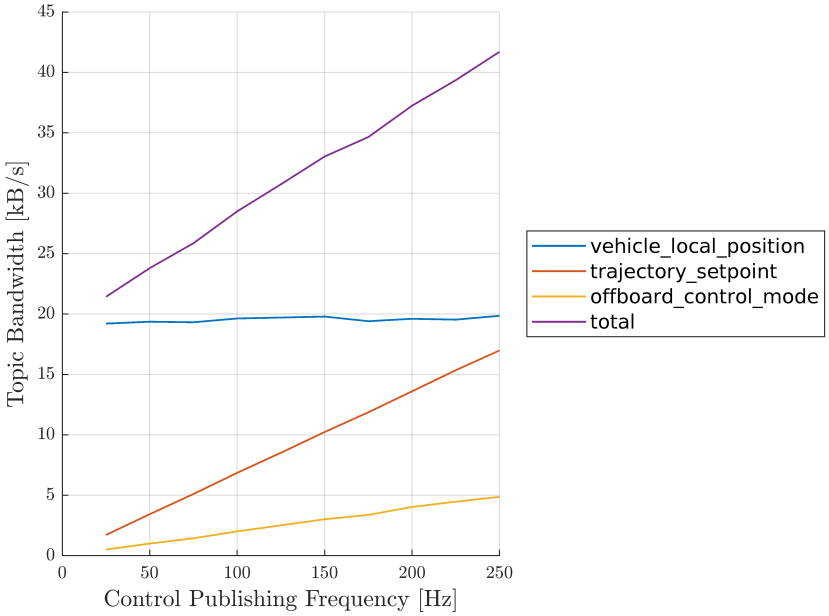


Figure 6.1: ROS2 topic bandwidth vs. control publishing frequency

6.2 ROS2 Application Round-trip Time

Real-time control and predictability are crucial when operating a UAV via ROS2 nodes: the UAV's stability and responsiveness depend on timely communication between the ground station and the UAV.
Round-trip time (RTT) is a measure of communication latency: low RTT ensures that the control algorithm is making use of close to real-time data. Accurate control relies on predictable communication delays, allowing control algorithms to make precise adjustments to the UAV's flight path.
Additionally, RTT measurements provide valuable diagnostics for network performance, helping to identify issues like congestion or packet loss, which can then be addressed to optimize the control system's effectiveness.

For these reasons, it is important to characterize the testbed in terms of round-trip time (RTT), especially at the ROS2 application layer, where the control algorithm is located.

Considering the testbed setup, the primary factors affecting RTT in the ROS2 application layer include:

- Control command processing time: The time required for processing and generating control commands within the ROS2 node at the gNB

- Zenoh bridge latency at the gNB: The communication delay introduced by the first Zenoh bridge located at the gNB.

- gNB-UE communication latency: The latency in data transmission between the gNB and the UE during both the uplink and downlink phases.
- Zenoh bridge latency at the UE: The delay introduced by the second Zenoh bridge located at the UE.
- UE-Simulator communication latency: The time delay in communication between the UE and the simulator laptop (connected via Ethernet)

6.2.1 Talker/Listener nodes

In order to measure RTT, a modified version of the ROS2 talker/listener nodes was employed:

- **Talker**: Simulates a message publisher in a ROS 2 system that logs round-trip latencies between itself and a list of known listeners
- **Listener**: Simulates a message responder that listens to messages from the Talker and sends back a response, facilitating the calculation of round-trip latencies

In summary, the communication flow is the following:

1. The Talker node periodically publishes a message containing a timestamp and its name to the "*topic*" topic. The period (and therefore the frequency) of the publication can be modified by the user
2. The Listener node subscribes to the "*topic*" topic, receives the message, extracts the timestamp, and publishes a response containing the original timestamp and its own name to the "*response_topic*" topic
3. The Talker node subscribes to the "*response_topic*" topic, receives the response, calculates the round-trip latency, and logs this data if the connection is active. If no response is received for a certain time interval, it stops logging until the connection is restored (this is useful if for some reason the 5G connection is lost). Multiple listener nodes are also supported, allowing simultaneous measurement of RTT towards different destinations.

The full implementation of the talker and listener nodes can be found in appendix B.

6.2.2 RTT measurements

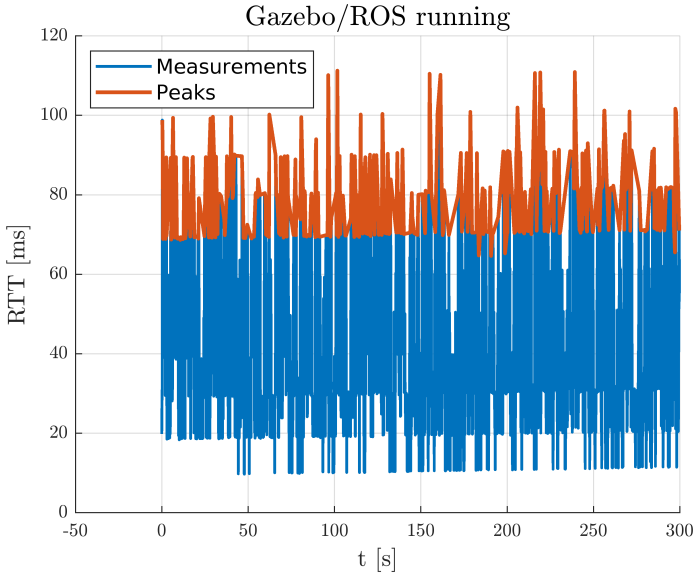
Figure 6.2 shows 2 plots of RTT measurements over a span of 5 minutes, taking 10 RTT measurements every second and keeping the simulated UAV at the UE in a fixed position compared to the gNB. Plot **(a)** refers to the "*Gazebo/ROS not running*" case, when only the talker and listener nodes are active without any other load on the network; on the other hand, plot **(b)** is for the "*Gazebo/ROS running*" case, when the whole testbed (including Gazebo simulator and the PX4 ROS2 node) is deployed.

Both situations encompass variability of the RTT over time, in fact a **"*spiking*" behavior** is observed: the latency spikes, already present in case (a), are a lot more frequent in case (b). In addition, the height of the peaks drammatically increases, going from 24 ms in case (a) to around 110 ms in case (b)!

The **average RTT** correspondingly increases, going from 10.9 ms (a) to 42.9 ms (b).

The high jitter is detrimental for the control of the UAV, because it produces inconsistent and unpredictable communication delays, reducing momentarily the accuracy of the control. The impact of such phenomenon will be quantified in section 6.3, which concerns the trajectory error.

(a)



(b)

Figure 6.2: RTT measurements over 5 minutes

6.2.3 Latency Spikes Investigation

It seemed necessary to take a deeper look at what was happening inside the system in order to try to justify the RTT "spikes". In particular, the objective was to gather information from the lower layers of the 5G protocol stack, such as a MAC or RLC.

Since there is no official documentation from OAI on how to retrieve such measurements, the first thing that came to mind was exploiting RNIS, however similar issues as for deploying OAI-MEP (which uses RNIS) were encountered. Secondly, diving into the OAI logging facility some useful measurements could be retrived, but the stream of text was several MB/s and it was having an impact on the performance of the laptops, leading to frequent crashes.

Other solutions were attempted, and finally the **E2 Agent** came to the rescue.

E2 Agent

The E2 Agent is a network function within the O-RAN architecture that acts as an interface point between the E2 nodes (such as gNBs, eNBs, and other RAN nodes) and the near-RT RIC (near-Real-Time RAN Intelligent Controller).
It collects real-time data and metrics from the RAN and sends this data to the near-RT RIC for processing and analysis. It also receives control commands and policies from the near-RT RIC to be enforced on the RAN elements.

The tutorial on how to setup the OAI 5G RAN with integrated E2 agent is found at [38].

⚠ **When building FlexRIC, if the compiler is issuing '*segmentation fault*' errors, the fix is to use exactly CMake v3.15 via *update-alternatives* tool or via direct install/downgrade**

Once the E2 Agent is successfully setup, different xApps may be started to provide monitoring and control over the RAN. xApps are specialized software applications that run on the nearRT-RIC designed to perform various radio network functions, analytics, and control tasks to optimize the performance of the RAN.
In particular, within the already provided xApps, the *(MAC + RLC + PDCP + GTP) monitor xApp* was found to provide a log with a large number of KPI measurements coming from different layers of the 5G protocol stack.

RAN KPIs Plots

The idea is to perform RTT measurements at the same time as KPI measurements and try to extract as much information as possible to justify the observed RTT spikes.

