Contents

1 Introduction **3**
1.1 ROS 2 . . . 3
1.1.1 Node . . . 4
1.1.2 Topics . . . 4
1.1.3 Service . . . 4
1.2 PX4 Autopilot . . . 5
1.3 ROS2 et PX4 Autopilot . . . 5

2 Installation et Setup **6**
2.1 Install Ubuntu 22.04 . . . 6
2.2 Install PX4 . . . 6
2.3 Intall ROS2 Humble . . . 6
2.3.1 Télécharger et construire l'espace de travail ROS2 . . . 6
2.4 Configuration Micro XRCE-DDS Agent & Client . . . 7
2.4.1 Configuration de l'Agent . . . 7
2.4.2 Configuration de Client . . . 7
2.5 Exécuter le programme de simulation . . . 8
2.6 Créer le modèle du signal . . . 9
2.6.1 La différence de phase . . . 10

Version	Date	Nom et Prénom	Note
v1.0	02/10/2023	Anh P.H.	Init
v1.1	17/10/2023	Anh P.H.	- Créer des signaux sonores d'explosion pour 5 drones - Retirer le paquet ros_bridge - N'utilisez pas d'émetteur microphone sur un gazebo car il ne peut détecter que si un signal est présent ou non.
-	-	-	-

1 Introduction

1.1 ROS 2

Quelques différences importantes entre ROS et ROS2		
Applications	**ROS**	**ROS2**
Plates-formes	Testé sur Ubuntu, maintenu sur d'autres versions de Linux ainsi que sur OS X	Actuellement testé et supporté sur Ubuntu, Xenial, OS X El Capitan ainsi que Windows 10
C++	C++03, n'utilise pas les capacités de C++11 dans ses API	Utilise principalement C+++11, commencez et prévoyez d'utiliser C++14 et C++17
Python	Cible Python 2	>= Python 3.5
Middleware	Format de sérialisation personnalisé (protocole de transport + mécanisme de découverte central)	Actuellement, toutes les implémentations de cette interface sont basées sur le standard DDS
Synchroniser la durée et les mesures de temps	La durée et les types de temps sont définis dans les bibliothèques clientes, ils sont codés en C++ et Python	Ces types sont définis comme des messages et sont donc cohérents d'un langage de programmation à l'autre
Composants avec cycle de vie	Chaque nœud a généralement sa propre fonction principale	Le cycle de vie peut être utilisé par des outils comme roslaunch pour démarrer un système composé de nombreux composants de manière déterministe
Modèle multi-threads	Le développeur ne peut choisir qu'entre une exécution mono-thread ou multi-thread	Des modèles d'exécution plus fins sont disponibles et des exécuteurs personnalisés peuvent être facilement implémentés.
Noeuds multiples	Il n'est pas possible de créer plus d'un nœud dans un processus	Il est possible de créer plusieurs nœuds dans un processus
roslaunch	Les fichiers de roslaunch sont définis en XML avec des capacités très limitées.	Les fichiers de lancement sont écrits en Python ce qui permet d'utiliser des logiques plus complexes comme les conditionnels

Par ailleurs, il existe des outils qui permettent de simuler des systèmes ROS 2 arbitraires et de mesurer ensuite leurs performances. `https://github.com/irobot-ros/ros2-performance` et [1].

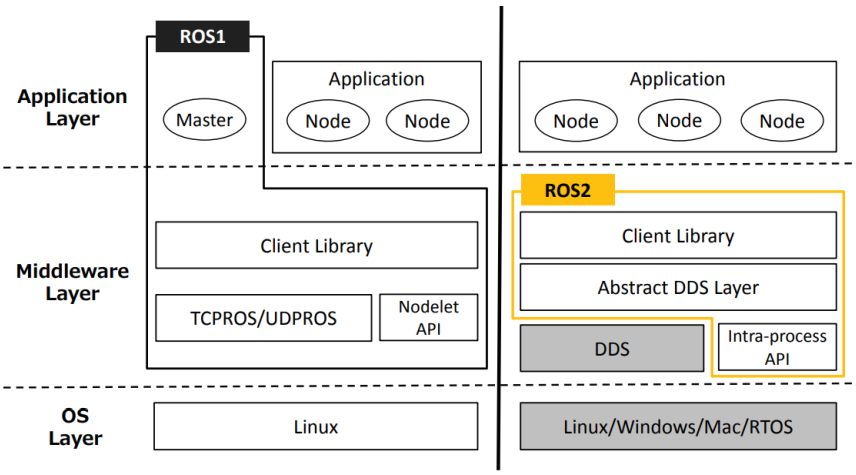

Figure 1: Architecture ROS1/ROS2

Dans la figure 1, l'implémentation de ROS1 comprend le système de communication, TCP/UDP ROS.

Cette communication nécessite un processus maître. C'est l'inconvénient de son application aux systèmes distribués. En revanche, ROS2 s'appuie sur DDS et contient une couche d'abstraction DDS. Les utilisateurs n'ont pas besoin de connaître les API de DDS grâce à cette couche d'abstraction. Cette couche permet à ROS2 d'avoir des configurations de haut niveau et optimise l'utilisation de DDS. En outre, grâce à l'utilisation de DDS, ROS2 n'a pas besoin d'un processus maître.

