Étude et conception d'une plate-forme de multi-simulation et de contrôle de drones

Yannick Presse

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

CENTRE REGIONAL DE LORRAINE

_____

MEMOIRE

présenté en vue d'obtenir

le DIPLOME d'INGENIEUR CNAM

SPECIALITE INFORMATIQUE
OPTION RESEAUX, SYSTEMES ET MULTIMEDIA (IRSM)

par

Yannick PRESSE

_____

Étude et conception d'une plate-forme de multisimulation et de contrôle de drones

Soutenu le 27 septembre 2012

_____

JURY

PRESIDENT :	J.P. ARNAUD	Professeur de Chaire au CNAM et président du jury
MEMBRES :	S. VIALLE	Responsable Régional de Spécialité informatique au CNAM de Lorraine, et tuteur CNAM du mémoire
	V. GALTIER	Enseignant-chercheur à SUPELEC, responsable du travail de mémoire
	L. CIARLETTA	Membre de l'équipe Madynes au LORIA, et enseignant à l'École Nationale Supérieure des Mines de Nancy, Université de Lorraine
	G.J. HERBIET	Technical manager à HOTCITY S.A.

Remerciements

Je tiens à remercier Monsieur Stéphane Vialle, Professeur à SUPELEC, de m'avoir offert l'opportunité d'effectuer mon stage au sein du groupe IDMaD de l'équipe IMS.

Je remercie particulièrement Madame Virginie Galtier, enseignante et chercheur à SUPELEC, pour son aide et sa patience pour la rédaction de ce document et pour ses conseils avisés sur la programmation en Java.

Je remercie aussi Monsieur Laurent Ciarletta, membre de l'équipe Madynes au LORIA et enseignant à l'École Nationale Supérieure des Mines de Nancy, pour m'avoir fourni le logiciel ARDroneMines et pour m'avoir accueilli au LORIA pour des séances d'essais en vol.

Table des matières

I Contexte du projet **8**

1 Introduction : complexité de l'évaluation des solutions d'informatique ambiante **9**

1.1 Définition et facteurs d'émergence de l'informatique ambiante 9
1.1.1 Début de maturité technologique des objets communicants . 9
1.1.2 Définition historique de l'informatique ambiante 9
1.1.3 Des apports indispensables à la vie au quotidien 9
1.2 Recherches et ressources en Lorraine 10
1.2.1 Une recherche multidisciplinaire 10
1.2.2 Des plates-formes d'expérimentation favorisant les échanges et l'approche contextuelle 11
1.3 Les difficultés de tests et de validation 12
1.4 Articulation de ce document 13
1.5 Réalisation d'un projet en collaboration 13

2 État de l'art en matière de simulation de systèmes complexes **15**

2.1 Simulation 15
2.1.1 Motivations 15
2.1.2 Modélisation 16
2.1.3 Simulateur 17
2.1.4 Validation et vérification 18
2.1.5 Données de simulation 18
2.1.6 Types de simulation/taxinomie 18
2.2 Multisimulation 19
2.2.1 Définition et intérêts 19
2.2.2 Niveaux de correspondance 19
Correspondance sémantique des modèles (multimodélisation) 19
Correspondance logicielle des simulateurs (multisimulation) . 20
2.2.3 Accréditation 20
2.2.4 Cadres pour la multisimulation 21
Cadres dédiés 21
Solutions non dédiées 24
2.2.5 Projets existants de cosimulation 25
Cosimulations ad hoc pour l'évaluation d'espaces intelligents 25
Projets intégrant un simulateur réseau 25
2.2.6 Bilan des solutions de multisimulation 27

II Définition et spécification d'un démonstrateur de multisimulation **28**

3 Exemple de scénario visé **29**

4 Simulateurs nécessaires **30**
4.1 Simulateurs de réseaux . . . 30
4.1.1 OMNeT++ . . . 30
Description . . . 31
INET . . . 32
INETMANET . . . 33
MiXiM . . . 33
4.1.2 NS-3 . . . 34
4.1.3 OPNET Modeler . . . 35
4.2 Simulateurs de drones . . . 36
4.2.1 Player/stage . . . 37
4.2.2 FlightGear . . . 38
4.2.3 SAMOVAR . . . 38

III Conception, développement et évaluation du démonstrateur **40**

5 Interface de contrôle d'AR.Drone **41**
5.1 Présentation de l'AR.Drone et du logiciel existant . . . 41
5.1.1 Description AR.Drone . . . 41
Caractéristiques du drone . . . 42
Les communications . . . 43
Firmware . . . 45
5.1.2 Présentation du logiciel ARDroneMines . . . 45
5.2 Solution JaSMinDrone . . . 51
5.2.1 Méthodologie de travail . . . 51
5.2.2 Corrections des limitations d'ARDroneMines . . . 51
5.2.3 Fonctionnalités ajoutées . . . 62
5.2.4 Accompagnement des utilisateurs et des développeurs . . . 70

6 Couplage des composants **72**
6.1 JaSMinDrone/SAMOVAR . . . 72
6.1.1 Architecture de la solution . . . 72
6.1.2 Lien de JaSMinDrone vers Matlab . . . 73
6.1.3 Interaction entre matlab et SAMOVAR . . . 73
6.1.4 Lien de SAMOVAR vers JaSMinDrone . . . 75
6.1.5 Amélioration par l'utilisation de S-Function . . . 77
6.1.6 Vol en formation de drones simulés . . . 78
6.2 JaSMinDrone/OMNet/SAMOVAR . . . 78
6.2.1 Étude de la solution Veins . . . 78
6.2.2 Communication de SAMOVAR à OMNet++ . . . 80
6.2.3 Communication de OMNet++ à SAMOVAR . . . 92
6.2.4 Limitations . . . 99

7 Utilisation et évaluation du démonstrateur **100**
7.1 Plates-formes de test . . . 100
7.2 Tests réalisés . . . 100
7.3 Facilité d'utilisation . . . 102
7.4 Tests futurs . . . 102

Conclusion et perspectives **105**
Bilan du travail réalisé . . . 105
Rappel des objectifs du projet . . . 105
L'interface de contrôle d'AR.Drone . . . 105
Association de l'interface à deux simulateurs . . . 106
Problèmes non résolus . . . 107
Perspectives . . . 107

IV Annexes **109**

8 Les informations ajoutées à l'interface utilisateur **110**

9 La classe Autopilot **114**

10 Le masque d'état du drone **117**

11 Documentation utilisateur **121**
11.1 About . . . 121
11.2 Technical requirements . . . 121
11.3 Interactive interface . . . 122
11.3.1 Connection . . . 122
11.3.2 Graphic User interface . . . 122
11.3.3 Multicast . . . 125
11.3.4 Flight scenarios . . . 127
Tag tracking . . . 127
Dynamic follow . . . 128
11.4 Non-interactive way . . . 131
11.5 Known issues . . . 132

12 Documentation développeur **133**
12.1 About . . . 133
12.2 ARDroneDemo package . . . 133
12.2.1 gui_client . . . 133
ARDroneDemo.java . . . 133
UserNetworkConfig.java . . . 133
ARDroneUserInterface.java . . . 134
12.2.2 non_interactive_client . . . 134
12.2.3 Library . . . 134
12.2.4 data . . . 134
12.3 ardronelib package . . . 134
12.3.1 lib.ardrone . . . 134
ARDroneEntity.java . . . 134
ARDroneRegistry.java . . . 135

Global.java 135
12.3.2 lib.ardrone.control 135
12.3.3 lib.ardrone.flightscenarii 136
DynamicFollowing.java 136
TagFollowing 136
12.3.4 lib.ardrone.listeners 136
12.3.5 lib.ardrone.navigationdata.structures 136
12.3.6 lib.ardrone.navigationdata 136
12.3.7 lib.ardrone.network 137
12.3.8 lib.ardrone.phone 137
12.3.9 lib.ardrone.video 138
12.3.10 lib.utils 138
12.3.11 parrot.video 138