The designed experiment involves:

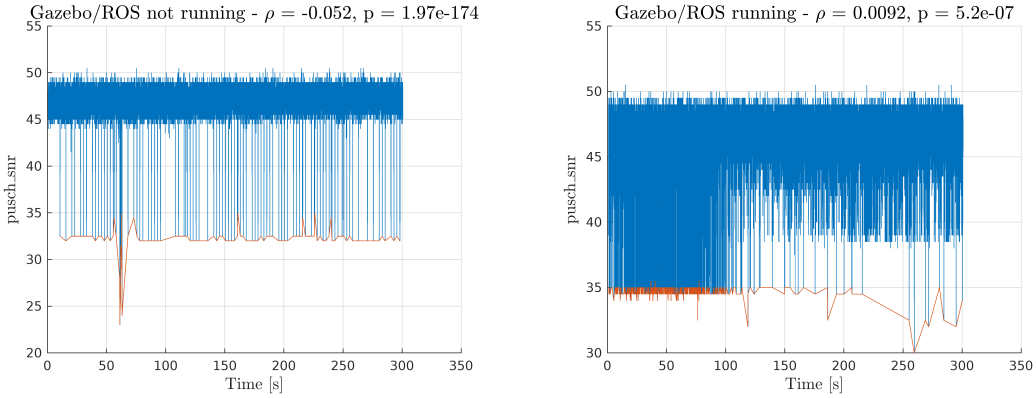
- Length: 5 minutes
- Talker/listener ROS2 nodes measuring RTT 10 times a second (RTT measurement instant is attributed to starting instant of measurement)

- *(MAC + RLC + PDCP + GTP) monitor xApp* logging RAN KPIs 1000 times a second
- Gazebo Simulator: publishing UAV position 250 times a second. This corresponds to the frequency at which drivers publish their state to the control in real vehicles
- PX4 ROS2 node: publishing UAV control commands 10 times a second

The experiment was repeat twice, once when Gazebo Simulator and PX4 ROS2 node were not running (**case (a)**) and once when they were (**case (b)**); each of the KPI plots also reports the **correlation coefficient** with respect to the RTT measurements and the corresponding **p-value**. The obtained graphs for the RTT are already reported in figure 6.2.

Figure 6.3 shows the evolution of the *pusch_snr* over time, the envelope of the peaks is drawn in red. The *pusch* (Physical Uplink Shared Channel) is the channel used for transmitting user data from the UE to the gNB, therefore *pusch_snr* is a measure of the quality of transmission of user data in uplink.
In case (a), a spiking behavior is once again observed: most measurements are between 45 and 50 dB, whereas the negative peaks are at around 32 dB. The correlation with RTT seems to be low; nevertheless, comparing the instant where the peaks happened:

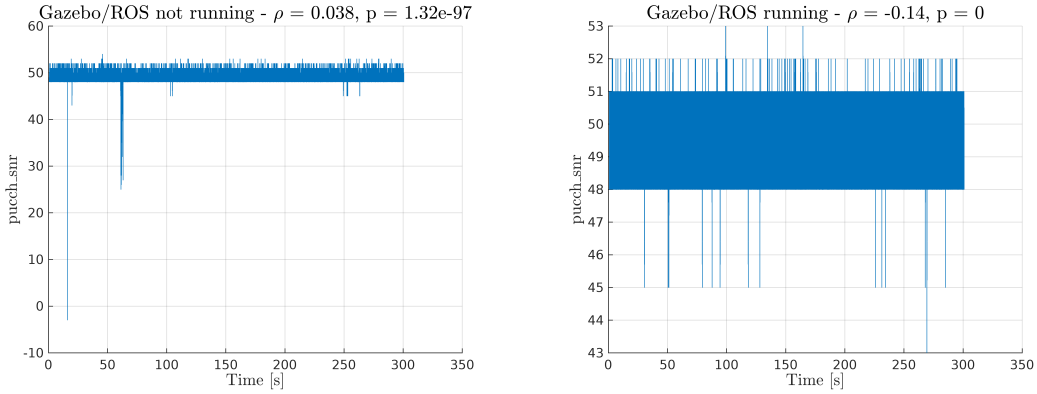
For case (a), 63% of spikes in RTT happen within 40 ms of a spike in *pusch_snr*.
For case (b), this percentage lowers to around 20%, so other factors must be at play.



(a) **(b)**

Figure 6.3: pusch_snr measurements

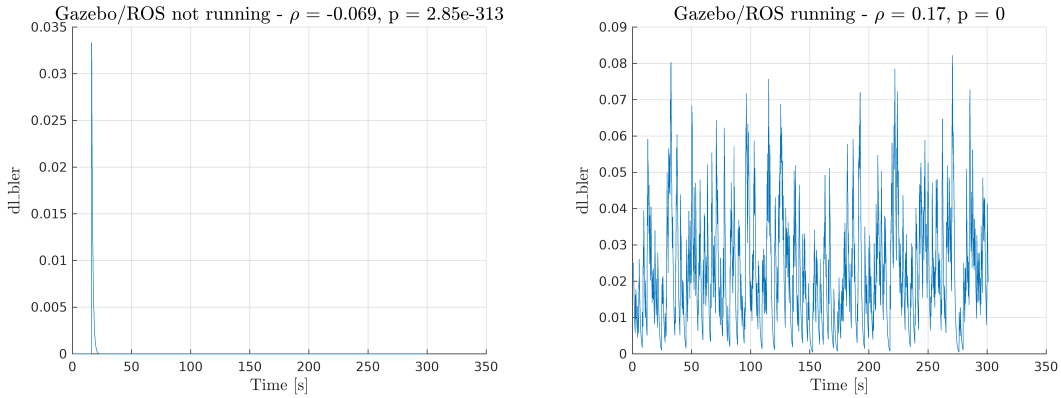
On the other hand, pucch_snr (fig. 6.3) is a lot more stable around 50 dB, meaning that control channel conditions are relatively consistent.
Block Error Rate (BLER) represents the percentage of transmitted data blocks that contain errors and cannot be decoded correctly by the receiver.
Taking a look at figures 6.5 and 6.6, it seems that both the *dl_bler* and the *ul_bler* are almost always zero in (a) but the situation drastically changes in (b): the BLER fluctuates a lot and it also has some correlation with RTT ($\rho = 0.17$)!



(a) **(b)**

Figure 6.4: pucch_snr measurements

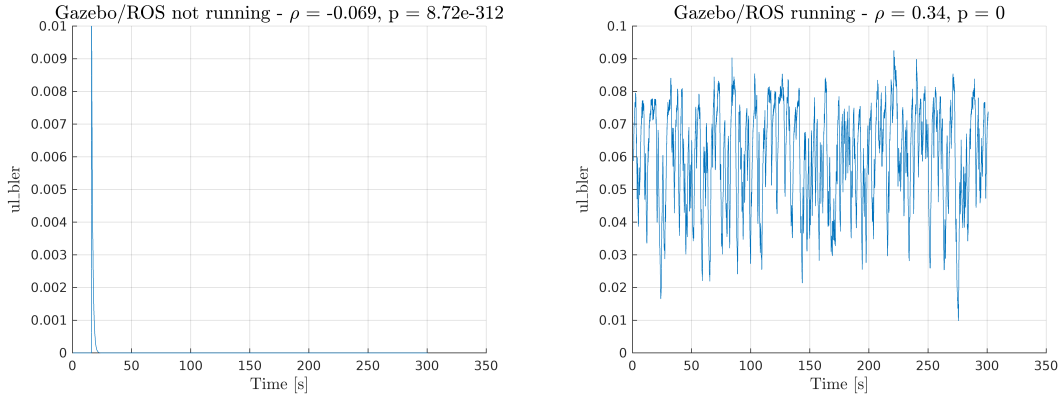
It seems that the spiking of BLER might induce the tall spikes in RTT (the ones up to 100 ms) when the UAV testbed is fully deployed. The fact that the BLER is different from 0 only when the application is interesting, because at the start of the thesis the maximum throughput of the gNB was tested to be in the 10s of MB/s, so network congestion should not be an issue (the load is in the kB/s range). For this reason, the increase in BLER might be attributed to the high CPU load exerted by simultaneously running the OAI 5G network, the Zenoh bridges and having to pass the ROS2 messages.



(a) **(b)**

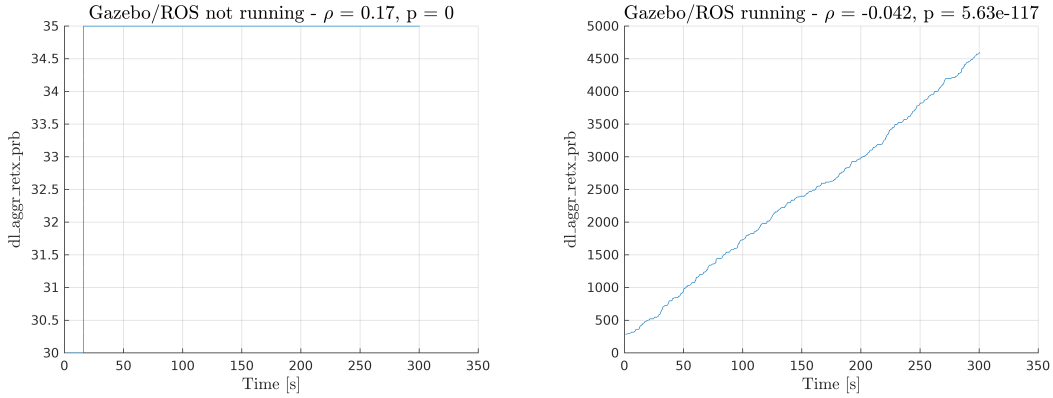
Figure 6.5: Downlink BLER measurements

Figures 6.7 and 6.8 show the log of the cumulative number of PRBs used for retransmissions in DL and UL. Retransmissions are happening only when the UAV testbed is fully deployed, therefore when the BLER is also fluctuating, as expected.
The last KPI reported is the transmission buffer occupancy in bytes (fig. 6.9), which reflects how much data is queued for transmission over the air interface. It can be noted that the metric almost doubles from (a) to (b); as the buffer fills up, packets have to wait longer in the queue