1.1.1 Node

Dans ROS2, chaque nœud doit être responsable d'un seul module (par exemple, un nœud pour contrôler les moteurs de roue, un autre pour contrôler un télémètre laser, etc.) Chaque nœud peut communiquer avec les autres nœuds par différentes méthodes comme Topic, Service...(voir Figure 2).

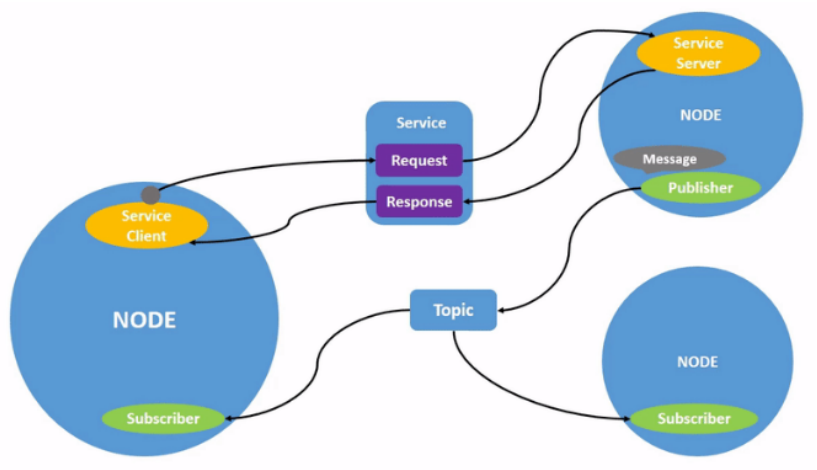

Figure 2: ROS2 Node

1.1.2 Topics

ROS 2 décompose les systèmes complexes en de nombreux nœuds modulaires. Les Topics (voir Figure 3) sont un élément essentiel du graphique ROS qui agit comme un bus permettant aux nœuds d'échanger des messages. Un nœud peut publier des données sur n'importe quel nombre de sujets et avoir simultanément des abonnements à n'importe quel nombre de sujets. Les Topics sont l'un des principaux moyens par lesquels les données sont déplacées entre les nœuds et donc entre les différentes parties du système.

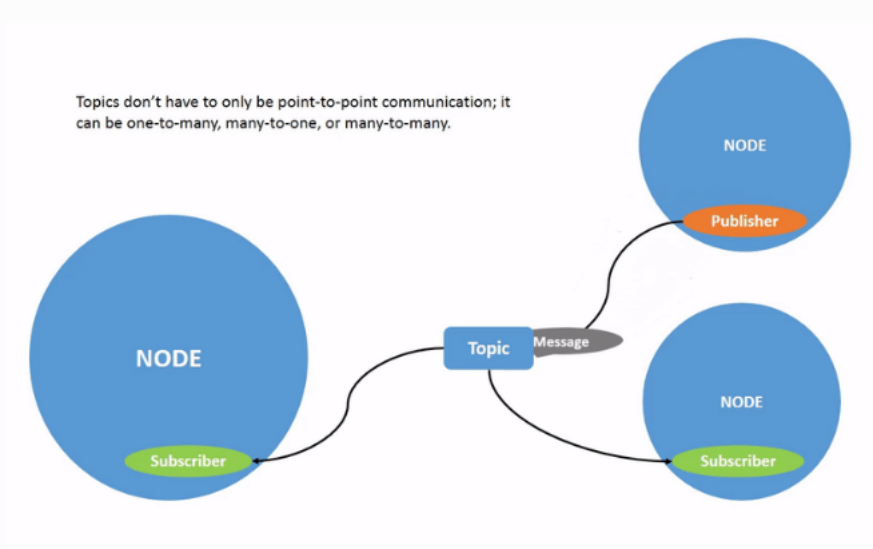

Figure 3: Topics

1.1.3 Service

Comme Topics, les services sont également une méthode de communication entre les nœuds de ROS2. Les Topics utilisent le modèle publisher-subscriber. En revanche, les services utilisent un modèle appel-réponse.

Nous pouvons abonner un nœud à un sujet pour recevoir des informations particulières avec des mises à jour continues. En même temps, un Service ne fournit des données que lorsqu'il est appelé explicitement par un Client (voir la figure 4).