Table des figures

1.1 L'évolution des ordinateurs (d'après JB Waldner, Nano-informatique et intelligence ambiante, Hermes Science, 2007) 10
1.2 Convergence de projets en Lorraine 14

2.1 Les composants d'une simulation, source *SISO* 15
2.2 Relations de modélisation et de simulation 16
2.3 Les types de modèles 17
2.4 Vue fonctionnelle d'une fédération HLA, source *DoD* 21
2.5 Hétérogénéité des RTI 22
2.6 Cosimulation FMI dans une infrastructure distribuée, source Modelisar 24
2.7 L'architecture iTETRIS 26
2.8 L'architecture de Veins 27

4.1 Les composants d'OMNeT++. 31
4.2 Execution d'une simulation dans l'environnement graphique. 32
4.3 Comparaison entre les composants d'un même noeud avec MiXiM (à gauche) et avec INETMANET (à droite). 33
4.4 Composition de NS-3. 34
4.5 Environnement de modélisation hiérarchique (source OPNET) . . . 36
4.6 Une simulation avec stage (à gauche) et avec Gazebo (à droite) . . . 37
4.7 Le rendu en trois dimensions d'un simulation FlightGear 38
4.8 Simulation d'un quadcopter dans une représentation de la smartroom de SUPELEC 39

5.1 L'AR.Drone équipé de sa coque d'extérieur 41
5.2 La coque d'intérieur de l'AR.DRone 42
5.3 Représentation de l'attitude d'un mobile 42
5.4 Les communications avec l'AR.Drone 43
5.5 Visualisation graphique des *NavData* 44
5.6 Diffusion des vidéos en mode PiP 44
5.7 L'interface utilisateur d'ARDroneMines. 46
5.8 Créer une nouvelle entité drone. 47
5.9 La configuration de la connexion. 47
5.10 Exemples de *tag 2D* 48
5.11 Le diagramme des séquences des *threads* principaux : Control, Video et NavData 52
5.12 Les interfaces de configuration du réseau d'ARDroneMines (à gauche) et de JaSMinDrone (à droite) 53
5.13 Les interfaces utilisateurs d'ARDroneMines (à gauche) et de JaSMinDrone (à droite) 53

5.14 Capture de paquet avec l'interface ARDroneMines 53
5.15 Capture de paquet avec l'exemple du SDK Parrot 54
5.16 Capture de paquet avec l'application Android 54
5.17 Comparaison du paquet envoyé au drone par Android (à gauche), ARDroneMines (au milieu) et le SDK (à droite) 54
5.18 Le diagramme des séquences pour le *thread* navdata 56
5.19 Le diagramme des séquences pour le *thread* navdata 57
5.20 Le flux UDP des ATCommands analysé par Wireshark 59
5.21 Le problème intermittent d'affichage de l'interface graphique 62
5.22 Connexion de deux AR.Drones avec JaSMinDrone 63
5.23 La configuration en multicast 64
5.24 La fenêtre utilisateur en mode multicast passif 65
5.25 Un seul drone est autorisé à se connecter en multicast 66
5.26 La séparation de l'interface graphique et de la bibliothèque 67
5.27 La configuration en mode *localhost* 68
5.28 Le mode NavDataDemo 69

6.1 L'architecture de couplage entre JaSMinDrone et SAMOVAR 72
6.2 Le modèle de quadcopter fourni avec SAMOVAR 73
6.3 L'affichage en clair dans Matlab des commandes reçues de JaSMinDrone 74
6.4 Le contenu du bloc Quadrotor 74
6.5 Capture Wireshark représentant la création d'un nouveau *socket* à chaque transmission de paquet 75
6.6 Le bloc permettant la réception et le traitement des commandes de JaSMinDrone 75
6.7 La composition d'un paquet NavData 76
6.8 La décomposition d'un paquet NavData capturé par *Wireshark* . . . 76
6.9 Le modèle SAMOVAR permettant le couplage avec JaSMinDrone . . 77
6.10 Contrôle par JaSMinDrone d'un drone simulé par SAMOVAR 77
6.11 Capture Wireshark représentant l'utilisation d'un seul *socket* après l'utilisation de S-Function 78
6.12 Les blocs Réception et Envoi ajoutés au code du *leader* 79
6.13 Un vol en formation de drones simulés, le *leader* étant contrôlé par JaSMinDrone 80
6.14 Architecture de la solution permettant la communication entre SAMOVAR et OMNet++ 80
6.15 Récupération des coordonnées du drone dans le bus *state*, *by-pass* à l'intérieur du sous-bloc Sensors et ajout au connecteur de sortie . . . 82
6.16 Le modèle d'un drone intégrant le couplage avec JaSMinDrone et permettant d'envoyer sa position par *socket* 84
6.17 Représentation graphique du réseau CouplageSam 87
6.18 Le couplage de l'ensemble des composants développés permet le contrôle d'un drone simulé et du nœud le représentant 89
6.19 Le modèle SAMOVAR comportant deux drones et permettant le couplage avec OMNet++ 91
6.20 Simulation mettant en jeu deux drones représentés par deux nœuds s'échangeant des messages ping 93
6.21 Le modèle de SAMOVAR permettant un couplage bidirectionnel avec OMNet++ 97
6.22 L'exécution d'une simulation couplant SAMOVAR et OMNet++ . . 98

8.1 Les interfaces utilisateur d'ARDroneMines (à gauche) et de JaSMinDrone (à droite) . . . 110
8.2 Information de mode *emergency* . . . 111
8.3 Information de batterie faible . . . 112
8.4 L'affichage lors d'un problème réseau . . . 113

11.1 Initial window . . . 122
11.2 drone Parameters . . . 122
11.3 User interface . . . 123
11.4 Navdata demo enabled . . . 124
11.5 Vertical cam . . . 124
11.6 Picture-in-Picture display . . . 125
11.7 Emergency mode . . . 126
11.8 Low battery . . . 126
11.9 Multicast connection . . . 127
11.10 Multicast passive interface . . . 127
11.11 Yellow 2d Tag . . . 128
11.12 Tag detection . . . 128
11.13 Drones wifi characteristics . . . 129
11.14 Ping on each drone . . . 130
11.15 Drones configuration in the software . . . 130
11.16 Two connected drones . . . 131

12.1 Sequence diagram (threads view) . . . 135
12.2 Sequence diagram (navdata view) . . . 137
12.3 NavData Stream . . . 137

Première partie

Contexte du projet

Chapitre 1

Introduction : complexité de l'évaluation des solutions d'informatique ambiante

1.1 Définition et facteurs d'émergence de l'informatique ambiante

1.1.1 Début de maturité technologique des objets communicants

Les objets communicants bénéficient d'un contexte technologique très dynamique. Il y a une explosion du développement des réseaux informatiques et des connexions à l'internet grâce à la miniaturisation des processeurs et des interfaces réseau. Les appareils mobiles profitent aussi d'une baisse de consommation d'énergie, d'une couverture réseau étendue et de plus en plus performante. On voit apparaitre de nouvelles technologies sans fil (*NFC*) et même sans énergie (*RFID*). Des progrès sont également effectués dans les technologies d'apprentissage, de connaissance du contexte et d'intelligence artificielle.

On assiste donc aujourd'hui à une multiplication des objets ayant des capacités de perception de l'environnement, de traitement de l'information et de communication.

1.1.2 Définition historique de l'informatique ambiante

Mark Weiser a inventé l'expression *informatique ubiquitaire* en 1988 pour faire référence à une informatique omniprésente, complètement intégrée dans les activités journalières de l'utilisateur et transparente pour celui-ci. La figure 1.1 illustre l'évolution de systèmes informatiques, d'un modèle non connecté et dépendant de l'énergie dans les années 1960 à des systèmes communicants, mobiles, intelligents et diffus de nos jours.

Les objets et les technologies communicants mis à la disposition du grand public et distribués aujourd'hui à grande échelle rendent cette vision de l'informatique ambiante de plus en plus réelle et présente au quotidien.

