RAPPORT TWIN TEMPORAIRE

Avancement du projet twin

université de BORDEAUX

Marcel Ezedine, Melvin Bertrand, Paulin Bardon, Mahamat Younous Abdraman

April 19, 2023

---

1 Introduction

COPIER DEPUIS ANCIEN DOC

1.1 Fonctionnement des drones

Chaque drone est équipé d'un contrôleur pixhawk ainsi que d'un Raspberry Pi. Le drone est contrôlé en envoyant les informations au Raspberry Pi qui se charge de les transmettre au contrôleur.

1.2 TWIN

Twin consiste à avoir un second drone dans un environnement contrôlé. Lorsqu'un accès non autorisé est détecté sur le drone en opération, cet accès est rédigé par le Raspberry Pi du drone en opération vers le drone en environnement contrôlé. Le Raspberry Pi du drone en opération agit comme un pont entre l'attaquant et le drone en environnement contrôlé. Les opérations sensibles demandées par l'attaquant sont effectuées sur le drone en environnement contrôlé tandis que les actions jugées non sensibles sont effectuées sur le drone en opération (par exemple demander les informations du capteur de température). Cela permet d'avoir des données cohérentes et de renforcer l'impression pour l'attaquant qu'il contrôle le drone en opération.

1.3 MDH

MDH consiste à multiplier le nombre de drones détectés par un attaquant en équipant les drones en opération avec plusieurs microcontrôleurs simulant un drone. Les drones simulés n'ayant pas besoin d'informations provenant du vrai drone, cette approche peut être mise en place très facilement en équipant simplement des drones avec les microcontrôleurs sans se soucier de les connecter au drone hôte. Pour tromper l'attaquant, les drones simulés peuvent altérer certaines informations renvoyées comme par exemple bouger les coordonnées GPS de quelques mètres pour qu'un missile lancé vers le drone ne le touche pas.

1.4 Utilisation simultanée des deux approches

Une opération étant limitée dans le temps, le but est de faire perdre le plus de temps possible à l'attaquant pour que l'opération se termine avant qu'il n'ait eu le temps de contrôler un drone en opération. La multiplication des drones avec MDH réduit les chances de se connecter à un vrai drone. Même si l'attaquant se rend compte qu'il est sur un drone simulé, rien ne garantit que sa prochaine tentative de connexion sera sur un vrai drone. Si l'attaquant trouve un vrai drone, l'utilisation de TWIN permet de l'empêcher de contrôler le drone sensible tout en lui faisant croire que c'est le cas.

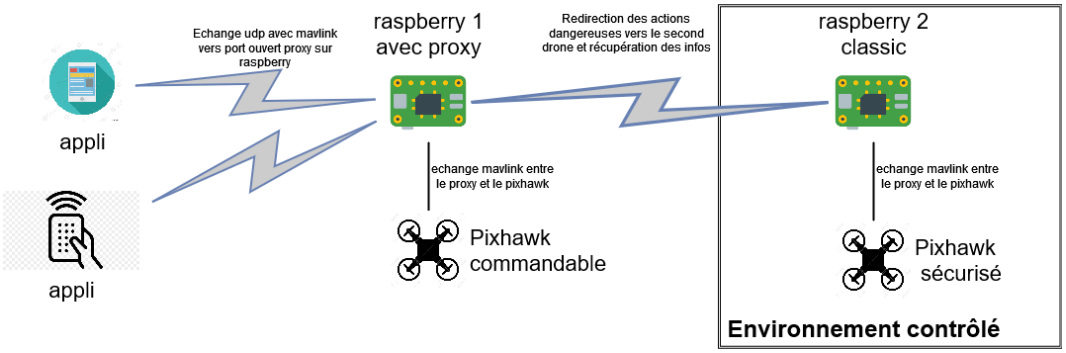

Figure 1: diagramme récapitulatif rapide de la partie twin

2 Avancement

2.1 Application

Dans le dossier AppDrone du git sont présent l'apk pour android et les fichiers qui servent a créer l'application . Pour le moment l'application possède un accueil ou l'on choisi de se connecter à un drone et ensuite une page de commande de drone . Les boutons print du texte mais n'exécute pas encore de commande.

A FAIRE :

-Améliorer l'interface utilisateur (dimensionner les boutons et le terminal)

-Ajouter en backend la partie connections et envoie de commande udp-mavlink vers une adresse et un port.