(a) **(b)**

Figure 6.6: Uplink BLER measurements



(a) **(b)**

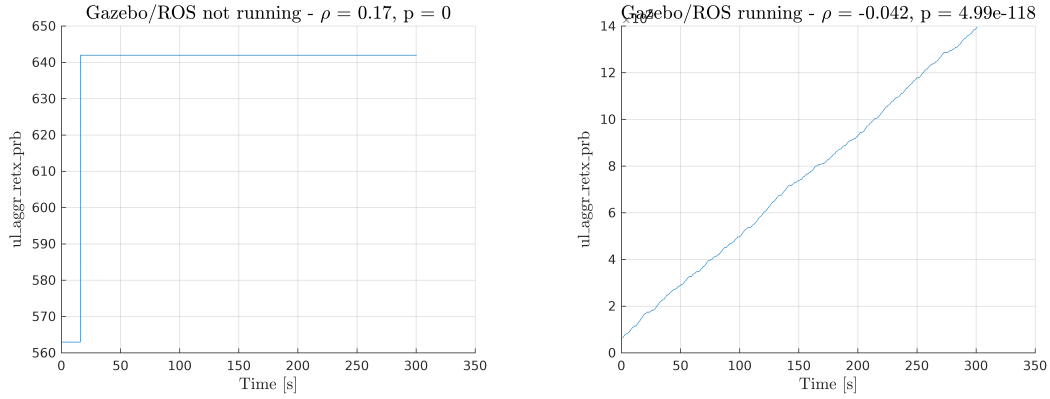
Figure 6.7: Downlink aggregate retransmission physical resource blocks

before being transmitted, which might be one of the reasons why the RTT increases in (b). From this experiment it is possible to draw that:

- When the UAV testbed is not deployed (case (a)), the observed spikes in RTT happen mainly due to the spikes in SNR

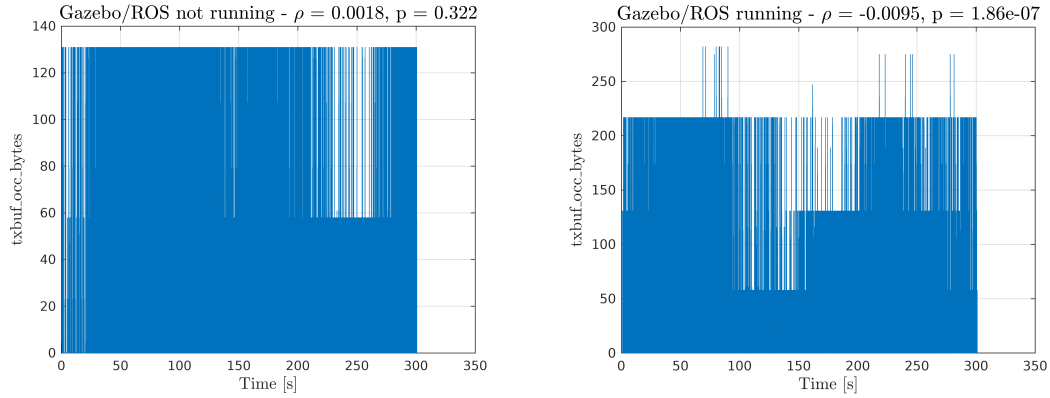
- When the UAV testbed is deployed (case (b)), the taller and more frequent spikes in RTT are due to BLER spikes, which induce retransmissions of packets. The variability in BLER might be caused by the high CPU load exerted when simultaneously running the OAI 5G network, the Zenoh bridges and having to pass the ROS2 messages.

  This evidence is in line with the fact that the employed laptops do not respect the minimum system requirements to run OAI5G (see section 3.1)



(a) **(b)**

Figure 6.8: Uplink aggregate retransmission physical resource blocks



(a) **(b)**

Figure 6.9: Transmission buffer occupancy in bytes

6.3 Trajectory Error

Trajectory error refers to the deviation between the UAV's actual flight path and the desired or planned trajectory. The trajectory of a UAV is the path it is intended to follow, which can be defined in terms of position, velocity, and sometimes orientation over time.
Trajectory error can arise due to various factors, such as:

- Communication Delay: Since RTT is not zero, the control algorithm will not be using the most recent information and commands will be received by the UAV with a certain delay. This is the most relevant source of error considering the scope of the thesis.

- Actuator Limitations: The trajectory might be impossible to follow for the UAV, which might happen because the actuators are not able to produce the required level of thrust within the dynamic constraints of the trajectory

- Environmental Disturbances: Such as wind gusts, turbulence, or changes in atmospheric pressure.
- Sensor Noise: Imperfections or inaccuracies in the sensors used for navigation (e.g. GPS errors).
- Modeling Errors: Differences between the UAV's actual dynamics and the model used for control.

By analyzing trajectory error, we can gauge the effectiveness of the control system, particularly when dealing with factors like communication delays, actuator limitations, etc. This evaluation is essential for ensuring safety, especially in environments where precision is critical, such as urban areas or close to obstacles. Furthermore, minimizing trajectory error contributes to more efficient flight, reducing unnecessary maneuvers.
Finally, understanding and mitigating trajectory error also allows for the optimization of control algorithms, sensor processing, and actuator performance, ultimately leading to more reliable and predictable UAV operations.

As explained in chapter 5, the simulated UAV receives *velocity* setpoints from the ROS2 node in order to follow a circular path. An advantage of this shape is that it is particularly simple to determine how accurately the UAV is following the trajectory:

1. **Define the Ideal Circular Trajectory:**

   Assume the ideal circular path has a center at (x_c, y_c) and a radius R. The trajectory can be parameterized in a 2D plane as:

   $$x(t) = x_c + R\cos(\omega t),$$
   $$y(t) = y_c + R\sin(\omega t),$$

   where ω is the angular velocity (in rad/s) for the circular motion.

2. **Determine the UAV's Actual Position:**

   Let the UAV's actual position at time t be $(x_{\text{actual}}(t), y_{\text{actual}}(t))$.

3. **Calculate the Radial Distance to the Circle's Center:**

   For the UAV's actual position $(x_{\text{actual}}(t), y_{\text{actual}}(t))$, compute its radial distance from the center of the ideal circle:

   $$d(t) = \sqrt{(x_{\text{actual}}(t) - x_c)^2 + (y_{\text{actual}}(t) - y_c)^2}.$$

4. **Compute the Instantaneous Trajectory Error:**

   The instantaneous trajectory error $e(t)$ at time t is the absolute difference between this radial distance $d(t)$ and the radius R of the ideal circle:

   $$e(t) = |d(t) - R|$$

For the evaluated UAV testbed:

- $x_c = 0\ m\ ,\ y_c = 0\ m$
- Position logging is performed at the PX4 ROS2 node side

6.3.1 vs. Angular Velocity

This section is concerned with the evaluation of trajectory error as function of the angular velocity of the drone while using a constant control publishing frequency. The trajectory to be followed is as always circular and with a 2 meter radius, the type of control is PID with coefficients set as in the previous chapter.
The experiments in this and the following sections are performed in 2 situations (represented in figure 6.10):

(a) **'5G' case** - PX4 ROS2 node running on the gNB, Gazebo simulator running on the simulator laptop, which is located in a fixed position with respect to the gNB. ROS2 messages have to travel through the OAI 5G network

(b) **'Localhost' case** - PX4 ROS2 node running on the simulator laptop alongside Gazebo. ROS2 messages are exchanged via the loopback interface. This situation acts as the control for the experiment and provides the lowest and most stable RTT (optimal control of the drone)

The methods for the experiment involve:

- Control publishing frequency $f_{pub} = 10$ Hz or 20 Hz
- Progressive sweeping of values of angular velocity ω, going from $\frac{2\pi}{40}$ (where trajectory error is negligible) to $\frac{2\pi}{4}$ (where control visibly failed trajectory tracking) with a $\frac{2\pi}{80}$ step
- Each value of angular velocity is kept for 60 seconds before moving to the next one
- Trajectory is logged directly by the PX4 ROS2 node whereas RTT is also recorded every second via the talker/listener nodes

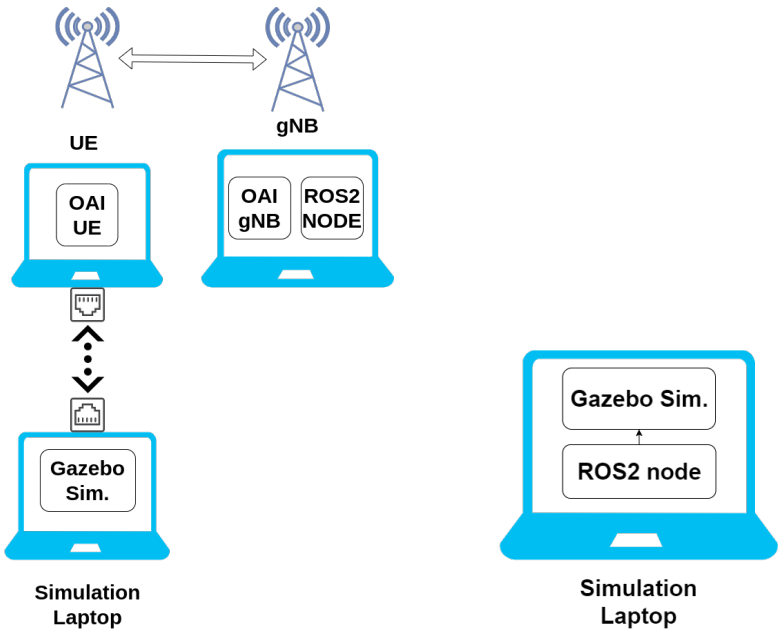
Figures 6.11 and 6.13 report the average value and the standard deviation of the trajectory error as a function of the angular velocity of the drone. At a first glance, all curves remain close to each other until $\omega = 1.4\ rad/s$, afterwards trajectory error has a sudden increase and the difference between '5G' and 'Localhost' cases becomes relevant (at both 10 and 20 Hz control publishing frequency). For this reason, from now on this velocity will be referred to as the **critical speed**:

$$\omega_{crit} = 1.4\ rad/s$$

The error starts to become different from zero at 0.4 rad/s, corresponding to 0.8 m/s of tangential velocity. The value might be improved by different tuning of the PID constants (which was not a main focus for the thesis), although it could also be a physical limit of the actuators of the simulated UAV (if this is the case, modifying the radius of the trajectory might move this point).