1.1.3 Des apports indispensables à la vie au quotidien

Il existe de nombreuses applications à l'informatique ambiante. On peut citer, entre autres, deux domaines émergents :

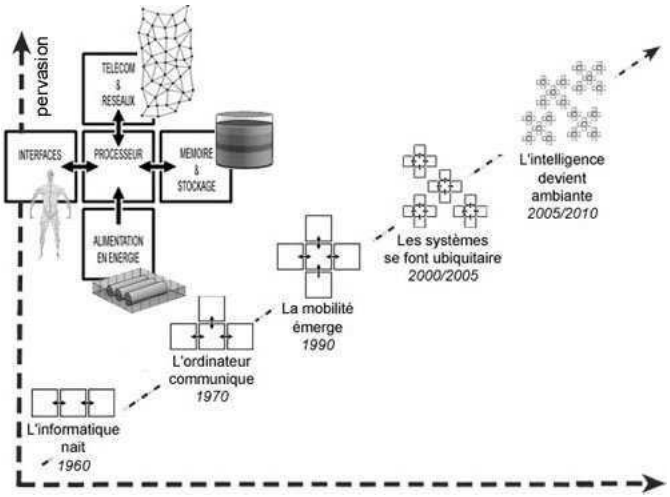


FIGURE 1.1 – L'évolution des ordinateurs (d'après JB Waldner, Nano-informatique et intelligence ambiante, Hermes Science, 2007)

- l'aide au maintien à domicile (*assisted living*) des personnes âgées ou à mobilité réduite apporte une réponse à de réels besoins grâce au développement de l'informatique ambiante et à un contexte social favorisant les recherches dans ce domaine. Par exemple le programme européen AAL JP(*Ambient Assisted Living Joint Programme*) a pour but l'amélioration de la condition de vie des personnes âgées et le renforcement des opportunités industrielles en Europe grâce à l'utilisation des technologies d'information et de communication [1] ;
- l'écologie et plus particulièrement l'optimisation des dépenses énergétiques, aussi bien au niveau des foyers (intégration de programmateurs horaires intelligents, d'appareils d'électroménager communicants et de moyens de production d'électricité ou de chaleur) qu'au niveau des fournisseurs avec le développement des réseaux électriques intelligents (*SmartGrids*) et des compteurs intelligents (*smart metering*) comme Linky d'ERDF qui, après une phase de test validée en 2011, devrait être installé dans 35 millions de foyers d'ici 2020 [2].

1.2 Recherches et ressources en Lorraine

1.2.1 Une recherche multidisciplinaire

L'informatique ambiante regroupe plusieurs domaines scientifiques et techniques, tels l'informatique distribuée, l'informatique mobile, les réseaux de capteurs, l'interaction homme-machine ou bien l'intelligence artificielle.

Ce stage s'est déroulé à SUPELEC, sur le campus de Metz, au sein du groupe IDMaD (Informatique Distribuée et Masse de Données) de l'équipe IMS (Informatique, Multimodalité et Signal). L'équipe IMS développe des recherches en calcul intensif et distribué ainsi qu'en apprentissage et traitement du signal dans le but de concevoir des systèmes dits situés, capables de s'adapter à leur environnement, ce qui correspond à un des éléments de l'informatique ambiante, le traitement contextualisé. L'équipe IMS est membre de l'UMI GeorgiaTech-CNRS 2958.

Le LORIA (Laboratoire lorrain de recherche en informatique et ses applications) est situé à Nancy et il est une unité de recherche commune au CNRS, à l'université de Lorraine et à l'INRIA. On peut citer les équipes de recherches suivantes qui travaillent sur des aspects touchant à l'informatique ambiante :

- MADYNES (supervision des réseaux et des services dynamiques) œuvre sur des nouveaux paradigmes et architectures de supervision et de contrôle capables de maîtriser la *dynamicité* croissante des services et résistantes au facteur d'échelle induit par l'Internet ubiquitaire ;
- MAIA (MAchine Intelligente Autonome) travaille sur la conception d'entités informatiques capables de percevoir l'environnement, de l'interpréter et d'agir sur celui-ci avec une relative autonomie ;
- TRIO (Temps Réel et InterOpérabilité) a pour objectif d'assister le concepteur dans les tâches de construction, de validation et de dimensionnement d'applications temps réel distribuées et travaille sur la formalisation des propriétés d'interopérabilité et sur leur vérification.

1.2.2 Des plates-formes d'expérimentation favorisant les échanges et l'approche contextuelle

Dans un contexte multidisciplinaire, il est nécessaire de communiquer et d'échanger les savoirs. Pour cela, des plates-formes d'étude réelles (*smartrooms*) offrant un cadre commun ont été conçues, et elles visent à développer un environnement intelligent.

Ainsi au LORIA, une plate-forme d'informatique située [1] a été bâtie comme un appartement intelligent pour l'assistance à la personne. Cette plate-forme permet :

- de développer physiquement un réseau de capteurs ;
- dans un but d'effectuer un suivi médical dans le temps d'une personne (actimétrie), la réalisation d'environnements intelligents et de robots d'assistance.

Le LORIA met également en place actuellement une plate-forme de *Cyber Physical System (CBS)* appelée AETOURNOS (*Airborne Embedded auTonomOUs Robust Network of Objects and Sensors*)[3] constituée d'un ensemble de drones volants et permettant de développer et d'évaluer des protocoles et applications adaptés à des contraintes physiques réelles et simulées.

À SUPELEC, une *smartroom* [2] a été construite en 2010 et elle permet elle aussi de profiter d'une plate-forme d'environnement intelligent. La *smartroom* permet de répondre à différents défis technologiques :

- le traitement du son et de la parole, comme la détection et la localisation de personne par un réseau de microphones ou l'utilisation de techniques d'holophonie pour permettre à des personnes malvoyantes de s'orienter grâce aux sons ;
- le traitement des images et de la vidéo pour la détection et la reconnaissance d'objets vus sous différents angles et dans des conditions de luminosité différentes ou bien pour détecter des situations anormales comme la chute d'une personne âgée ;

1. Voir le site web : http ://infositu.loria.fr/
2. Voir la documentation : http ://www.supelec.fr/actu/DossierSmartRoom.pdf

– la robotique cognitive par l'approche de l'apprentissage afin de permettre l'autonomie lors d'un changement d'environnement et d'améliorer l'interaction homme-machine, utile pour une application d'aide médicale (infirmière robotique).

1.3 Les difficultés de tests et de validation

Ces plates-formes d'expérimentation réelle sont une aide nécessaire pour la conception et l'expérimentation de projets. Elles présentent cependant des limites :

– au niveau des coûts, aussi bien matériels que humains dus à la maintenance de ces environnements ;
– à cause de l'échelle réduite et des possibilités de configuration ;
– au niveau des équipements matériels ou logiciels, qui sont limités aux produits disponibles sur le marché et qui peuvent être frappés d'obsolescence en cours de projet ;
– de non-reproductibilité des expériences à cause de la complexité des systèmes et de la mise en jeu de sujets humains.

Les conséquences de ces limitations sont des difficultés à apporter des garanties industrielles, des tests et une validation étant nécessaires avant d'envisager une production ou une mise à la disposition du grand public, et des difficultés à se comparer à d'autres projets, le cadre expérimental étant différent dans chaque environnement.

Il existe bien un projet visant à effectuer des comparaisons de solutions en matière d'*Assisted Living* en organisant un concours annuel. Le projet EvAAL (*Evaluating Ambient Assisted Living systems*) [4] a pour but de rassembler les communautés de recherches scientifiques et industrielles afin d'évaluer différentes approches et d'envisager de nouvelles possibilités de recherches. Ce projet est sponsorisé par des industriels (Texas Instrument, Asus) et par des projets de recherches (UniversAAL, GiraffPlus). Cependant, tous les compétiteurs doivent se déplacer pour utiliser les mêmes installations de manière à pouvoir faire des comparaisons objectives. De plus, et contrairement à d'autres domaines scientifiques, il n'existe pas de liste standard admise de critères d'évaluation ou de benchmarks.