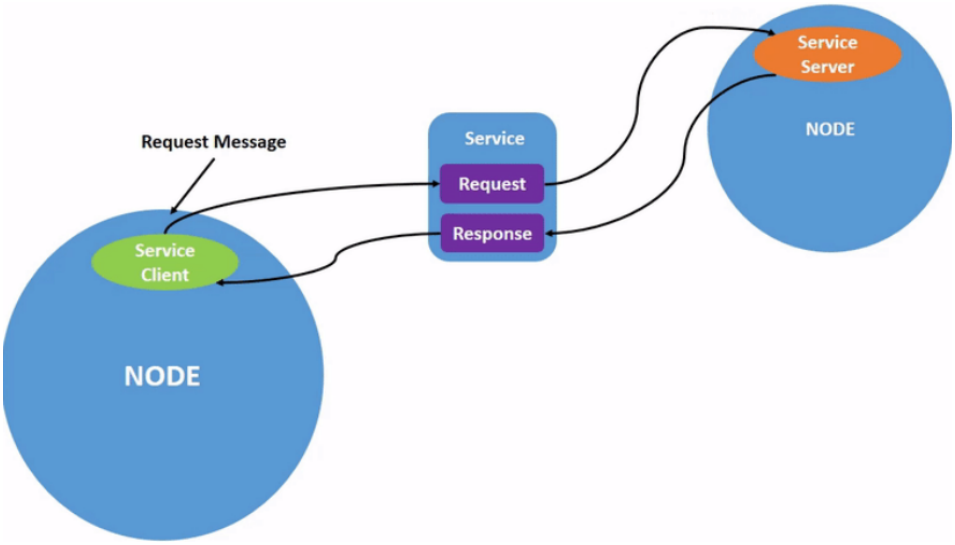

Figure 4: ROS2 Service

1.2 PX4 Autopilot

PX4 est un logiciel de pilotage open source pour les drones et autres véhicules sans pilote. Le projet fournit un ensemble d'outils flexibles permettant aux développeurs de drones de partager des technologies afin de créer des solutions sur mesure pour les applications de drones. PX4 fournit un standard pour le support matériel et la plateforme logicielle des drones, permettant à un écosystème de construire et d'entretenir du matériel et des logiciels de manière évolutive (voir [1]).

PX4 est un élément central d'une plateforme de drone plus large qui comprend la station au sol QGroundControl, le matériel Pixhawk et le MAVSDK pour l'intégration avec des ordinateurs complémentaires, des caméras et d'autres matériels utilisant le protocole MAVLink. PX4 est soutenu par le projet Dronecode. PX4 a été initialement conçu pour fonctionner sur les contrôleurs de la série Pixhawk, mais peut maintenant fonctionner sur des ordinateurs Linux et d'autres matériels.

PX4 supporte à la simulation Software In the Loop (SITL), où la plateforme de vol fonctionne sur un ordinateur (soit le même ordinateur, soit un autre ordinateur sur le même réseau) et la simulation Hardware In the Loop (HITL) en utilisant un firmware de simulation sur une vraie carte de contrôleur de vol (voir [2]).

1.3 ROS2 et PX4 Autopilot

L'architecture ROS 2-PX4 fournit une intégration profonde entre ROS 2 et PX4, permettant aux Subscribers de ROS 2 ou aux nœuds de Publisher de s'interfacer directement avec les Topics uORB de PX4 (voir 5) (ref. [3]).

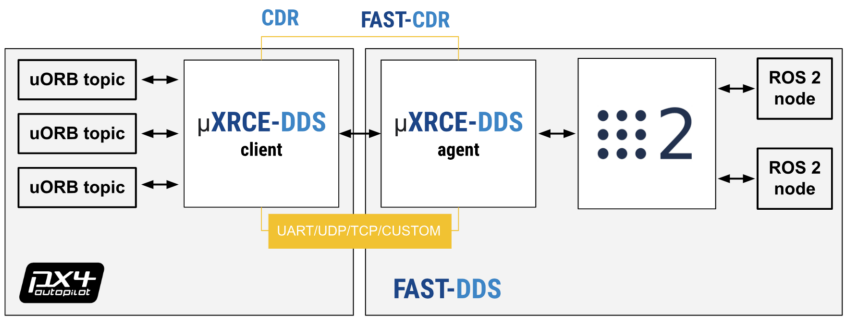

Figure 5: Architecture ROS2 et PX4 Autopilot

[1] https://docs.px4.io/main/en/
[2] https://dev.px4.io/v1.10_noredirect/en/simulation/
[3] https://docs.px4.io/main/en/ros/ros2_comm.html

Le pipeline d'application pour ROS 2 est très complet, grâce à l'utilisation de l'intergiciel de communication uXRCE-DDS.

L'intergiciel uXRCE-DDS consiste en un client fonctionnant sur PX4 et un agent fonctionnant sur l'ordinateur complémentaire, avec un échange de données bidirectionnel entre eux sur un lien sériel, UDP, TCP ou personnalisé. L'agent agit comme un proxy pour le client qui publie et s'abonne à des Topics dans l'espace de données DDS global.

Le client PX4 uxrce_dds_client est généré au moment de la construction et inclus par défaut dans le micrologiciel PX4. Il comprend à la fois le code client micro XRCE-DDS "générique" et le code de traduction spécifique au PX4 qu'il utilise pour publier vers/depuis les Topics uORB.

Les applications ROS 2 doivent être construites dans un espace de travail qui a les mêmes définitions de messages que celles utilisées pour créer le module client uXRCE-DDS dans le firmware PX4.

Il faut normalement démarrer le client et l'agent lorsqu'on utilise ROS 2. Il faut noter également que le client uXRCE-DDS est intégré par défaut dans le firmware mais qu'il n'est pas lancé automatiquement sauf dans le cas des simulateurs.