What is more of concern for the thesis is the difference between the trajectory error over 5G and the one over Localhost in figure 6.12. It can be seen that the performance of the 5G case deviates significantly from the Localhost case starting from the critical speed. In particular there seems to be a positive peak at the critical speed followed by a valley, both of which are more pronounced when the control publishing frequency is lower. The full effect of modifying control publishing frequency will be discussed in section 6.3.2.

The next analysis has to do with the standard deviation of the trajectory error, which quantifies how consistently the trajectory error is committed. For instance, if the UAV follows a *perfect*



(a) '5G' **(b)** 'Localhost'

Figure 6.10: Reminder of the UAV system configuration

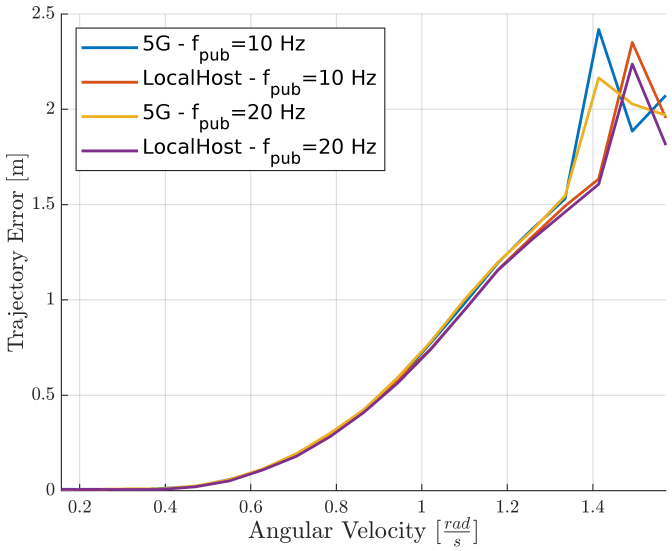


Figure 6.11: Mean trajectory error over 60 seconds vs. angular velocity

circle of radius 3 m instead of 2 m, then the average trajectory error will be 1 m and the standard deviation will be 0 m. Conversely, if the 3 m circle presents imperfections, meaning that the distance of the UAV from the origin is variable over time, the standard deviation will be greater than 0 m.

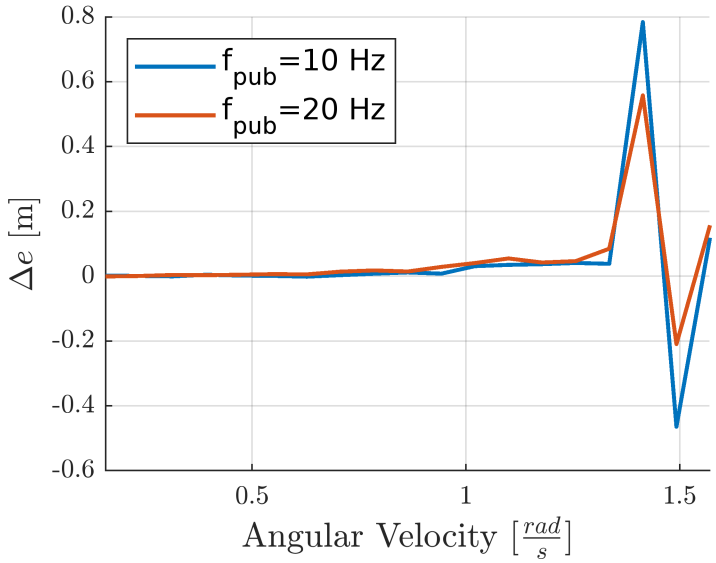


Figure 6.12: Difference between mean trajectory error of 5G case and localhost case

In other words, it measures how close the trajectory is to a circle (with any radius!).

The behavior of the standard deviation of the trajectory error as a function of angular velocity is similar to the one of the average trajectory error, however the interpretation of the plot differs: when the UAV reaches the critical speed, the traced path moves farther away from a circle. In fact, as can be seen in figure 6.15, the trajectory over 5G becomes inconsistent across different laps, whereas this does not happen for the Localhost case.

6.3.2 vs. Control Publishing Frequency

The second experiment involves keeping the angular velocity constant and varying control publishing frequency instead.
As observed in the previous section, there seems to be a threshold angular velocity, the critical speed, after which the trajectory error explodes and the '5G' curve becomes distant from the 'LocalHost' one. This phenomenon is observed at both 10 and 20 Hz control publishing frequency.

A question arises: is it possible to regain control of the UAV at the critical speed by employing a higher control publishing frequency?
In order to provide an answer, an experiment was set up, similar to the angular velocity one:

- Angular velocity fixed to the critical speed $\omega = \omega_{crit} = 1.4 rad/s$
- Progressive sweeping of values of control publishing frequency f_{pub}, going from 5 to 50 Hz (and eventually up to 250 Hz, equal to the default UAV position publishing frequency in Gazebo) with a 5 Hz step
- Each value of publishing frequency is kept for 60 seconds before moving to the next one

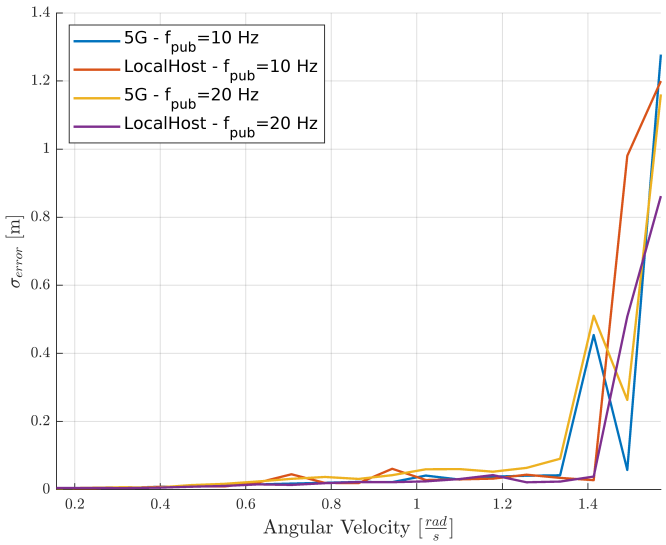


Figure 6.13: Standard deviation of trajectory error over 60 seconds vs. angular velocity

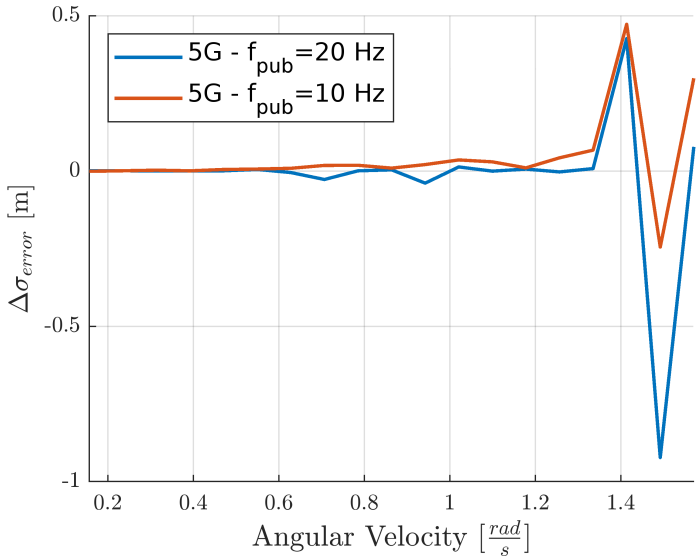


Figure 6.14: Difference between standard deviation of trajectory error of 5G case and Localhost case

- Trajectory is logged directly by the PX4 ROS2 node whereas RTT is also recorded every second via the talker/listener nodes (as in the previous experiment)

Figure 6.16 plots the average trajectory error for each value of control publishing frequency. Trajectory error seems to be improved by increasing f_{pub} only up to a certain point, which is reached at 10-15 Hz for both cases.