Les *smartrooms* composent un système complexe mettant en œuvre des composants hétérogènes et interconnectés. Il est donc très difficile de prédire le comportement global en étudiant individuellement ces composants. De plus, les experts utilisent des outils propres à leurs domaines scientifiques et cela provoque les problèmes suivants :

– les outils ne sont pas réutilisables par des équipes travaillant dans d'autres domaines scientifiques ;
– une perte de temps induite par un développement à partir de zéro ;
– il est impossible de comparer les solutions entre elles et il existe un manque de références pour l'évaluation d'une solution.

Une solution pour lever ces limitations est d'avoir recours à la simulation. Il n'existe cependant pas de simulation de *smartrooms* et la simulation envisagée fait intervenir plusieurs modèles, par exemple les communications, l'environnement, la partie logicielle ou les utilisateurs, et concerne plusieurs domaines scientifiques, comme

l'informatique, la physique, la biologie, etc.

1.4 Articulation de ce document

Les *smartrooms* sont un bon moyen pour développer de nouvelles technologies, cependant des tests sont nécessaires pour valider ces technologies. Les *smartrooms* présentant des limitations pour la réalisation de ces tests, le recours à la simulation est indispensable pour l'évaluation des solutions développées.

Il n'existe pas de simulation de *smartrooms* et deux possibilités s'offrent à nous :
- développer un simulateur à partir de zéro ;
- utiliser des solutions de simulations existantes.

L'utilisation de solutions existantes est l'approche choisie dans ce projet, mais dans un contexte d'informatique ambiante impliquant de nombreux domaines scientifiques, elle suppose d'avoir recours à des simulations hétérogènes et de les faire collaborer.

Le chapitre 2 dresse un état de l'art de la simulation et de la multisimulation (assemblage de simulations). La partie II décrit le démonstrateur visé faisant intervenir un simulateur réseau ainsi qu'un simulateur et une interface de contrôle de drones volants. La partie III présente le développement de cette interface de contrôle avant d'expliquer la réalisation du couplage entre les différents composants. Le chapitre 7 ébauche des pistes d'évaluation du démonstrateur. La partie 7.4 dresse le bilan du projet et introduit les perspectives envisagées.

1.5 Réalisation d'un projet en collaboration

Ce projet est issu d'une collaboration entre SUPELEC, le LORIA et l'École Nationale Supérieure des Mines de Nancy (ENSMN), trois établissements situés en Lorraine comme l'illustre la figure 1.2.

Ce projet aboutit donc à la convergence de plusieurs projets et il a été mené à SUPELEC, avec les matériels de la salle robotique de la *smartroom*. Le simulateur de drone utilisé et décrit au 4.2 est issu d'un projet du LORIA. L'interface originale de contrôle des AR.Drones a été développée par des étudiants de l'ENSMN, elle sera présentée au 5.1.2.

Pour les besoins du projet, j'ai été appelé à travailler parfois au LORIA à Nancy.

J'ai aussi été amené à encadrer deux projets de conception concernant le couplage de l'interface de contrôle avec un simulateur de drone. Un projet a été mené par des étudiants de SUPELEC sur le simulateur SAMOVAR, l'autre par des étudiants de l'ENSMN sur le simulateur FlightGear.

FIGURE 1.2 – Convergence de projets en Lorraine

Chapitre 2

État de l'art en matière de simulation de systèmes complexes

2.1 Simulation

Une simulation consiste en l'imitation à l'aide d'un modèle d'un processus ou d'un système du monde réel et s'exécutant dans un simulateur. Zeigler [5] définit la simulation comme étant un ensemble composé du système étudié, de son modèle, de l'exécution de celui-ci, de données de simulation et du cadre expérimental, comme l'illustre la figure 2.1.

Ainsi, il sera possible d'établir un modèle à partir du système que l'on veut étudier dans un cadre expérimental défini à l'aide de relations de modélisation, puis d'exécuter ce modèle dans un simulateur à l'aide de relations de simulation comme l'illustre la figure 2.2. Des processus de validation et de vérification sont nécessaires pour attester de la justesse de la simulation.

2.1.1 Motivations

La simulation sert à répondre à des questions que l'on se pose dans le cadre d'une étude. Le but de l'étude peut être le test ou la validation d'un système. Il peut s'agir aussi d'entrainements de personnel, de prise de décision par comparaisons successives ou bien de la visualisation et de l'analyse d'une solution encore non existante. On peut aussi avoir la volonté d'effectuer un passage à l'échelle, par exemple dans un

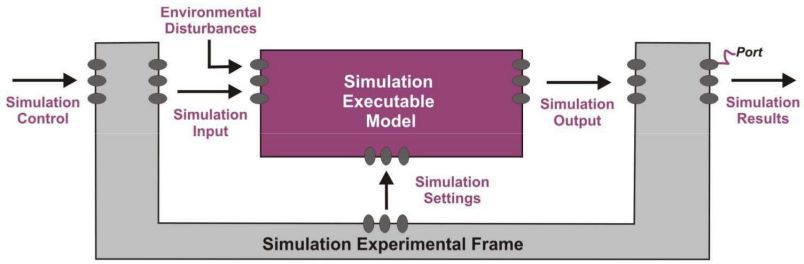


FIGURE 2.1 – Les composants d'une simulation, source *SISO*

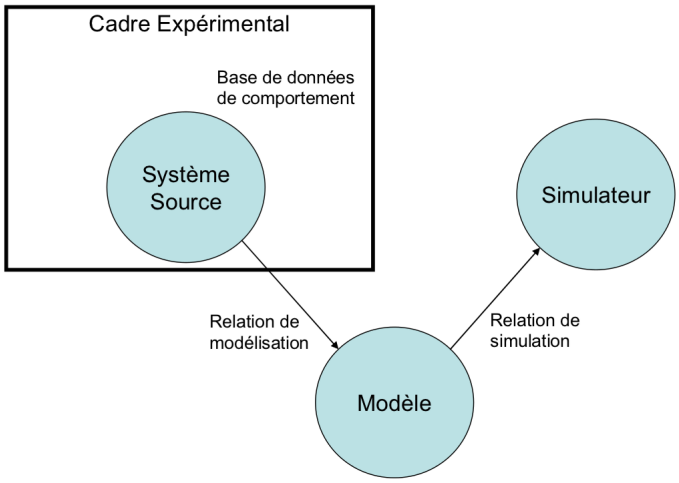


FIGURE 2.2 – Relations de modélisation et de simulation

projet de réseaux de capteurs sans fil. Il peut être difficile d'obtenir des réponses avec le système réel, car il n'existe tout simplement pas (cas du prototypage) ou bien il est difficilement manipulable ou observable :

- le système est trop cher, aussi bien à acquérir qu'à maintenir à jour ;
- l'expérience serait dangereuse pour l'homme ou la nature, par exemple une explosion nucléaire ou l'étude de la chute d'une personne âgée pour un projet d'aide au maintien à domicile ;
- le système est imprévisible ou non reproductible, comme une catastrophe naturelle ou un mouvement de foule ;
- le système est complexe (système de systèmes) et met en œuvre un grand nombre de composants et d'interactions, comme dans le cas de l'informatique ubiquitaire ;
- les personnes étudiant le système travaillent à distance les une des autres ou du système.