2 Installation et Setup

Les plateformes ROS2 supportées pour le développement de PX4 sont ROS2 "Humble" sur Ubuntu 22.04, et ROS 2 "Foxy" sur Ubuntu 20.04. (Ref. [4]). ROS 2 "Humble" est recommandé car il s'agit de la distribution ROS 2 LTS actuelle. ROS 2 "Foxy" a atteint sa fin de vie en mai 2023, mais est toujours stable et fonctionne avec PX4. **Ce document est destiné à être utilisé ROS2 Humble et Ubuntu 22.04**.

2.1 Install Ubuntu 22.04

Supposons que vous ayez installé Ubuntu 22.04 avec succès. Sinon, vous trouverez ci-dessous le lien de référence pour installer Ubuntu 22.04

`https://linuxgenie.net/how-to-download-and-install-ubuntu-22-04/.`

2.2 Install PX4

Installer la chaîne d'outils de développement PX4 afin d'utiliser le simulateur.

```
$ cd
$ git clone https://github.com/PX4/PX4-Autopilot.git --recursive
$ bash ./PX4-Autopilot/Tools/setup/ubuntu.sh
$ cd PX4-Autopilot/
$ make px4_sitl
```

2.3 Intall ROS2 Humble

```
$ sudo apt update && sudo apt install locales
$ sudo locale-gen en_US en_US.UTF-8
$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
$ export LANG=en_US.UTF-8
$ sudo apt install software-properties-common
$ sudo add-apt-repository universe
$ sudo apt update && sudo apt install curl -y
$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/
    keyrings/ros-archive-keyring.gpg
$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.
    gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" |
     sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
$ sudo apt update && sudo apt upgrade -y
$ sudo apt install ros-humble-desktop
$ sudo apt install ros-dev-tools
$ source /opt/ros/humble/setup.bash && echo "source /opt/ros/humble/setup.bash" >> .bashrc
```

2.3.1 Télécharger et construire l'espace de travail ROS2

Il s'agit d'un programme développé sur la plateforme ROS2 pour recevoir des signaux et envoyer des signaux de contrôle aux drones. Télécharger le fichier **SwarmZ_ROS2**.

[4] `https://docs.px4.io/main/en/ros/ros2_comm.html`

```
$ cd ..
$ git clone https://github.com/phamhoanganhbk/SwarmZ_ROS2.git
```

Pour construire le programme

```
$ cd SwarmZ_ROS2
$ source /opt/ros/humble/setup.bash
$ colcon build
```

Exécutez le programme de démonstration (il faut d'abord lancer des drones sur le Gazebo, voir section 2.5)

```
$ source install/local_setup.bash
$ ros2 run px4_ros_com offboard_control_px4_1
```

Dans ce cas, il y a déjà deux nœuds pour contrôler les Drones 1 et 2. Créez des nœuds similaires pour contrôler les Drones 3,4,5.

2.4 Configuration Micro XRCE-DDS Agent & Client

Pour que ROS 2 puisse communiquer avec PX4, le client uXRCE-DDS doit fonctionner sur PX4, connecté à un agent micro XRCE-DDS fonctionnant sur l'ordinateur.

2.4.1 Configuration de l'Agent

L'agent peut être installé sur l'ordinateur compagnon de plusieurs manières. Nous montrons ci-dessous comment construire l'agent "autonome" à partir des sources et se connecter à un client fonctionnant sur le simulateur PX4.

Pour configurer l'Agent :

```
$ git clone https://github.com/eProsima/Micro-XRCE-DDS-Agent.git
$ cd Micro-XRCE-DDS-Agent
$ mkdir build
$ cd build
$ cmake ..
$ make
$ sudo make install
$ sudo ldconfig /usr/local/lib/
```

Démarrer l'agent avec les paramètres de connexion au client uXRCE-DDS fonctionnant sur le simulateur:

```
$ MicroXRCEAgent udp4 -p 8888
```

L'agent est maintenant en cours d'exécution. Vous pouvez laisser l'agent fonctionner dans ce terminal! À noter qu'un seul agent est autorisé par canal de connexion.

2.4.2 Configuration de Client

Le simulateur PX4 démarre automatiquement le client uXRCE-DDS, en se connectant au port UDP 8888 sur l'hôte local.

Pour démarrer le simulateur (et le client) :

- Ouvrir un nouveau terminal dans la root du repo de PX4 Autopilot qui a été installé ci-dessus. Commencer une simulation PX4 Gazebo en utilisant :

```
    $ make px4_sitl gz_x500

```

L'agent et le client sont maintenant en cours d'exécution et devraient se connecter. Le terminal PX4 affiche la sortie de la console système NuttShell/PX4 pendant que PX4 démarre et s'exécute. Dès que l'agent se connecte, la sortie doit inclure des messages INFO indiquant la création de rédacteurs de données :

```
    ...
INFO  [uxrce_dds_client] synchronized with time offset 1675929429203524us
INFO  [uxrce_dds_client] successfully created rt/fmu/out/failsafe_flags data writer, topic id
    : 83
INFO  [uxrce_dds_client] successfully created rt/fmu/out/sensor_combined data writer, topic
    id: 168
```

```
INFO  [uxrce_dds_client] successfully created rt/fmu/out/timesync_status data writer, topic id: 188
...
```