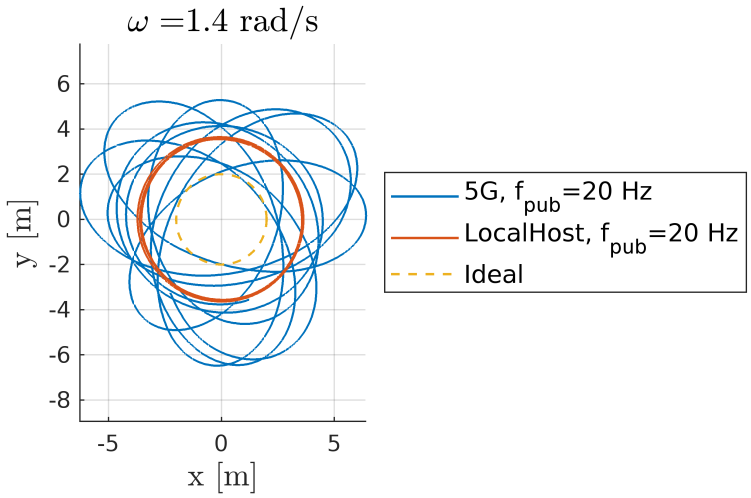


Figure 6.15: Trajectory of UAV at the critical velocity, 20 Hz control publishing frequency

Comparing the 5G and Localhost curves, it seems that the two are similar, however the 5G error is consistently about 1 meter higher than the Localhost one. A similar behavior occurs when analyzing the standard deviation of the trajectory error (figure 6.17): the Localhost curve converges to a low value (only a few centimeters) after 15 Hz, whereas the 5G curve is always higher than 0.8 m, which means that trajectory is remains far away from the ideal circle.

Given this evidence, unfortunately, the 5G the ROS2 node is never able to properly control the UAV at the critical speed with the deployed PID control algorithm. Further confirmation of this can be obtained by taking a direct look at the trajectory in figure **??**: the path does not resemble a circle even at 50 Hz control publishing frequency.
Increasing control publishing frequency over 50 Hz up to the simulator publishing frequency (250 Hz) provides delivers comparable or worse performance at critical speed, even when the node runs on Localhost. This might be due to the PID control loop becoming unstable at higher frequencies, however this was not investigated further as it is not a main focus for the thesis.
In addition, it was observed that RTT over 5G does not seem to be strongly influenced by the control publishing frequency: the average over 1200 measurements in a 10 minute time span was 42.69 ms at $f_{pub} = 25$ Hz and 42.95 ms at $f_{pub} = 250$ Hz. This is in line with the fact that the available throughput of the network (around 14.6 MB/s) is much larger than the utilization in the testbed, which is in the kB/s range.
Similarly, LocalHost RTT does not signficantly vary with control publishing frequency either, with an average of 0.59 ms throughout the whole frequency range.

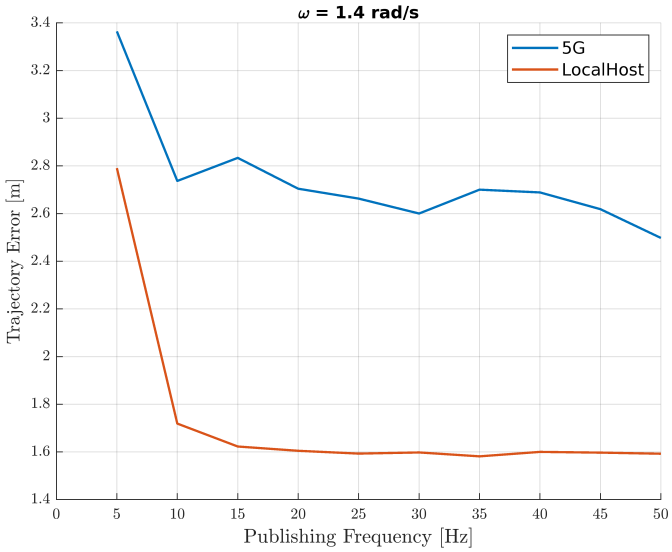


Figure 6.16: Mean trajectory error over 60 seconds vs. control publishing frequency, $\omega = \omega_{crit} = 1.4\ rad/s$

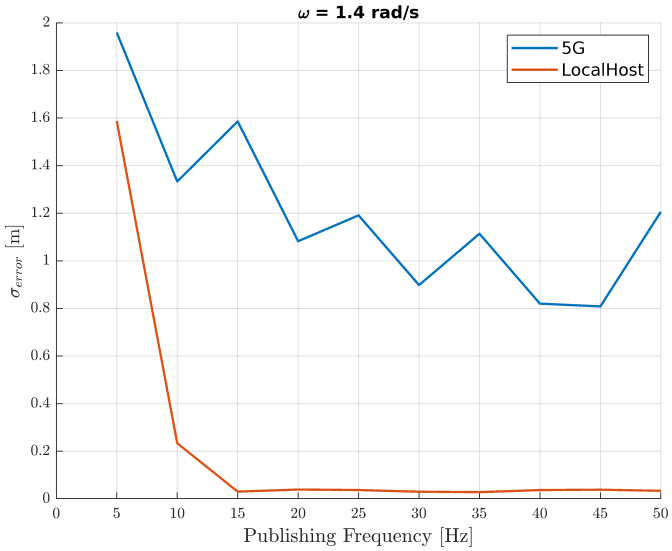


Figure 6.17: Standard deviation of trajectory error over 60 seconds vs. control publishing frequency, $\omega = \omega_{crit} = 1.4\ rad/s$

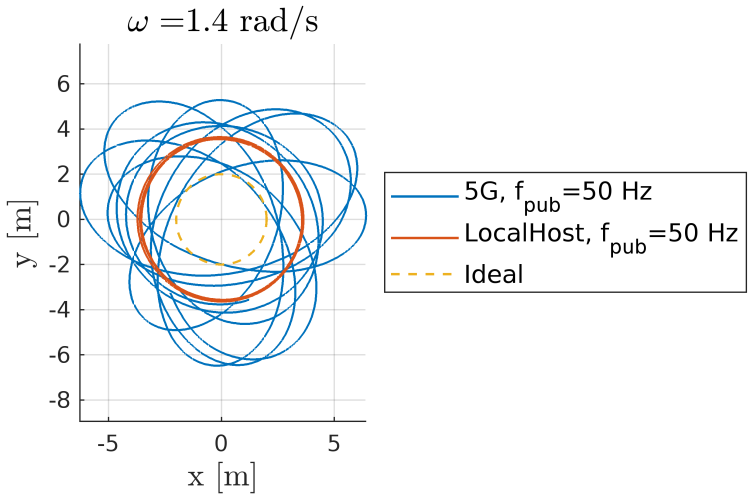


Figure 6.18: Trajectory of the UAV at critical speed, 50 Hz control publishing frequency

Chapter 7

Conclusions

The thesis aimed to achieve control of a simulated UAV via an experimental 5G network, exploiting edge computing to minimize latency.

Unlike previous generations of mobile technologies, such as 4G LTE, 5G provides substantial advancements that are particularly beneficial for UAV operations: the lower latency and higher reliability ensure responsive control, which is crucial for maintaining precision during complex maneuvers in dynamic environments.

The edge computing paradigm has recently emerged as an opportunity for the UAV to outsource demanding tasks, such as flight planning and control computations, to the network's edge. By leveraging edge computing, these computationally heavy operations are processed close to the UAV, achieving low control latency and longer battery life.

The research carried out in this thesis demonstrates the potential of a 5G-enabled UAV control system through its detailed implementation, deployment and performance analysis.

Deployment of the UAV testbed involved troubleshooting multiple issues and bugs related to OpenAirInterface5G, PX4 Autopilot and the ROS2 environments, especially when it came to communication between the PX4 ROS2 node and the Gazebo simulator. All fixes were documented for reproducibility purposes.

The analysis involved the characterization of the testbed in terms of round-trip time (RTT), trajectory error and occupied network bandwidth.

The behavior of the ROS2 application layer RTT was studied, especially when it came to its variability over time (the 'spiking' phenomenon). The OAI5G E2 Agent enabled the recording of network KPIs, which revealed that RTT spikes are mainly caused by the *pusch_snr* when the simulator and PX4 ROS2 control node are not running, whereas the worsening of the phenomenon when the testbed is fully deployed is explained by the variability in BLER, possibly due to the intense CPU load on the gNB/UE (which, as noted in section 3.1, do not comply with the minimum hardware requirements to run OAI5G).

Trajectory error between the desired and actual UAV paths was also investigated, in particular when varying angular velocity over the circular path and control publishing frequency. With the employed PID position control algorithm, there exists a critical angular velocity at 1.4 rad/s after which control over the UAV is lost and trajectory error over 5G differs greatly when compared to the low-latency 'LocalHost' control. For this reason, when following a curved

trajectory, tangential velocity shall be kept strictly under 10.1 km/h when planning trajectories with this specific testbed. With such speeds, a control publishing frequency of 10 Hz is sufficient and further increases lead to similar or worsened performance in terms of trajectory error.

A main contribution of this research lies in the knowledge gathered in the deployment of the UAV testbed, especially concerning the establishment of communication between ROS2 and the UAV simulation environment via the OAI5G custom network. Overcoming bugs was the greatest difficulty in this work, and the knowledge on how to fix them will speed up future deployments.

Overall, this thesis was able to achieve UAV control via an experimental 5G network, underscoring the transformative potential of 5G technology combined with edge computing in UAV control and marking a departure from the limitations of earlier mobile networks.

7.1 Future Works

The main limitations in this study lie in the use of a simulated UAV and in the restricted computational power of the gNB/UE. The former implies that the experiment may not fully capture the variability in channel conditions, especially SNR and other environmental factors, that would naturally occur in real-world scenarios. The latter caused problems in terms of RTT variability, which should not happen in a state-of-the-art deployment.