2.1.2 Modélisation

Le but de la simulation étant l'étude d'un système ou d'un phénomène, la première étape va consister à modéliser ce système ou phénomène, c'est-à-dire à créer une représentation plus ou moins simplifiée et abstraite de celui-ci. L'activité de modélisation consiste d'abord à comprendre le système. Un système constitue un groupe d'objets, il existe des interactions entre ces objets, et enfin il y a l'environnement avec lequel le système interagit. Il est donc nécessaire de fixer des limites à l'étude, le cadre expérimental, c'est-à-dire la spécification des conditions dans lesquelles le système est observé ou dans lesquelles on l'expérimente. Ce processus est centré-observateur [6] et pour un même système le modèle sera différent selon l'observateur, le point de vue de celui-ci et les questions qu'il se pose. Le modèle ainsi créé doit être validé afin de certifier que les réponses obtenues en travaillant sur le modèle seront les mêmes que celles qui auraient été obtenues en travaillant sur le système réel. Cette validation s'obtient en collectant des données sur le système et en comparant les résultats fournis par le modèle. Comme l'indique la figure 2.3 l'évolution du système peut se faire de manière continue ou bien par changement d'états discrets. De même, certaines dimensions comme l'espace et le temps peuvent

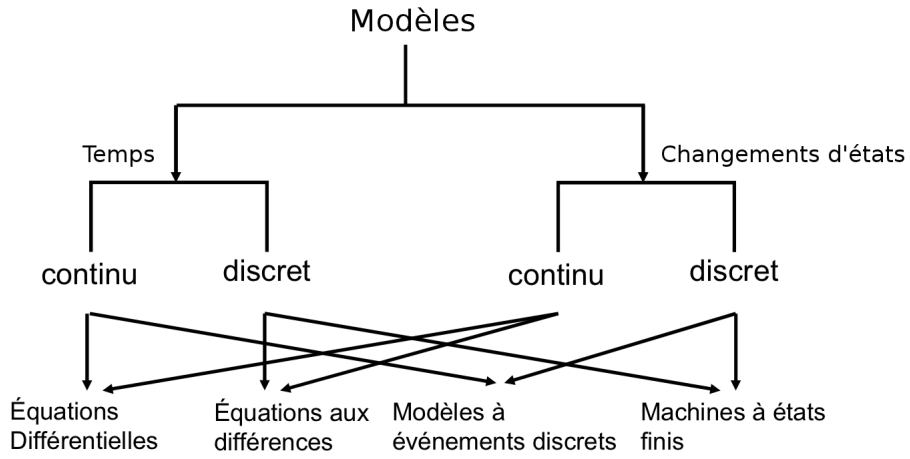


FIGURE 2.3 – Les types de modèles

être discrétisées ou au contraire être traitées comme des grandeurs continues. Il résulte de ces choix différents paradigmes de modélisation et différentes manières de décrire formellement le modèle, par exemple par des équations différentielles ou par un automate à états finis [7].

2.1.3 Simulateur

Pour obtenir les résultats attendus de la simulation, il faut exécuter le modèle créé, avec différentes données comme les valeurs d'état initial, la durée d'exécution ou un objectif à atteindre. Le modèle est exécuté au sein du simulateur. Le simulateur ou logiciel de simulation est donc composé d'une version exécutable du modèle et d'un moteur d'exécution.

L'accomplissement de la simulation en ce qui concerne les résultats obtenus dépendent du simulateur et des outils qu'il offre, comme des fonctions de visualisation et d'analyse des données. Différents critères sont à prendre en compte pour choisir un simulateur. Une perspective *non fonctionnelle* peut orienter le choix sur un produit commercial ou libre, ou bien sur un simulateur qui fait référence dans la communauté scientifique. Un point de vue purement *technique* rentre bien évidemment en compte, avec la compatibilité avec les formalismes discrets ou continus, les capacités du simulateur à un déroulement pas-à-pas ou d'une traite, ses performances, la gestion des données (*big data*), etc. Le type de simulateur doit aussi être en adéquation avec la simulation. Il existe par exemple :

- les simulateurs non pilotés dans lesquels l'humain n'intervient pas au cours de la simulation ;
- les simulateurs pilotés ou interactifs, comme le cas d'étude de simulation de drones avec un pilote réel aux commandes.

Le moteur d'exécution peut aussi être générique et prendre en charge des modèles exécutables, car écrits dans un langage de description formel comme VHDL (*VHSIC Hardware Description Language*), ou au contraire le simulateur peut être spécifique dans le cas de modélisation de certaines normes qui sont exprimées en langage naturel.

2.1.4 Validation et vérification

Deux processus vont permettre de qualifier la simulation [8]. La validation est le processus qui détermine si le modèle conceptuel est une représentation fidèle du système à analyser. La validation sert à *créer le modèle correct*. La vérification est le processus permettant de déterminer si le programme de simulation fonctionne comme attendu. La vérification sert à *exécuter le modèle correctement*.

2.1.5 Données de simulation

Dans le but de répondre à des questions, la simulation nécessite des données et génère elle-même des informations. Le projet ITCS (Infrastructure Technique Commune de Simulation) de la DGA [9], présenté lors d'une conférence de l'OTAN, établit trois groupes de données :

- les données caractérisant ce que l'on veut simuler (le scénario)
- les données liées aux outils utilisés pour cette simulation (les modèles)
- les données produites par ces simulations (les résultats)

Ces données sont parfois issues d'un processus d'élaboration long et il est important d'avoir un cadre permettant le stockage et la réutilisation de ces informations. La réutilisation ne concerne pas seulement les résultats ou les modèles, mais aussi le scénario qui est fondamental pour, par exemple, tester la non-régression d'un modèle. Le stockage des données doit donc se faire de manière structurée et inclure toutes les informations utiles à la réutilisabilité : principes de modélisation, métadonnées, modèle implémenté, résultats, etc.

2.1.6 Types de simulation/taxinomie

Pascal Cantot est ingénieur en chef des études et techniques de l'armement à la DGA dans le domaine de la modélisation et de la simulation. À travers son expérience et en effectuant la synthèse de divers standards et ouvrages académiques, il a constitué un glossaire de la simulation [10].

Il y définit plusieurs types de simulation :

- la simulation scientifique ou simulation numérique fermée, elle est purement logicielle et a des besoins en puissance de calcul, elle est non temps-réel et il n'y a pas d'opérateur humain dans la boucle. Il s'agit par exemple de l'écoulement d'un fluide autour d'un solide ;
- simulation hybride (intégration de matériel réel dans un environnement simulé) servant typiquement à des fins d'essai de matériel (qualification, test) ;
- simulation constructive (logicielle, fait intervenir le facteur humain de façon simulée) comme l'optimisation d'une chaîne de production ;
- simulation virtuelle (simulation pilotée, *man-in-the-loop*) fait intervenir des opérateurs réels dans un environnement simulé, comme un simulateur de vol ;
- simulation instrumentée (*live simulation*) qui intègre des matériels réels opérants dans un environnement physique réel, destinée à l'entrainement dans des conditions très réalistes (soldats dont les fusils sont munis d'émetteurs lasers).

Agostino G. Bruzzone est responsable de l'équipe du laboratoire de simulation du DIPTEM de l'université de Gênes. Dans son cours sur la modélisation et la simulation [11], il propose une classification suivant quatre caractères principaux qui peuvent être combinés. Selon lui, une simulation peut être :

– *stand-alone*, parallèle, distribuée ou fondée sur le web ;
– à événements discrets, continue, combinée ou hybride ;
– déterministe, stochastique ou *man-in-the-loop* ;
– à temps réel, "à temps lent", quasi temps réel ou "à temps rapide".

2.2 Multisimulation

2.2.1 Définition et intérêts

La simulation permet de répondre à des questions posées sur un système en créant une simplification de ce système, le modèle, et en l'exécutant dans un simulateur. Malgré cette simplification, la simulation ne permet pas toujours d'obtenir les résultats attendus. Il est très difficile d'établir un modèle reproduisant fidèlement un système complexe, car les composants sont nombreux, s'influencent mutuellement et concernent des domaines variés. L'élaboration d'un tel modèle requiert une expertise multidisciplinaire. Il existe souvent des modèles déjà établis pour chacun des composants du système complexe, modèles ayant déjà été éprouvés dans leurs domaines respectifs. La multimodélisation consiste à établir les correspondances nécessaires aux frontières des différents modèles afin d'obtenir un modèle global cohérent et une vue riche du système, ce qui permettra d'évaluer les interactions et les influences mutuelles. La multisimulation consiste alors en l'interconnexion de simulateurs exécutant chacun un des modèles associés de la multimodélisation. La capacité à faire fonctionner ensemble plusieurs simulateurs peut également être exploitée pour faire fonctionner ensemble plusieurs instances d'un même simulateur, s'exécutant sur des machines séparées : cette exécution distribuée peut parfois permettre de résoudre des problèmes de passage à l'échelle, lorsqu'un réel travail de parallélisation n'est pas possible. Enfin, la cosimulation consiste à faire collaborer des éléments réels avec des éléments simulés.