Le terminal de l'agent micro XRCE-DDS devrait également commencer à afficher des données de sortie, car des Topics équivalentes sont créées dans le réseau DDS:

```
...
[1675929445.268957] info     | ProxyClient.cpp    | create_publisher         | publisher created      | client_key: 0x00000001, publisher_id: 0x0DA(3), participant_id: 0x001(1)
[1675929445.269521] info     | ProxyClient.cpp    | create_datawriter        | datawriter created     | client_key: 0x00000001, datawriter_id: 0x0DA(5), publisher_id: 0x0DA(3)
[1675929445.270412] info     | ProxyClient.cpp    | create_topic             | topic created          | client_key: 0x00000001, topic_id: 0x0DF(2), participant_id: 0x001(1)
...
```

Cette étape a permis d'installer avec succès l'agent et le client Micro XRCE-DDS.

2.5 Exécuter le programme de simulation

Lancer Drone 1 - 2 - 3 - 4 - 5 (Chaque drone tourne sur un terminal différent). Initialiser la position d'origine de tous les drones. Lancer 5 fois sur 5 terminaux avec 5 drones.

```
export PX4_HOME_LAT=43.13471
export PX4_HOME_LON=6.01507
export PX4_HOME_ALT=6
```

```
$ cd PX4-Autopilot
$ PX4_SYS_AUTOSTART=4001 PX4_GZ_MODEL_POSE="5,0" PX4_GZ_MODEL=x500 ./build/px4_sitl_default/bin/px4 -i 1
$ PX4_SYS_AUTOSTART=4001 PX4_GZ_MODEL_POSE="-5,0" PX4_GZ_MODEL=x500 ./build/px4_sitl_default/bin/px4 -i 2
$ PX4_SYS_AUTOSTART=4001 PX4_GZ_MODEL_POSE="0,5" PX4_GZ_MODEL=x500 ./build/px4_sitl_default/bin/px4 -i 3
$ PX4_SYS_AUTOSTART=4001 PX4_GZ_MODEL_POSE="0,-5" PX4_GZ_MODEL=x500 ./build/px4_sitl_default/bin/px4 -i 4
$ PX4_SYS_AUTOSTART=4001 PX4_GZ_MODEL_POSE="10,0" PX4_GZ_MODEL=x500 ./build/px4_sitl_default/bin/px4 -i 5
```

Lancer XRCE Agent

```
$ MicroXRCEAgent udp4 -p 8888
```

Lancer ROS2 pour contrôler Drone-1

```
$ cd SwarmZ_ROS2/
$ source install/local_setup.bash
$ ros2 run px4_ros_com offboard_control_px4_1
```

Nouveau terminal pour lancer ROS2 pour contrôler Drone-2

```
$ cd SwarmZ_ROS2/
$ source install/local_setup.bash
$ ros2 run px4_ros_com offboard_control_px4_2
```

Il a besoin de créer des packages plus similaires pour contrôler le drone 3, 4, 5.

Le programme de simulation de drones sur Gazebo est mis en œuvre (voir la figure 6).