Future work could concentrate on repeating the performance evaluation with an actual UAV and on mitigating RTT variability by applying a URLLC slice to the UAV. In addition, further investigations on the deployment OAI MEC might be performed and the technology could be leveraged to further improve UAV control performance. Another direction could be replacing the velocity control algorithm with one that anticipates the effects of latency and adjusts commands accordingly.

List of Figures

List of Tables

Appendix A

Offboard Control C++ Application

content/chapters/5/docs/offboard_control_f_exp.cpp

```cpp
#include <px4_msgs/msg/offboard_control_mode.hpp>
#include <px4_msgs/msg/trajectory_setpoint.hpp>
#include <px4_msgs/msg/vehicle_command.hpp>
#include <px4_msgs/msg/vehicle_local_position.hpp>
#include <rclcpp/rclcpp.hpp>
#include <rmw/qos_profiles.h>
#include <stdint.h>

#include <chrono>
#include <iostream>
#include <fstream>
#include <cmath>
#include <limits>
#include <thread>

using namespace std::chrono;
using namespace std::chrono_literals;
using namespace px4_msgs::msg;

const double PI = 3.14159265358979323846;

class OffboardControl : public rclcpp::Node
{
public:
    OffboardControl()
        : Node("offboard_control"),
          timer_(nullptr),
          offboard_control_mode_publisher_(nullptr),
          trajectory_setpoint_publisher_(nullptr),
          vehicle_command_publisher_(nullptr),
          vehicle_local_position_subscription_(nullptr),
          offboard_setpoint_counter_(0),
          radius_(2.0),
          altitude_(5.0),
          angular_velocity_(1.41372),
          control_frequency_(25.0),
          min_control_frequency_(1.0),
          max_control_frequency_(251.0),
          control_frequency_step_(75.0),
          log_interval_(600),
          last_log_time_(this->now().seconds()),
          last_position_time_(this->now()),
          current_x_(0.0),
          current_y_(0.0),
          current_z_(0.0),
```

```
46            prev_error_x_(0.0),
47            prev_error_y_(0.0),
48            integral_error_x_(0.0),
49            integral_error_y_(0.0),
50            logfile_(nullptr),
51            logging_active_(true),
52            logfile_index_(0)
53        {
54            offboard_control_mode_publisher_ = this->create_publisher<OffboardControlMode>("/fmu/in
      /offboard_control_mode", 10);
55            trajectory_setpoint_publisher_ = this->create_publisher<TrajectorySetpoint>("/fmu/in/
      trajectory_setpoint", 10);
56            vehicle_command_publisher_ = this->create_publisher<VehicleCommand>("/fmu/in/
      vehicle_command", 10);
57
58            // Use rmw_qos_profile_sensor_data for subscription
59            rmw_qos_profile_t qos_profile = rmw_qos_profile_sensor_data;
60            auto qos = rclcpp::QoS(rclcpp::QoSInitialization(qos_profile.history, 5), qos_profile);
61
62            vehicle_local_position_subscription_ = this->create_subscription<VehicleLocalPosition>(
63                "/fmu/out/vehicle_local_position", qos,
64                [this](const VehicleLocalPosition::UniquePtr msg) {
65                    current_x_ = msg->x;
66                    current_y_ = msg->y;
67                    current_z_ = msg->z;
68                    last_position_time_ = this->now();  // Update the last received message time
69
70                    if (logging_active_ && logfile_)
71                    {
72                        double timestamp = msg->timestamp / 1e6; // Convert from microseconds to
      seconds
73                        *logfile_ << std::fixed << std::setprecision(6) << timestamp << "," <<
      current_x_ << "," << current_y_ << "," << current_z_ << std::endl;
74                    }
75                });
76
77            auto timer_callback = [this]() -> void {
78                // Arm and switch to Offboard mode after 10 setpoints
79                if (offboard_setpoint_counter_ == 10)
80                {
81                    this->publish_vehicle_command(VehicleCommand::VEHICLE_CMD_DO_SET_MODE, 1, 6);
82                    this->arm();
83                }
84
85                // Ensure offboard control mode is paired with trajectory setpoint
86                publish_offboard_control_mode();
87                publish_trajectory_setpoint();
88
89                // Increment the setpoint counter
90                if (offboard_setpoint_counter_ < 11)
91                {
92                    offboard_setpoint_counter_++;
93                }
94
95                auto now = this->now();
96                auto time_since_last_position = now - last_position_time_;
97
98                // Check if we haven't received position messages for more than 3 seconds
99                if (time_since_last_position > 3s)
100               {
101                   if (logging_active_)
102                   {
103                       logging_active_ = false;
104                       RCLCPP_WARN(this->get_logger(), "No position messages received for more
      than 3 seconds. Stopping logging.");
105                   }
106               }
107               else if (!logging_active_)
108               {
109                   logging_active_ = true;
110                   RCLCPP_INFO(this->get_logger(), "Position messages received again. Restarting
      logging.");
```

```
111                 std::this_thread::sleep_for(std::chrono::seconds(2)); //wait for some time
    before sending arm command
112                 this->arm();
113                 start_new_logfile();
114                 last_log_time_ = now.seconds();  // Restart the 60-second timer
115             }
116 
117             // Change control frequency every log_interval_ seconds if logging is active
118             if (logging_active_ && (now.seconds() - last_log_time_ >= log_interval_))
119             {
120                 last_log_time_ = now.seconds();
121                 change_control_frequency();
122                 start_new_logfile();
123             }
124         };
125 
126         timer_ = this->create_wall_timer(std::chrono::duration<double>(1.0 / control_frequency_
    ), timer_callback);
127 
128         start_new_logfile();
129         RCLCPP_INFO(this->get_logger(), "Control frequency: %f Hz", control_frequency_);
130     }
131 
132     ~OffboardControl()
133     {
134         if (logfile_)
135         {
136             logfile_->close();
137             delete logfile_;
138         }
139     }
140 
141     void arm();
142     void disarm();
143 
144 private:
145     rclcpp::TimerBase::SharedPtr timer_;
146 
147     rclcpp::Publisher<OffboardControlMode>::SharedPtr offboard_control_mode_publisher_;
148     rclcpp::Publisher<TrajectorySetpoint>::SharedPtr trajectory_setpoint_publisher_;
149     rclcpp::Publisher<VehicleCommand>::SharedPtr vehicle_command_publisher_;
150     rclcpp::Subscription<VehicleLocalPosition>::SharedPtr vehicle_local_position_subscription_;
151 
152     uint64_t offboard_setpoint_counter_;
153     double radius_;
154     double altitude_;
155     double angular_velocity_;
156     double control_frequency_;
157     double min_control_frequency_;
158     double max_control_frequency_;
159     double control_frequency_step_;
160     double log_interval_;
161     double last_log_time_;
162     rclcpp::Time last_position_time_;
163 
164     float current_x_;
165     float current_y_;
166     float current_z_;
167     double prev_error_x_;
168     double prev_error_y_;
169     double integral_error_x_;
170     double integral_error_y_;
171 
172     std::ofstream *logfile_;
173     bool logging_active_;
174     int logfile_index_;
175 
176     void publish_offboard_control_mode();
177     void publish_trajectory_setpoint();
178     void publish_vehicle_command(uint16_t command, float param1 = 0.0, float param2 = 0.0);
179     void change_control_frequency();
180     void start_new_logfile();
181 };
```

```cpp

/**
 * @brief Send a command to Arm the vehicle.
 */
void OffboardControl::arm()
{
    publish_vehicle_command(VehicleCommand::VEHICLE_CMD_COMPONENT_ARM_DISARM, 1.0);
    RCLCPP_INFO(this->get_logger(), "Arm command sent");
}

/**
 * @brief Send a command to Disarm the vehicle.
 */
void OffboardControl::disarm()
{
    publish_vehicle_command(VehicleCommand::VEHICLE_CMD_COMPONENT_ARM_DISARM, 0.0);
    RCLCPP_INFO(this->get_logger(), "Disarm command sent");
}

/**
 * @brief Publish the offboard control mode.
 *        For this example, only velocity control is active.
 */
void OffboardControl::publish_offboard_control_mode()
{
    OffboardControlMode msg{};
    msg.position = false;
    msg.velocity = true;
    msg.acceleration = false;
    msg.attitude = false;
    msg.body_rate = false;
    msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
    offboard_control_mode_publisher_->publish(msg);
}

/**
 * @brief Publish a trajectory setpoint.
 *        For this example, it sends a trajectory setpoint to make the
 *        vehicle follow a circular path at a given altitude using velocity setpoints.
 */
void OffboardControl::publish_trajectory_setpoint()
{
    TrajectorySetpoint msg{};
    auto now = this->now();
    double t = now.seconds();

    // Desired position in the circle
    double x_d = radius_ * cos(angular_velocity_ * t);
    double y_d = radius_ * sin(angular_velocity_ * t);

    // PID control gains
    double k_p = 2.5; // Proportional gain
    double k_i = 1.7; // Integral gain
    double k_d = 0.8; // Derivative gain

    // Errors
    double error_x = x_d - current_x_;
    double error_y = y_d - current_y_;

    // Integrate errors
    integral_error_x_ += error_x / control_frequency_;
    integral_error_y_ += error_y / control_frequency_;

    // Saturate integral error to avoid windup
    integral_error_x_ = std::clamp(integral_error_x_, -1.0, 1.0);
    integral_error_y_ = std::clamp(integral_error_y_, -1.0, 1.0);

    // Derivative errors
    double derivative_x = (error_x - prev_error_x_) * control_frequency_;
    double derivative_y = (error_y - prev_error_y_) * control_frequency_;

    // Velocity setpoints with feedback to correct errors
```

```
254     float vx = static_cast<float>(-radius_ * angular_velocity_ * sin(angular_velocity_ * t) +
        k_p * error_x + k_i * integral_error_x_ + k_d * derivative_x);
255     float vy = static_cast<float>(radius_ * angular_velocity_ * cos(angular_velocity_ * t) +
        k_p * error_y + k_i * integral_error_y_ + k_d * derivative_y);
256     float vz = 2 * (-altitude_ - current_z_); // Maintain constant altitude
257
258     msg.velocity = {vx, vy, vz};
259     msg.position = {std::numeric_limits<float>::quiet_NaN(), std::numeric_limits<float>::
        quiet_NaN(), static_cast<float>(-altitude_)}; // Ensure position is ignored
260     msg.yaw = std::numeric_limits<float>::quiet_NaN(); // Ensure yaw is ignored
261     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
262     trajectory_setpoint_publisher_->publish(msg);
263
264     prev_error_x_ = error_x;
265     prev_error_y_ = error_y;
266 }
267
268 /**
269  * @brief Publish vehicle commands.
270  * @param command   Command code (matches VehicleCommand and MAVLink MAV_CMD codes)
271  * @param param1    Command parameter 1
272  * @param param2    Command parameter 2
273  */
274 void OffboardControl::publish_vehicle_command(uint16_t command, float param1, float param2)
275 {
276     VehicleCommand msg{};
277     msg.param1 = param1;
278     msg.param2 = param2;
279     msg.command = command;
280     msg.target_system = 1;
281     msg.target_component = 1;
282     msg.source_system = 1;
283     msg.source_component = 1;
284     msg.from_external = true;
285     msg.timestamp = this->get_clock()->now().nanoseconds() / 1000;
286     vehicle_command_publisher_->publish(msg);
287 }
288
289 void OffboardControl::change_control_frequency()
290 {
291     control_frequency_ += control_frequency_step_;
292     if (control_frequency_ > max_control_frequency_)
293     {
294         RCLCPP_INFO(this->get_logger(), "Experiment finished.");
295         control_frequency_ = max_control_frequency_;
296         rclcpp::shutdown();
297     }
298     else
299     {
300         logfile_index_++; // Increment the log file index only when the control frequency
        changes
301     }
302     timer_->cancel();
303     timer_ = this->create_wall_timer(std::chrono::duration<double>(1.0 / control_frequency_), [
        this]() -> void {
304         // Arm and switch to Offboard mode after 10 setpoints
305         if (offboard_setpoint_counter_ == 10)
306         {
307             this->publish_vehicle_command(VehicleCommand::VEHICLE_CMD_DO_SET_MODE, 1, 6);
308             this->arm();
309         }
310
311         // Ensure offboard control mode is paired with trajectory setpoint
312         publish_offboard_control_mode();
313         publish_trajectory_setpoint();
314
315         // Increment the setpoint counter
316         if (offboard_setpoint_counter_ < 11)
317         {
318             offboard_setpoint_counter_++;
319         }
320
321         auto now = this->now();
```

```
322         auto time_since_last_position = now - last_position_time_;
323
324         // Check if we haven't received position messages for more than 3 seconds
325         if (time_since_last_position > 3s)
326         {
327             if (logging_active_)
328             {
329                 logging_active_ = false;
330                 RCLCPP_WARN(this->get_logger(), "No position messages received for more than 3
    seconds. Stopping logging.");
331             }
332         }
333         else if (!logging_active_)
334         {
335             logging_active_ = true;
336             RCLCPP_INFO(this->get_logger(), "Position messages received again. Restarting
    logging.");
337             std::this_thread::sleep_for(std::chrono::seconds(2)); //wait for some time before
    sending arm command
338             this->arm();
339             start_new_logfile();
340             last_log_time_ = now.seconds();  // Restart the 60-second timer
341         }
342
343         // Change control frequency every log_interval_ seconds if logging is active
344         if (logging_active_ && (now.seconds() - last_log_time_ >= log_interval_))
345         {
346             last_log_time_ = now.seconds();
347             change_control_frequency();
348             start_new_logfile();
349         }
350     });
351     RCLCPP_INFO(this->get_logger(), "New control frequency: %f Hz", control_frequency_);
352 }
353
354 void OffboardControl::start_new_logfile()
355 {
356     if (logfile_)
357     {
358         logfile_->close();
359         delete logfile_;
360     }
361
362     std::string filename = "traj_f_" + std::to_string(logfile_index_) + ".txt";
363     logfile_ = new std::ofstream(filename);
364     if (logfile_->is_open())
365     {
366         //*logfile_ << "f = " << control_frequency_ << " Hz" << std::endl;
367         *logfile_ << "w = " << angular_velocity_ << " rad/s , " << "pub_freq = " <<
    control_frequency_ << " Hz" << std::endl;
368         *logfile_ << "timestamp [s] , x [m] , y [m] , z [m]" << std::endl;
369         RCLCPP_INFO(this->get_logger(), "Logging trajectory to '%s'", filename.c_str());
370     }
371     else
372     {
373         RCLCPP_ERROR(this->get_logger(), "Failed to open log file: %s", filename.c_str());
374     }
375 }
376
377 int main(int argc, char *argv[])
378 {
379     std::cout << "Starting offboard control node..." << std::endl;
380     setvbuf(stdout, NULL, _IONBF, BUFSIZ);
381     rclcpp::init(argc, argv);
382     rclcpp::spin(std::make_shared<OffboardControl>());
383
384     rclcpp::shutdown();
385     return 0;
386 }
```