Dans le cas de la multisimulation, il faut effectuer une mise en relation des différents logiciels de simulation, c'est-à-dire établir des niveaux de correspondances à la fois sémantique et logicielle.

2.2.2 Niveaux de correspondance

Correspondance sémantique des modèles (multimodélisation)

Afin de pouvoir interagir, les modèles des différentes simulations doivent posséder une sémantique commune, le niveau minimum étant situé à leurs frontières. TENA (*Test and Training Enabling Architecture*) a été conçue par le *Department of Defense (DoD)* pour assurer l'interopérabilité des systèmes d'entrainement, y compris les simulations. Dans le document présentant cette architecture [12], l'*interopérabilité sémantique* est l'exigence la plus importante pour l'interopérabilité. Cette *interopérabilité sémantique* est construite sur un langage et un contexte commun de communication. Elle inclut non seulement une correspondance au niveau de la description des objets et des interactions, mais aussi une représentation commune des données. Un *bosquet* par exemple peut être représenté en tant qu'objet unique, d'un seul volume et opaque dans un modèle et comme un ensemble d'arbres au travers desquels on peut voir dans un autre modèle. Lors de l'interconnexion de deux modèles utilisant cet objet, des soldats situés derrière le *bosquet* dans le premier modèle et pensant être bien cachés seront clairement désavantagés face à un char évoluant

dans le deuxième modèle. Le projet ITCS [9] ainsi que TENA [13] recommandent le standard SEDRIS (*Synthetic Environment Data Representation Interface Specification*) [14] pour la représentation et la spécification des données.

De même, il existe une ontologie pour l'informatique ambiante [15] et de nombreux travaux de recherches tentent d'apporter des réponses à la question de la mise en correspondance automatique de modèles hétérogènes. Cependant, dans le cadre de ce stage, cette problématique a été écartée afin de s'intéresser plutôt à la correspondance des simulateurs.

Correspondance logicielle des simulateurs (multisimulation)

Les simulateurs sont les moteurs d'exécution logiciels des modèles. L'interconnexion de ces moteurs doit se faire selon certaines correspondances logicielles, comme la communication, la politique d'exécution ou bien le type de synchronisme [16]. Faire collaborer des logiciels implique d'établir un moyen de communication entre ces logiciels. On peut envisager une connexion de manière ad hoc, mais cette méthode trouvera vite ses limites si le nombre de simulateurs connectés est grand. On peut aussi utiliser un bus de communication, appelé bus de simulation [17], si les simulateurs sont prévus pour. Dans tous les cas, il faut tenir compte de la localisation des simulateurs (sur la même machine ou sur des machines distantes), de la quantité de données échangées et des niveaux de la communication utilisée (niveau applicatif ou bas niveau). Ces critères vont influer sur les temps de communication, la bande passante nécessaire du support et sur le compromis surcoût en temps/flexibilité d'utilisation. L'exécution de la simulation peut être dirigée par le temps ou par les évènements. Dans le premier cas, le temps de simulation est une séquence de pas de tailles égales, et la simulation est cadencée par une horloge qui dirige l'évolution de pas en pas. Dans le second, il n'est simulé que les dates associées à l'occurrence d'un évènement. Il n'y a pas d'horloge globale et le temps avance par saut de tailles variables, d'évènement en évènement à traiter. Le type de synchronisme concerne le respect de la causalité. Le problème de la causalité, en simulation distribuée, représente l'incohérence qui résulte lorsqu'un simulateur tente d'influer sur le passé d'un autre simulateur. Afin de résoudre ce problème, il existe deux approches de gestion du temps. L'approche conservative (ou pessimiste) permet de toujours respecter la contrainte de causalité : les simulateurs s'attendent les uns les autres avant de faire avancer leur temps de simulation. L'inconvénient est que le temps d'exécution est tributaire du simulateur le plus lent à chaque pas. Dans l'approche spéculative (ou optimiste), chaque simulateur s'exécute jusqu'à ce que la contrainte de causalité soit enfreinte. Dans ce cas un retour en arrière par un mécanisme de *rollback* est nécessaire et l'exécution doit recommencer en tenant compte de la donnée ayant provoquée le problème de causalité. Cela nécessite de stocker l'état des simulateurs en tout instant et introduit un coût temporel lors d'un *rollback*.

2.2.3 Accréditation

La multisimulation implique une modélisation hétérogène, et donc plusieurs *sous-modèles* qui doivent être conçus, exécutés et validés séparément. Cependant, la validation et la vérification individuelle des sous-systèmes ne permettent pas de conclure que la simulation globale est validée et vérifiée [8]. Il est nécessaire de conduire des tests et des expériences et de remplir une procédure spécifique d'accréditation afin de garantir ce résultat à l'ensemble de la simulation.

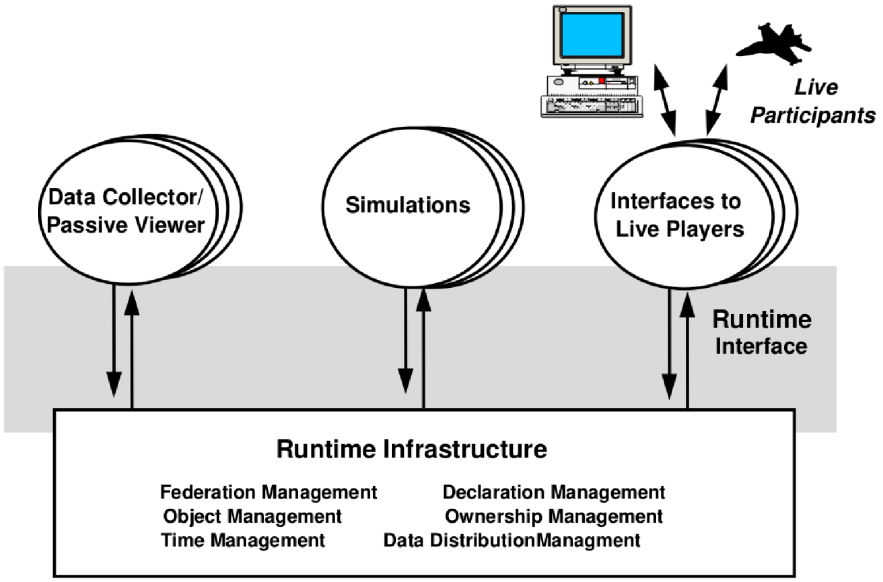


FIGURE 2.4 – Vue fonctionnelle d'une fédération HLA, source *DoD*

Le *SISO* travaille sur l'élaboration d'un standard pour la validation, vérification et l'accréditation des modèles, des simulations et des données [18].

2.2.4 Cadres pour la multisimulation

La multisimulation consiste donc à interconnecter plusieurs simulations indépendantes. Cette interconnexion peut être effectuée dans un cadre particulier à la multisimulation ou en utilisant des solutions non spécifiques.

Cadres dédiés

High Level Architecture (HLA)

Présentation HLA a été développée par le *Department of Defense (DoD)* afin de supporter une large variété de simulations. HLA fait suite aux solutions orientées militaires DIS et ALSP. C'est une architecture technique développée pour faciliter la réutilisabilité et l'interopérabilité des simulations. La figure 2.4 illustre ses principaux composants.

Correspondance logicielle HLA définit un ensemble de simulations appelé *fédération* [19]. Les participants (*fédérés*) sont aussi bien des simulations que des interfaces pour les participants humains ou des systèmes de monitoring et de collectes de données. Tous les fédérés doivent permettre à leurs propres objets d'interagir avec les objets d'autres fédérés, par des services communs de communication (RTI).