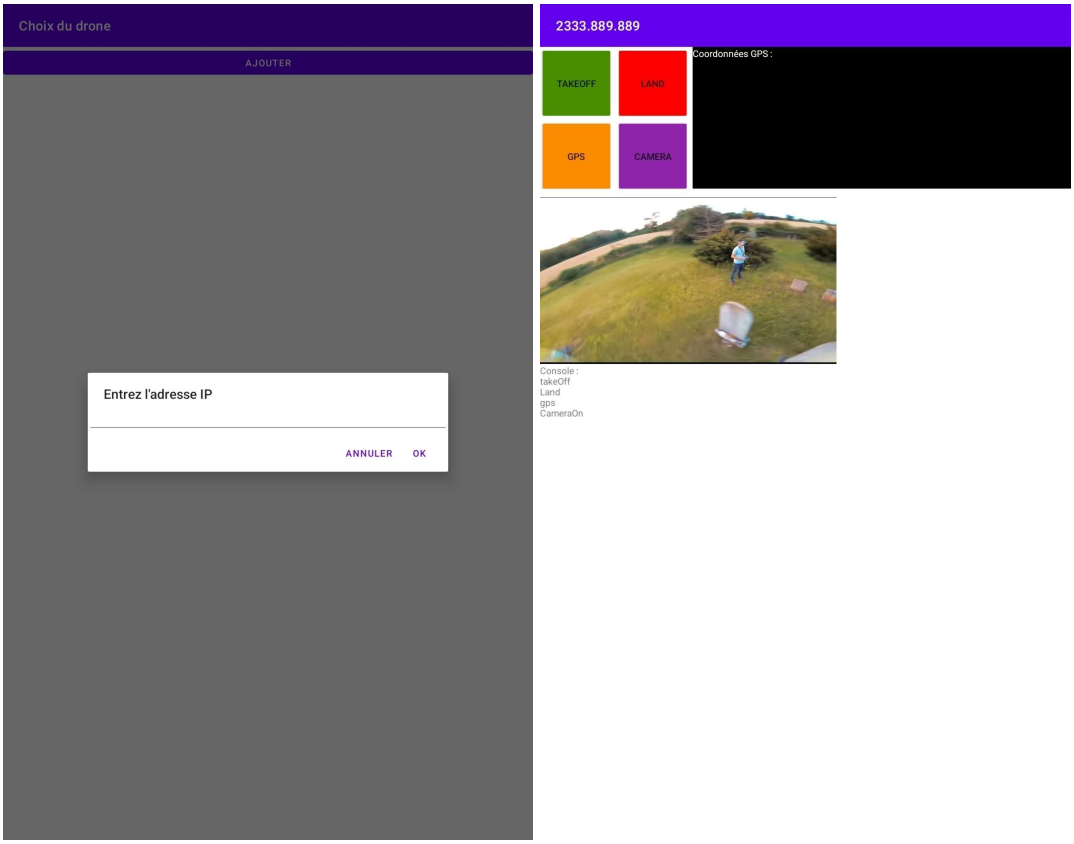

Figure 2: Screen application

2.2 Prise en mains de la simulation

Nous avons fait fonctionner la simulation Jmavsim en utilisant PX4. Nous avons expérimenté les contrôles depuis le terminal avec "commander" depuis Qgroundcontrol et depuis des fichiers python avec mavsdk et pymavlink.

2.3 UDP commande v1

Dans le dossier une version simplifier d'une connexion entre simulation - serveur - client. Lire le README pour voir le fonctionnement. Cela nous a permis d'appréhender le fonctionnement du serveur proxy.
A FAIRE :
-uttiliser plusieurs drones dans une même simulation ou deux simulations pour mettre en place l'architecture twin
-passer entièrement au protocole mavlink.