Appendix B

ROS2 Talker/Listener nodes

B.1 Talker

content/chapters/6/docs/rtt/talker_w_exp.cpp

```cpp
#include <chrono>
#include <memory>
#include <fstream>
#include <iomanip>
#include <sstream>
#include <map>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

using namespace std::chrono_literals;

class Talker : public rclcpp::Node
{
public:
    Talker(const std::string &name, std::chrono::milliseconds publish_interval)
        : Node("talker"), talker_name_(name), file_index_(0), time_since_last_file_(0s),
    last_response_time_(this->now()), connection_active_(true), publish_interval_(
    publish_interval)
    {
        publisher_ = this->create_publisher<std_msgs::msg::String>("topic", 10);
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "response_topic", 10, std::bind(&Talker::listener_callback, this, std::placeholders
    ::_1));
        timer_ = this->create_wall_timer(
            publish_interval_, std::bind(&Talker::timer_callback, this));

        // Initialize log files for known listeners
        create_log_files();
    }

    ~Talker()
    {
        for (auto &file : log_files_)
        {
            if (file.second.is_open())
            {
                file.second.close();
            }
        }
    }

private:
    void timer_callback()
    {
        auto now = this->now();
```

```
43          std::ostringstream message_data;
44          message_data << now.nanoseconds() << " " << talker_name_;
45
46          auto message = std_msgs::msg::String();
47          message.data = message_data.str();
48          start_time_ = now;
49          publisher_->publish(message);
50
51          time_since_last_file_ += publish_interval_;
52
53          if ((now - last_response_time_).seconds() > 3.0)
54          {
55              if (connection_active_)
56              {
57                  RCLCPP_WARN(this->get_logger(), "Connection lost. Stopping logging.");
58                  stop_logging();
59                  connection_active_ = false;
60              }
61          }
62          else
63          {
64              if (!connection_active_)
65              {
66                  RCLCPP_INFO(this->get_logger(), "Connection resumed. Restarting logging.");
67                  start_new_logfile();
68                  connection_active_ = true;
69                  time_since_last_file_ = 0ms;
70              }
71          }
72
73          if (connection_active_ && time_since_last_file_ >= 60s)
74          {
75              file_index_++;
76              create_log_files();
77              time_since_last_file_ = 0ms;
78          }
79      }
80
81      void listener_callback(const std_msgs::msg::String::SharedPtr msg)
82      {
83          last_response_time_ = this->now();
84          auto now = this->now();
85
86          // Extract timestamp and listener name from the message
87          std::istringstream ss(msg->data);
88          int64_t sent_time_ns;
89          std::string listener_name;
90          ss >> sent_time_ns >> listener_name;
91
92          auto round_trip_time = (now.nanoseconds() - sent_time_ns) / 1e6; // convert to
        milliseconds
93
94          auto sys_now = std::chrono::system_clock::now();
95          auto duration = sys_now.time_since_epoch();
96          auto millis = std::chrono::duration_cast<std::chrono::milliseconds>(duration).count();
97          double timestamp = millis / 1000.0; // Convert milliseconds to seconds
98
99          if (log_files_.find(listener_name) != log_files_.end() && log_files_[listener_name].
        is_open())
100         {
101             log_files_[listener_name] << std::fixed << std::setprecision(3)
102                                       << "Timestamp: " << timestamp << " s, "
103                                       << "Round-trip latency: " << round_trip_time << " ms" <<
        std::endl;
104         }
105         else
106         {
107             RCLCPP_ERROR(this->get_logger(), "Log file for listener '%s' not open",
        listener_name.c_str());
108         }
109     }
110
111     void create_log_files()
```

```
112     {
113         for (auto &file : log_files_)
114         {
115             if (file.second.is_open())
116             {
117                 file.second.close();
118             }
119         }
120 
121         for (const auto &listener : known_listeners_)
122         {
123             std::ostringstream filename;
124             filename << "RTT_" << talker_name_ << "_to_" << listener << "_" << file_index_ << "
    .txt";
125             log_files_[listener].open(filename.str(), std::ios::out | std::ios::trunc);
126 
127             if (!log_files_[listener].is_open())
128             {
129                 RCLCPP_ERROR(this->get_logger(), "Failed to open log file for listener '%s'",
    listener.c_str());
130             }
131             else
132             {
133                 RCLCPP_INFO(this->get_logger(), "Logging latencies to '%s'", filename.str().
    c_str());
134             }
135         }
136     }
137 
138     void stop_logging()
139     {
140         for (auto &file : log_files_)
141         {
142             if (file.second.is_open())
143             {
144                 file.second.close();
145             }
146         }
147     }
148 
149     void start_new_logfile()
150     {
151         create_log_files();
152     }
153 
154     rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
155     rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
156     rclcpp::TimerBase::SharedPtr timer_;
157     rclcpp::Time start_time_;
158     rclcpp::Time last_response_time_;
159     std::string talker_name_;
160     std::map<std::string, std::ofstream> log_files_;
161     std::vector<std::string> known_listeners_ = {"sim", "UE"};
162     int file_index_;
163     std::chrono::milliseconds time_since_last_file_;
164     std::chrono::milliseconds publish_interval_;
165     bool connection_active_;
166 };
167 
168 int main(int argc, char *argv[])
169 {
170     rclcpp::init(argc, argv);
171     auto node = std::make_shared<Talker>("gNB", 100ms); // Modify the second argument to change
      the publishing frequency
172     rclcpp::spin(node);
173     rclcpp::shutdown();
174     return 0;
175 }
```