La *RTI* (Run-Time Infrastructure) est une interface commune de communication pour la fédération. Toutes les interactions entre les fédérés doivent se faire via la RTI. Les *interfaces d'exécution* permettent aux fédérés d'interagir avec la RTI, comme invoquer des services ou répondre aux requêtes de celle-ci.

Nom	Éditeur	standard	Interface	Licence
MÄK High Performance	MÄK Technologies	1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)	C++, Java	Commercial
Pitch pRTI	Pitch Technologies	1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)	C++, Java	Commercial
CAE RTI	CAE Inc.	1.3, IEEE 1516	C++	Commercial
Chronos RTI	Magnetar Games	IEEE 1516	C++, .NET	Commercial
CERTI	ONERA	1.3 partiel, IEEE 1516 partiel	C++, Matlab, Fortran90, Python, Java	GPL
PORTICO	Littlebluefrog labs	1.3, IEEE 1516	Java, C++	CDDL
OpenHLA		1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)	Java	Apache License

FIGURE 2.5 – Hétérogénéité des RTI

Correspondance sémantique La *spécification d'interface* décrit les services fournis par la RTI aux fédérés, et par les fédérés à la RTI. Les *modèles d'objet* HLA sont les descriptions des objets partagés d'une simulation (SOM) ou d'une fédération (FOM). L'interopérabilité est assurée par l'utilisation de patrons (OMT). Ces modèles d'objet permettent de garantir la correspondance sémantique de la multisimulation.

Limitations Bien que la volonté à l'origine de HLA soit l'interopérabilité et la réutilisabilité, il existe des facteurs limitant l'utilisation de cette architecture.

HLA a été définie par plusieurs standards et normes depuis sa création. La première version issue du *Department of Defense (DoD)* est la 1.3, puis l'IEEE a standardisé les versions 1516-2000 et 1516-2010. Parallèlement aux standards IEEE, le SISO (*Simulation Interoperability Standards Organization*) a publié une spécification complémentaire appelée *HLA Evolved*. HLA a été aussi sujet d'un accord de normalisation de l'OTAN sous la référence STANAG 4603. Ce foisonnement de versions amène à des incompatibilités entre ces standards.

HLA est une spécification, les logiciels l'implémentant sont appelés RTI et il existe de nombreuses déclinaisons suivant le standard supporté, le langage utilisé pour les interfaces d'exécution et le type de licence. La figure 2.5 illustre l'hétérogénéité entre plusieurs RTI. De plus, comme le décrit le mémorandum technique du ministère de la Défense canadien [20], les RTI ne sont pas interopérables et le processus de sélection d'un RTI pour une infrastructure de multisimulation doit tenir compte des possibilités d'évolution et de compatibilité.

Il est aussi nécessaire d'utiliser des simulateurs compatibles HLA. Dans le cas d'une simulation déjà existante, cette compatibilité du simulateur n'est pas assurée. C'est le cas par exemple de Matlab/Simulink et OMNet++, deux simulateurs qui seront détaillés dans la section 4. Il existe des greffons pour ces simulateurs, mais ils utilisent des RTI différents (PORTICO pour OMNet++ et CERTI pour Matlab/Simulink) et ne sont pas interopérables.

Il apparait aussi que HLA est complexe à mettre en œuvre. Pour réaliser ce

projet de multisimulation, j'ai envisagé dans un premier temps l'utilisation d'ESCADRE, un *framework* de simulation compatible HLA et développé par la DGA [21]. Après avoir effectué la demande de licence, j'ai été contacté par Dominique Canazzi, ingénieur-chef de projet pour ESCADRE, afin d'avoir des explications sur mon projet. Il est ressorti de cet entretien que ESCADRE était bien en totale adéquation fonctionnellement avec le projet, mais que la complexité de la mise en œuvre (formation, soutien indispensable de la part de la DGA, etc.) rendait son utilisation peu réaliste.

Il n'a été trouvé aucun exemple dans la littérature indiquant qu'il était possible de coupler des simulateurs s'intéressant à des aspects différents d'un même objet, comme le couplage d'un simulateur s'occupant des déplacements d'un drone avec un simulateur réseau gérant la réception du signal Wi-Fidu même drone en fonction de sa position.

On peut aussi remarquer que HLA est issue du domaine militaire et bien qu'étant devenue un standard IEEE, la plupart des documents traitants de cette architecture proviennent du monde militaire (ministères de la Défense [22], OTAN [19], DGA [9], etc.).

FMI FMI (*Functional Mock-up Interface*) est un standard permettant l'utilisation de modèles provenant de différents environnements de simulation dans d'autres environnements.

Présentation FMI est issu de MODELISAR, un projet ITEA2 (*Information Technology for European Advancement*) qui s'est déroulé de juillet 2008 à décembre 2011. MODELISAR comprenait 29 partenaires dont Dassault Systems, Daimler, Volkswagen ou encore Volvo. Le but de MODELISAR était l'amélioration significative de la conception des systèmes et logiciels embarqués dans les véhicules. FMI permet l'utilisation d'un modèle dans un autre environnement soit par un processus d'échange du modèle (*FMI for Model Exchange*) ou par cosimulation (*FMI for Co-simulation*) [23].

FMI for Co-simulation Le but est de fournir un standard d'interface permettant le couplage de deux (ou plus) outils dans un environnement de cosimulation. FMI définit une interface ouverte qui sera implémentée par un exécutable appelé FMU(*Functional Mock-up Unit*). Les fonctions FMI sont appelées par un outil de simulation pour créer une ou plusieurs instances de FMU, appeler un modèle et exécuter ce modèle. L'utilisation typique est la collaboration avec d'autres modèles, chaque simulation compose alors un sous-système. L'échange de données entre les sous-systèmes est restreint à des points discrets. Entre deux communications, les sous-systèmes sont indépendants les uns des autres. Un sous-système *maître* contrôle les communications et la synchronisation entre les sous-systèmes *esclaves*. Des informations concernant les sous-systèmes *esclaves* et leurs capacités à supporter des fonctions avancées sont fournies dans un fichier XML. Ces fonctions concernent par exemple l'utilisation de pas de temps variables entre deux communications. La cosimulation peut s'effectuer sur une machine unique ou, comme l'illustre la figure 2.6, dans une infrastructure distribuée. Cette configuration nécessite qu'un *middleware* (appelé *FMI co-simulation backbone*) en charge du réseau de communication entre le *maître* et l'*esclave* soit installé sur ces machines.

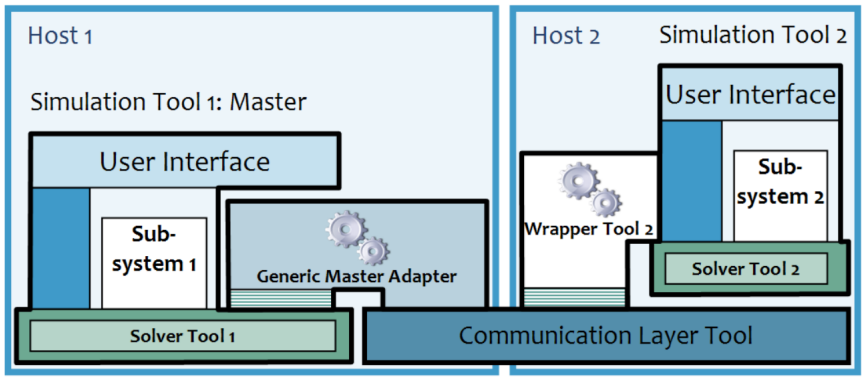


FIGURE 2.6 – Cosimulation FMI dans une infrastructure distribuée, source Modelisar

Un sous-système, suivant ses capacités, peut assumer le rôle de *maître* ou d'*esclave*. Une liste d'outils compatibles avec FMI est disponible sur le site internet de MODELISAR. Cette liste détaille les capacités de chacun à être *maître* ou d'*esclave*, comme à importer ou exporter des modèles dans le cas de l'utilisation de *FMI for Model Exchange*. Elle comprend des logiciels de modélisation fondés sur Modelica (Dymola de Dassault Systèmes, JModelica de Modelon, SimulationX de ITI, etc.), des environnements de simulations comme CosiMate de ChaisTek et des *toolbox* ajoutant la compatibilité FMI à Matlab/Simulink (FMI toolbox de Modelon, Real-Time Workshop via Dymola). Contrairement à HLA, FMI a été conçu et est utilisé par le monde industriel. Initialement destiné au secteur automobile, FMI commence à intéresser d'autres domaines de l'industrie utilisant des simulations, comme le domaine des SmartGrids. Une évolution rapide vers une architecture plus générique paraît possible et elle pourrait par conséquent répondre à certaines problématiques de notre projet. On peut cependant craindre qu'elle ne devienne aussi complexe et difficile à utiliser qu'HLA dans ce processus.