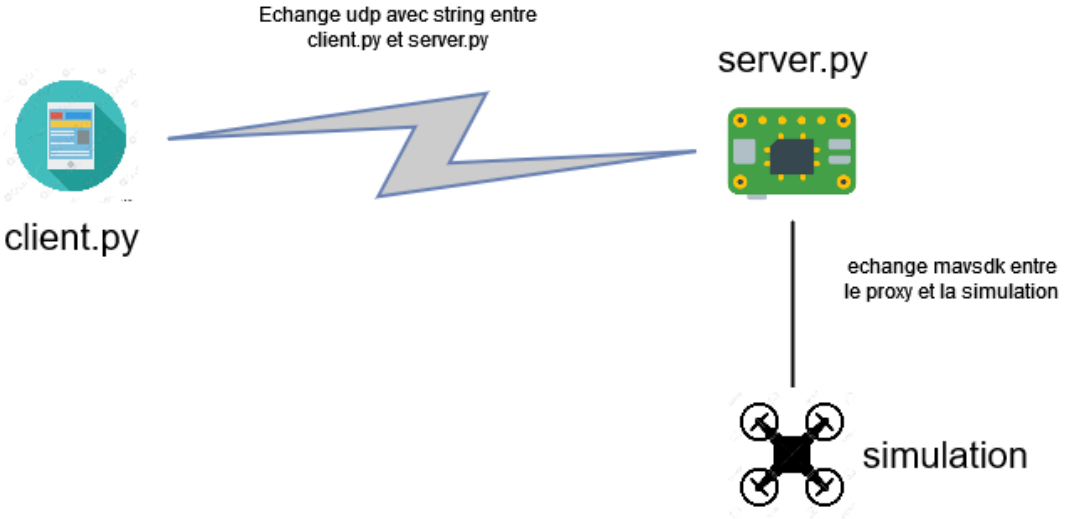

Figure 3: schéma du fonctionnement

2.4 Dossier exemples

il contient un serveur echo qui nous permet de print l'intégralité du packet mavlink pour pouvoir l'analyser, le découper et le comprendre grâce à la documentation.

```
#!/usr/bin/env python3
"""UDP server that prints the messages it receives and sends them back.

Usage:
    python server-echo.py
"""

import socket
import conf

s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
s.bind(('', conf.PROXY_PORT))

while True:
    bytes, address = s.recvfrom(1500)
    print('received data :')
    h = bytes.hex()
    s.sendto(bytes, address)
    print(f"{address} : {' '.join(h[i:i+2] for i in range(0, len(h), 2))}")

print ("closing connection")
s.close()
```

Le serveur proxy lui fait simplement la passerelle entre la simulation et le client sans comprendre le paquet:

```
while True:
    # using mavutil
    # client_msg = master.recv()
    # if not client_msg:
    #     continue

    # client_addr = list(master.clients)[0]
    client_msg, client_addr = s_client.recvfrom(1500);
    print(f"↓ c {client_addr} : {space_hex(client_msg.hex())}")

    # send the packet to the drone
    ret = s_drone.send(client_msg)
    print(f"↑ d {drone_addr} : {space_hex(client_msg.hex())} --- {ret}\n"

    # get the response from the drone
    drone_msg = s_drone.recv(1500)
    print(f"↓ d {drone_addr} : {space_hex(drone_msg.hex())}")

    ret = s_client.sendto(drone_msg, client_addr)
    print(f"↑ c {client_addr} : {space_hex(drone_msg.hex())} --- {ret}\n"
```

Code mavsdk coté client qui parle au serveur proxy :
Ce code fonctionnent quand envoyer directement à la simulation et réalise un takeoff

```
import mavsdk
import asyncio
import conf


async def main():
    drone = mavsdk.System()

    drone_address = f'udp://{conf.PROXY_ADDRESS}:{conf.PROXY_PORT}'
    #drone_address='udp://:14550'
    print(f'connecting to {drone_address}')
    await drone.connect(drone_address)
    print(f'connected to  {drone._mavsdk_server_address}:{drone._port}')

    print('arming')
    await drone.action.arm()
    print('armed')

    print('takeoff')
```

```
    await drone.action.takeoff()
    print('took off')


if __name__ == '__main__':
    asyncio.run(main())
```

Nous rencontrons des problèmes de transmissions :
-le paquet échanger n'évolue pas
-le client ne fait pas la connexion et ne passe pas au takeoff
Remarque drone.connect(adresse) fonction de mavsdk envoie un PING + 5 HEARTBEAT mais n'acceptes pas la connexion si on renvoie un ACK pour chacun ou un HEARTBEAT ou une réponse a PING qui est d'ailleurs une fonction obsolètes dans la doc (DEPRECATED: Replaced by SYSTEM_TIME (2011-08). to be removed / merged with SYSTEM$_TIM$)
A FAIRE :
-Approfondir la documentation mavsdk et comprendre complètement la fonction connect.
-rajouter la fonction de compréhension du paquets pour pouvoir décider vers qu'elles simulations l'envoyer.
- construire l'architecture twin

2.5 UDP commande v2 et aide

Ici on cherche à reprendre le premier udp commande et rendre la partie entre le client et le serveur compatible avec le protocole mavlink. Plutot que de passer par mav sdk qui a une architecture complexe, on va utiliser les commandes de la bibliothèque pymavlink.