B.2 Listener

content/chapters/6/docs/rtt/listener.cpp

```cpp
#include <memory>
#include <sstream>
#include "rclcpp/rclcpp.hpp"
#include "std_msgs/msg/string.hpp"

class Listener : public rclcpp::Node
{
public:
    Listener(const std::string &name)
        : Node(name), name_(name)
    {
        publisher_ = this->create_publisher<std_msgs::msg::String>("response_topic", 10);
        subscription_ = this->create_subscription<std_msgs::msg::String>(
            "topic", 10, std::bind(&Listener::listener_callback, this, std::placeholders::_1));
    }

private:
    void listener_callback(const std_msgs::msg::String::SharedPtr msg)
    {
        RCLCPP_INFO(this->get_logger(), "Received message: '%s'", msg->data.c_str());

        // Extract the timestamp from the received message
        std::istringstream iss(msg->data);
        int64_t timestamp;
        std::string talker_name;
        iss >> timestamp >> talker_name;

        // Prepare the response message
        auto response_msg = std_msgs::msg::String();
        response_msg.data = std::to_string(timestamp) + " " + name_;
        publisher_->publish(response_msg);
        RCLCPP_INFO(this->get_logger(), "Sent response: '%s'", response_msg.data.c_str());
    }

    std::string name_;
    rclcpp::Publisher<std_msgs::msg::String>::SharedPtr publisher_;
    rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subscription_;
};

int main(int argc, char *argv[])
{
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<Listener>("sim"));
    rclcpp::shutdown();
    return 0;
}
```

Bibliography

[1] *5G system overview*. URL: https://www.3gpp.org/technologies/5g-system-overview (cit. on p. 9).

[2] *System architecture for the 5G System (5GS) (Release 18) - V18.4.0*. Dec. 2023. URL: https://www.3gpp.org/ftp/Specs/archive/23_series/23.501/23501-i40.zip (cit. on p. 11).

[3] Carla Fabiana Chiasserini. «Spread Spectrum Techniques». Course: Mobile and Sensor Networks (2023) - Polytechnic Of Turin (cit. on p. 12).

[4] Roberto Fantini (TIM). «Workshop: Evolution from 4G to 5G». Course: 5G and next-generation mobile computing (2023) - Polytechnic Of Turin (cit. on pp. 12, 13).

[5] H. Dinh et al. «A survey of mobile cloud computing: architecture, applications, and approaches». In: (). DOI: 10.1002/wcm.1203 (cit. on p. 14).

[6] *What Is Edge Computing? 8 Examples and Architecture You Should Know*. URL: https://www.fsp-group.com/en/knowledge-app-42.html (cit. on p. 14).

[7] Wu Hassan Yau. «Edge Computing in 5G: A Review». In: (Aug. 2019). DOI: 10.1109/ACCESS.2019.2938534 (cit. on p. 15).

[8] *Mobile Edge Computing (MEC); Framework and Reference Architecture*. Group Specification. 2016 (cit. on pp. 15, 16).

[9] *OpenAirInterface: A Flexible Platform for 5G Research*. URL: https://mosaic5g.io/resources/mosaic5g_oai.pdf (cit. on p. 17).

[10] Luis Pereira. *OAI 5G NR SA tutorial with OAI nrUE*. URL: https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/doc/NR_SA_Tutorial_OAI_nrUE.md (cit. on p. 18).

[11] *Creating a Software Radio Spectrum Analyzer*. URL: https://wiki.gnuradio.org/index.php/Guided_Tutorial_Hardware_Considerations (cit. on p. 19).

[12] Sagar Arora. *OpenAirInterface Multi-access Edge Computing Platform Blueprint*. URL: https://gitlab.eurecom.fr/oai/orchestration/blueprints/-/blob/master/mep/README.md (cit. on pp. 22, 25).

[13] *Signal quality [LTE/5G] - LTE and 5G signal quality parameters*. URL: https://support.zyxel.eu/hc/en-us/articles/360005188999-Signal-quality-LTE-5G-LTE-and-5G-signal-quality-parameters (cit. on p. 29).

[14] *PX4 Software Overview*. URL: https://px4.io/software/software-overview/ (cit. on p. 31).

[15] *PX4 Basic Concepts*. URL: https://docs.px4.io/main/en/getting_started/px4_basic_concepts.html (cit. on p. 31).

[16] *PX4 Architectural Overview*. URL: https://docs.px4.io/main/en/concept/architecture.html (cit. on pp. 31–33, 35).

[17] *PX4 Architectural Overview*. URL: https://docs.px4.io/main/en/concept/px4_systems_architecture.html (cit. on p. 31).

[18] *Reactive Manifesto*. URL: https://www.reactivemanifesto.org/ (cit. on p. 32).

[19] *Reactive Manifesto*. URL: https://docs.px4.io/main/en/middleware/uorb.html (cit. on p. 34).

[20] *Simulation*. URL: https://docs.px4.io/main/en/simulation/ (cit. on p. 35).

[21] *Flight Modes*. URL: https://docs.px4.io/main/en/flight_modes_mc/ (cit. on p. 37).

[22] *Offboard Mode*. URL: https://docs.px4.io/main/en/flight_modes/offboard.html (cit. on p. 38).

[23] *ROS2*. URL: https://docs.px4.io/main/en/ros2/ (cit. on p. 38).

[24] *ROS2 User Guide*. URL: https://docs.px4.io/main/en/ros2/user_guide (cit. on p. 39).

[25] *Publish–subscribe pattern*. URL: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern (cit. on p. 39).

[26] *ROS on DDS*. URL: https://design.ros2.org/articles/ros_on_dds.html (cit. on p. 40).

[27] *What is DDS?* URL: https://fast-dds.docs.eprosima.com/en/latest/fastdds/getting_started/definitions.html (cit. on p. 40).

[28] *Discovery*. URL: https://fast-dds.docs.eprosima.com/en/latest/fastdds/discovery/discovery.html#disc-phases (cit. on p. 41).

[29] A. M. Qasim N. H. Jawad. «5G-enabled UAVs for energy-efficient opportunistic networking». In: *Heliyon, 10(12)* (2024) (cit. on p. 41).

[30] Koumaras H.; Makropoulos G.; Batistatos M.; Kolometsos S.; Gogos A.; Xilouris G.; Sarlas A.; Kourtis M.-A. «5G-Enabled UAVs with Command and Control Software Component at the Edge for Supporting Energy Efficient Opportunistic Networks». In: *Energies 2021, 14, 1480* (2021) (cit. on p. 41).

[31] *ROS 2 Offboard Control Example*. URL: https://docs.px4.io/main/en/ros2/offboard_control.html (cit. on p. 43).

[32] *Zenoh*. URL: https://github.com/eclipse-zenoh/zenoh (cit. on p. 45).

[33] *ROS 2 Alternative middleware report*. URL: https://discourse.ros.org/t/ros-2-alternative-middleware-report/33771 (cit. on p. 45).

[34] *A Zenoh bridge for ROS 2 over DDS*. URL: https://github.com/eclipse-zenoh/zenoh-plugin-ros2dds?tab=readme-ov-file (cit. on p. 45).

[35] *Zenoh Deployment*. URL: https://zenoh.io/docs/getting-started/deployment/#zenoh-router (cit. on p. 45).

[36] *A Zenoh bridge for ROS 2 over DDS*. URL: https://zenoh.io/blog/2021-04-28-ros2-integration/ (cit. on p. 45).

[37] *ROS 2 Offboard Control Example*. URL: https://docs.px4.io/main/en/ros/ros2_offboard_control.html (cit. on p. 46).

[38] *E2AP readme*. URL: https://gitlab.eurecom.fr/oai/openairinterface5g/-/blob/develop/openair2/E2AP/README.md (cit. on p. 54).