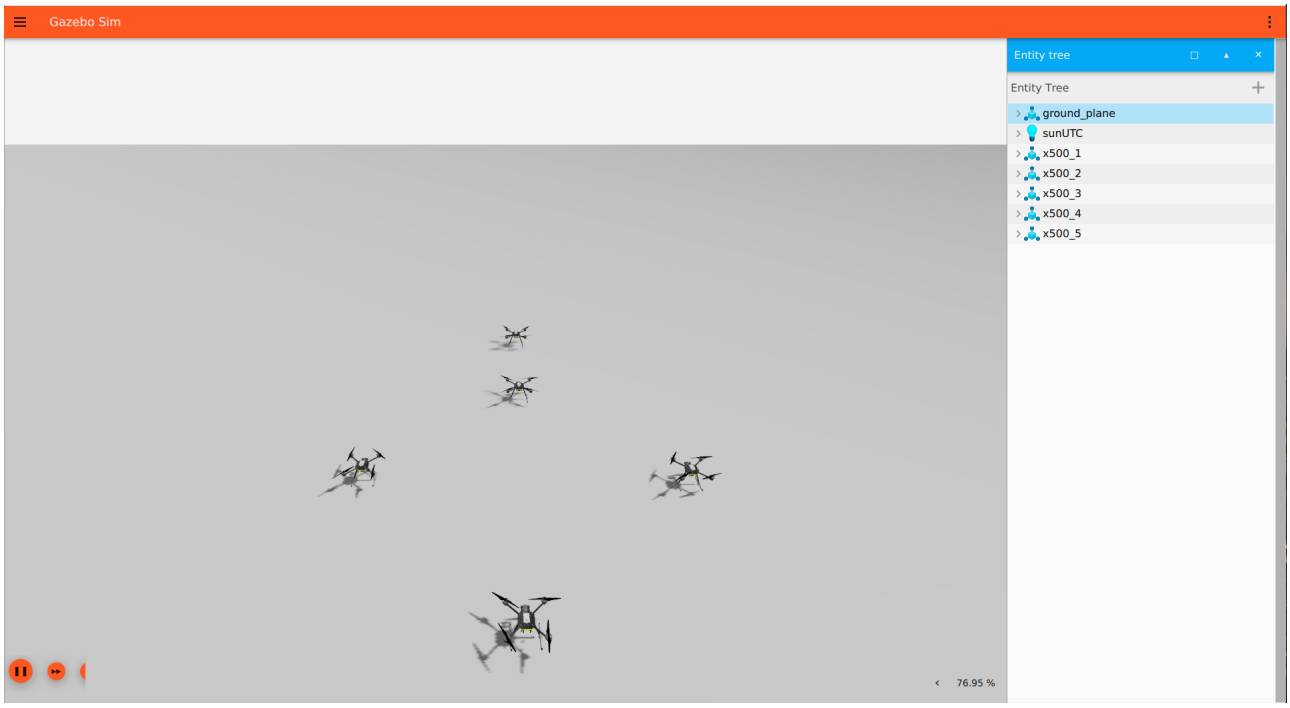

Figure 6: Simulation de drones sur Gazebo

2.6 Créer le modèle du signal

Le programme de simulation de signal est construit sur ROS2. Au cœur de ce programme se trouve un nœud qui a pour mission de générer et de gérer divers signaux. En fonction de la distance entre chaque source de signal et les drones, ce nœud effectue des calculs pour déterminer le déphasage $\Delta\phi(t)$ et l'atténuation de l'amplitude $\Delta A(t)$ de chaque signal jusqu'aux récepteurs situés sur les robots (voir la Figure 7). Ces valeurs seront calculées et envoyées via le réseau ROS2 toutes les 100 ms.

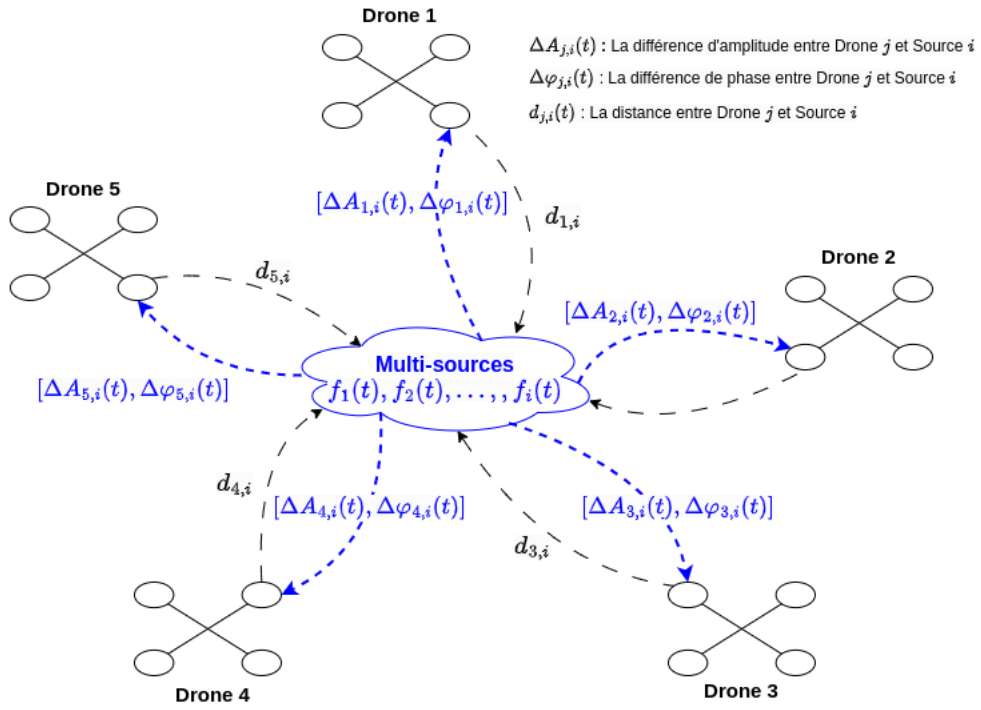

Figure 7: Un nœud gère et calcule l'atténuation du signal et la différence de phase

Les hypothèses sous-jacentes et la méthodologie de calcul sont expliquées en détail ci-dessous.

2.6.1 La différence de phase

Pour calculer la différence de phase entre un signal transmis et un signal reçu lorsque la distance entre les deux est connue, vous devez connaître la vitesse de propagation du signal dans le milieu spécifique. La différence de phase peut être calculée à l'aide de la formule suivante :

Différence de phase (en radians):

$$\Delta\varphi(t) = \frac{2 \times \pi \times d \times f}{c} \tag{1}$$

Où :
Distance d: Il s'agit de la distance entre la source du signal émetteur et le point de réception, généralement mesurée en mètres (m).

Vitesse de propagation du signal c: C'est la vitesse à laquelle le signal se déplace dans le milieu spécifique. Par exemple, dans le vide, la vitesse de la lumière est d'environ 3×10^8 mètres par seconde (m/s).

Fréquence du signal f: La fréquence du signal est la fréquence à laquelle le signal oscille, mesurée en hertz (Hz).