Les commandes sont de cette forme(voir la doc pour comprendre):

```
master = mavutil.mavlink_connection('udp:127.0.0.1:14550')
master.wait_heartbeat()

master.mav.command_long_send(
        master.target_system,  # Système cible (0 pour le système par déf
        master.target_component,  # Composant cible (0 pour le composant
        mavutil.mavlink.MAV_CMD_COMPONENT_ARM_DISARM,  # Code de commande
        0,  # Confirmation automatique de la commande
        1,  # Armer le drone (0 pour désarmer)
        21196,  # Non utilisé pour l'armement
        0,  # Non utilisé pour l'armement
        0,  # Non utilisé pour l'armement
        0,  # Non utilisé pour l'armement
        0,  # Non utilisé pour l'armement
        0)  # Non utilisé pour l'armement

 master.mav.command_long_send(
            master.target_system,    # Système cible (0 pour le système pa
```

```
        master.target_component, # Composant cible (0 pour le composa
        mavutil.mavlink.MAV_CMD_NAV_TAKEOFF, # Code de commande de dé
        0,     # Confirmation automatique de la commande
        5,     # Paramètre 1 : Hauteur du décollage en mètres
        0,     # Paramètre 2 : Latitude cible en degrés
        0,     # Paramètre 3 : Longitude cible en degrés
        0,     # Paramètre 4 : Altitude cible en mètres
        0,     # Paramètre 5 : Paramètre vide
        0,     # Paramètre 6 : Paramètre vide
        0)     # Paramètre 7 : Paramètre vide
```

Cette suite d'instruction sur le port d'une simulation est sensé lancer le takeoff (comme le ferait mavsdk) mais ça ne marche pas. En fait la simulation n'autorise pas la commande "arm" et se disarm avant le takeoff. c'est un problème de sécurité a cause du mode de la simulation mais si je passe en mode manuel pour enlever le blocage la simulation crash chez moi. J'ai aussi tester pleins d'autre facon de desactiver ou de changer le mode rien a faire pour le moment. Cependant l'armement apparaît dans les logs et si on "arm" depuis le terminal le takeoff qui était en attente s'exécute montrant le bon fonctionnement de ces instructions .
Si on envoie ces instructions à un serveur proxy, on peut les analyser et les comprendre avec la fonction:

```python
while True:
    # Attendre de recevoir un message
    msg = connection.recv_match()

    # Vérifier si le message est valide
    if not msg:
        continue
    print(msg)
    # Traitement des messages reçus
    if msg.get_type() == 'HEARTBEAT':
        print(f'Received HEARTBEAT message from system {msg.get_srcSystem
    if msg and msg.get_type() == 'PING':
        print(f'Received PING message from system {msg.get_srcSystem()}')
    if msg and msg.get_type() == 'GLOBAL_POSITION_INT':
        print('etc')
```

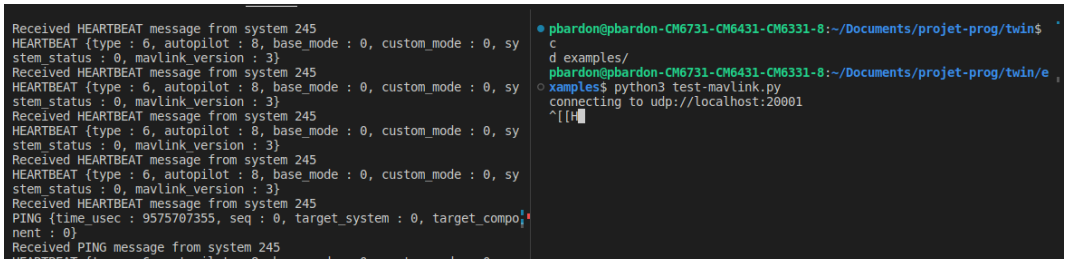

Figure 4: Screen démonstrations de la reconnaissance de fonction

Donc on peut envoyer des commandes mavlinks et les comprendre et les renvoyer sur la sim (ou les sims) qui seront plus tard nos drones. Pour le moment je vais continuer d'utiliser mavsdk pour le coté serveur - simulation pour éviter le problème citer plus haut.

A FAIRE :
-Se rapprocher de l'architecture twins avec deux drones et implémenter correctement le serveur