Supposons que les signaux sources S_α, S_β, S_γ, ... S_n soient représentés sous forme de composantes harmoniques f_i et $3f_i$ (où $i = \alpha, \beta, \gamma, ..., n$). L'amplitude A_{i1}, A_{i3}, la fréquence f_i et la phase initiales des signaux φ_{i1}, φ_{i3} sont prédéterminées (réglables).

Source S_α, Source S_β, Source S_γ, Source S_i :

$$S_\alpha = A_{\alpha1}\sin(2\pi f_\alpha t + \varphi_{\alpha1}) + A_{\alpha3}\sin(2\pi 3 f_\alpha t + \varphi_{\alpha3}) \tag{2}$$

$$S_\beta = A_{\beta1}\sin(2\pi f_\beta t + \varphi_{\beta1}) + A_{\beta3}\sin(2\pi 3 f_\beta t + \varphi_{\beta3}) \tag{3}$$

$$S_\gamma = A_{\gamma1}\sin(2\pi f_\gamma t + \varphi_{\gamma1}) + A_{\gamma3}\sin(2\pi 3 f_\gamma t + \varphi_{\gamma3}) \tag{4}$$

$$.... \tag{5}$$

$$S_n = A_{n1}\sin(2\pi f_n t + \varphi_{n1}) + A_{n3}\sin(2\pi 3 f_n t + \varphi_{n3}) \tag{6}$$

Le signal reçu à robot R_j

$$\begin{aligned}
S_{R_j} &= S_\alpha + S_\beta + S_\gamma + S_i \\
&= \frac{A_{\alpha1}}{d^2_{R_jS_\alpha}} \times \sin(2\pi f_\alpha t + \varphi_{\alpha1} + \Delta\varphi_{\alpha1}) + \frac{A_{\alpha3}}{d^2_{R_jS_\alpha}} \times \sin(2\pi 3 f_\alpha t + \varphi_{\alpha3} + \Delta\varphi_{\alpha3}) \\
&+ \frac{A_{\beta1}}{d^2_{R_jS_\beta}} \times \sin(2\pi f_\beta t + \varphi_{\beta1} + \Delta\varphi_{\beta1}) + \frac{A_{\beta3}}{d^2_{R_1S_\beta}} \times \sin(2\pi 3 f_\beta t + \varphi_{\beta3} + \Delta\varphi_{\beta3}) \\
&+ \frac{A_{\gamma1}}{d^2_{R_jS_\gamma}} \times \sin(2\pi f_\gamma t + \varphi_{\gamma1} + \Delta\varphi_{\gamma1}) + \frac{A_{\gamma3}}{d^2_{R_jS_\gamma}} \times \sin(2\pi 3 f_\gamma t + \varphi_{\gamma3} + \Delta\varphi_{\gamma3}) \\
&..... \\
&+ \frac{A_{n1}}{d^2_{R_jS_n}} \times \sin(2\pi f_n t + \varphi_{n1} + \Delta\varphi_{n1}) + \frac{A_{n3}}{d^2_{R_jS_n}} \times \sin(2\pi 3 f_n t + \varphi_{n3} + \Delta\varphi_{n3})
\end{aligned} \tag{7}$$

Avec

$$\Delta\varphi_{ik}(t) = \frac{2 \times \pi \times d_{R_1S_i} \times f}{c}$$

Où $i = \alpha, \beta, \gamma, ..., n$ et $k = 1, 3$
$d_{R_jS_i}$ est la distance entre le signal émis et le robot j
f est le fréquence du signal (Hz)
c est la vitesse de propagation du signal (m/s)
Toutes les 100 ms, on calcule S_{R_j} et les partagé sur le réseau ROS2

Un exemple de définition des paramètres du signal

```
#define SPEED_OF_LIGHT 299792458 // m/s
#define PI 3.1415

#define AMPLITUDE_SOURCE_R1_1 1
#define AMPLITUDE_SOURCE_R1_3 1
#define FREQUENCY_SOURCE_R1 100 // hz
#define PHASE_SOURCE_R1_1 0
#define PHASE_SOURCE_R1_3 0

#define AMPLITUDE_SOURCE_R2_1 1
#define AMPLITUDE_SOURCE_R2_3 1
#define FREQUENCY_SOURCE_R2 100 // hz
#define PHASE_SOURCE_R2_1 0
#define PHASE_SOURCE_R2_3 0

#define AMPLITUDE_SOURCE_R3_1 1
#define AMPLITUDE_SOURCE_R3_2 1
#define FREQUENCY_SOURCE_R3 100 // hz
#define PHASE_SOURCE_R3_1 0
#define PHASE_SOURCE_R3_3 0

#define AMPLITUDE_SOURCE_R4_1 1
#define AMPLITUDE_SOURCE_R4_3 1
#define FREQUENCY_SOURCE_R4_3 100 // hz
#define PHASE_SOURCE_R4_1 0
#define PHASE_SOURCE_R4_3 0

#define AMPLITUDE_SOURCE_R5_1 1
#define AMPLITUDE_SOURCE_R5_3 1
#define FREQUENCY_SOURCE_R5 100 // hz
#define PHASE_SOURCE_R5_1 0
#define PHASE_SOURCE_R5_3 0

#define AMPLITUDE_SOURCE_NOISE_WHITE_1 1
#define AMPLITUDE_SOURCE_NOISE_WHITE_3 1
#define FREQUENCY_SOURCE_NOISE_WHITE 100 // hz
#define PHASE_SOURCE_NOISE_WHITE_1 0
#define PHASE_SOURCE_NOISE_WHITE_3 0

#define AMPLITUDE_SOURCE_EXPLOSION_1 100
#define AMPLITUDE_SOURCE_EXPLOSION_3 0
#define FREQUENCY_SOURCE_EXPLOSION 20000 // 20 Khz
#define PHASE_SOURCE_EXPLOSION_1 0
#define PHASE_SOURCE_EXPLOSION_3 0

#define POS_SOURCE_NOISE_WHITE_X  0
#define POS_SOURCE_NOISE_WHITE_Y  0
#define POS_SOURCE_NOISE_WHITE_Z  0

#define POS_SOURCE_EXPLOSION_LAT 43.13471   //deg
#define POS_SOURCE_EXPLOSION_LONG 6.01507   //deg
#define POS_SOURCE_EXPLOSION_ALT 6.0        //Height above mean sea (m)
```

Exécuter le programme ROS2 pour créer le source d'explosion

```
$ cd SwarmZ_ROS2/
$ source install/local_setup.bash
$ ros2 run examples_rclcpp_minimal_publisher publisher_member_function
```

Les sources de l'explosion seront enregistrés dans 5 topics correspondant à 5 drones.

```
/source_explosion/toDrone_1
/source_explosion/toDrone_2
/source_explosion/toDrone_3
/source_explosion/toDrone_4
/source_explosion/toDrone_5
```

Lancer plot Data

```
$ sudo apt install ros-humble-plotjuggler-ros
$ ros2 run plotjuggler plotjuggler
```

Le son de l'explosion sera effectué comme la figure 8

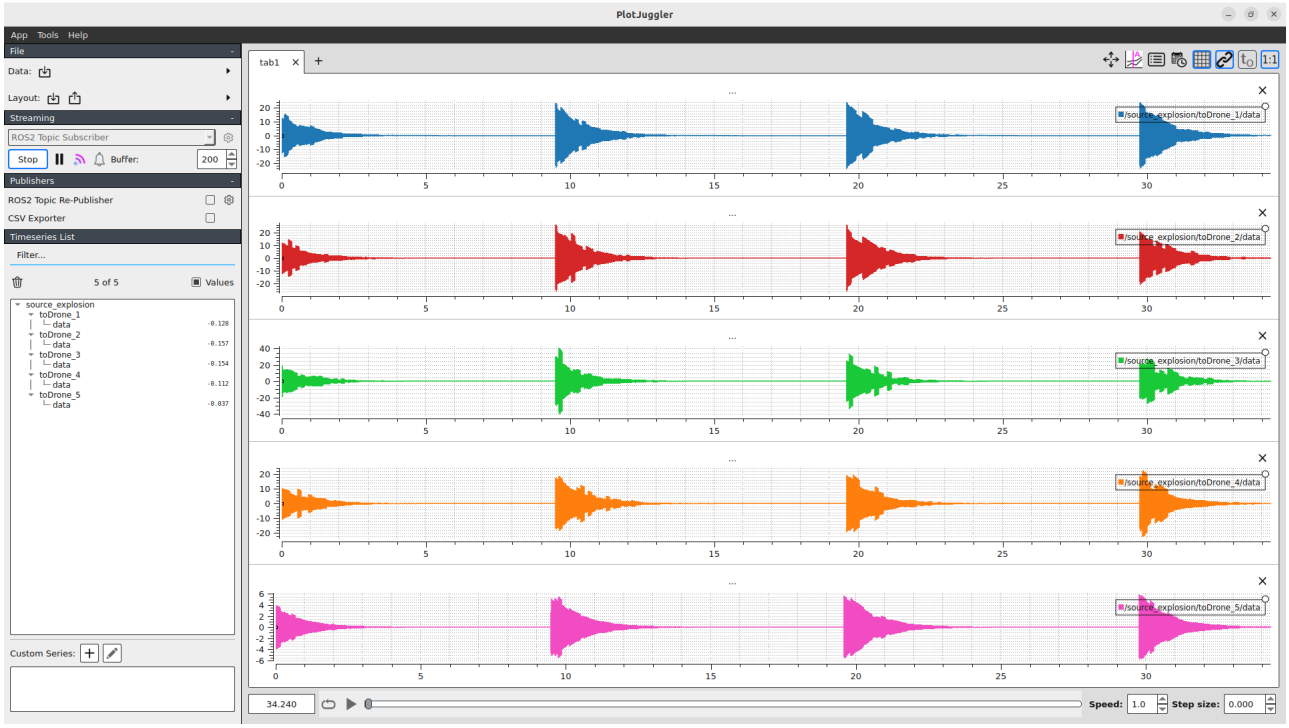

Figure 8: Le type de signal d'explosion que 5 drones ont reçu à différentes distances

References

[1] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *International Conference on Embedded Software*, 2016.