Solutions non dédiées

En dehors de cadres spécifiques conçus dans ce but, il est possible de faire collaborer des simulateurs de différentes manières.
Les simulateurs peuvent être couplés par un *middleware* de communication ou un bus logiciel, par exemple :

- DDS (*Data Distribution Service*) est un standard de l'OMG pour la communication et la distribution de données [24]. DDS offre de meilleures performances que HLA lors d'une mise à l'échelle [25] et des travaux sont en cours afin de réaliser une plate-forme de multisimulation avec DDS [26] [27] [28].
- CORBA, avec notamment l'utilisation de TAO (*The ACE ORB*) qui est orienté temps réel et performances [29].
- MSI (*Multi-Simulator Interface*) [30] est une version allégée de HLA. Elle a été développée par les laboratoires ATL de Lookheed Martin [31] pour les besoins de leur logiciel maison de simulation CSIM.

Malheureusement, aucune de ces solutions n'apporte une réponse simple et universelle correspondant à notre projet.

Il est aussi possible d'effectuer des couplages deux à deux, de manière ad hoc. Dans tous les cas, il est nécessaire de développer une interface d'intermédiation :

pour s'accrocher au bus logiciel ou bien directement à l'autre simulateur. Il existe des outils permettant de faciliter ce travail :

- CODIS [32] est un *framework* permettant la génération automatique d'interfaces lors de couplage de systèmes hétérogènes continu/discret. Un couplage entre Matlab/Simulink et SystemC a été réalisé comme cas d'étude.
- ModHel'X [33] est un *framework* pour la simulation de modèles hétérogènes.

Lorsque les simulateurs visés ne permettent pas l'utilisation des outils, le développement spécifique de code est nécessaire. Ce travail est parfois délicat, prend du temps et la maintenance du système est difficile.
Des solutions plus génériques effectuant du *sandboxing* avec les simulateurs peuvent aussi être envisagées, mais il n'a été trouvé aucune référence à ces solutions dans la littérature.

2.2.5 Projets existants de cosimulation

La cosimulation est une solution pour l'évaluation des espaces intelligents et de l'informatique ambiante. Il sera fait une présentation de l'existant dans un premier temps. Les communications réseau constituant un élément important dans ces environnements, des projets de multisimulation intégrant un simulateur réseau seront détaillés dans un second temps.

Cosimulations ad hoc pour l'évaluation d'espaces intelligents

Il existe de nombreux travaux dans le domaine de la simulation des espaces intelligents et de l'informatique ambiante, voici une liste de solutions ayant été implémentées :

- UbiWise des laboratoires HP (2002-2003) a mené au développement d'un simulateur proche d'un jeu vidéo pour tester la perception de l'utilisateur d'un environnement virtuel ;
- le projet UbiREAL du Nara IST et de l'université de Shiga (2006) a consisté à la création d'un monde virtuel en 3D où des appareils simulés pouvaient interagir avec des appareils réels ;
- le projet SitCom d'IBM (2008) a permis de concevoir un *toolkit* pour le développement de services ayant une connaissance du contexte (*context aware*) ;
- le projet ViSi de l'université de Groningen a mené au développement d'un *middleware* orienté services pour l'informatique ubiquitaire ;
- le projet SHSim (Smart Home Simulator) de l'université Tsinghua de Pékin (2009) permet l'installation dynamique de services et d'appareils virtuels dans un espace intelligent.

Ces projets sont issus aussi bien du monde industriel qu'académique et ils permettent de voir qu'il est possible d'obtenir un environnement virtuel ayant une connaissance du contexte, mélangeant des objets réels et simulés, et sur lequel l'humain peut agir. Ce sont des points qui sont nécessaires à une solution d'évaluation par simulation. Ces projets restent néanmoins isolés les uns des autres et ils ne permettent pas d'obtenir une solution complète d'évaluation.

Projets intégrant un simulateur réseau

Projet iTETRIS iTETRIS [34] est un projet financé par la commission européenne dans le cadre du *7th Framework Programme*.

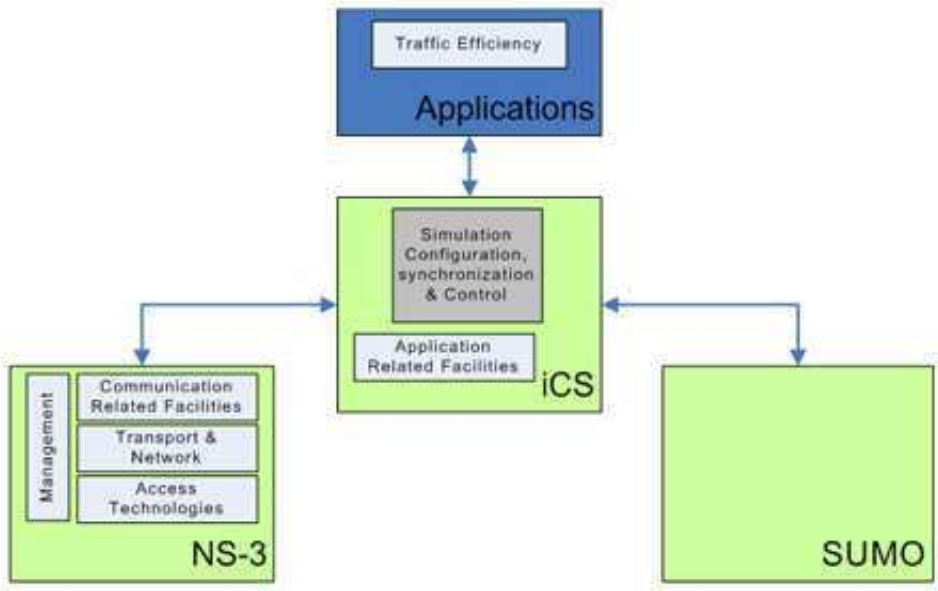


FIGURE 2.7 – L'architecture iTETRIS

Le but de iTETRIS est de créer une plate-forme de simulation de trafic de véhicule routier. Pour cela, le simulateur réseau NS-3 est couplé à SUMO (*Simulation of Urban MObility*) via un module central appelé iTETRIS Control System (iCS) comme l'illustre la figure 2.7.

L'étude de cette plate-forme permettrait de comprendre comment le couplage avec NS-3 a été effectué. Malheureusement le code n'est pas disponible, et je n'ai pas eu de réponses à mes demandes de renseignements et de livrables.

Thèse de master *Integration of NS-3 with MATLAB/Simulink* La thèse de Dmitry Kachan [35] de l'université de Luleå en Suède porte sur le couplage du simulateur réseau NS-3 et de Matlab/Simulink. Le couplage est réalisé par la mise en réseau de deux simulateurs. Il est nécessaire d'injecter des modules dans NS-3 afin de modifier la gestion du temps de celui-ci. Les simulateurs ne s'exécutent pas de manière simultanée : l'un exécute une tâche pendant que l'autre est en position d'attente, puis se déroule une phase de communication, etc.

HLA-OMNeT++ Emanuele Galli de l'université de Rome a développé un simulateur réseau compatible HLA [36]. Ce simulateur est fondé sur l'utilisation de OMNet++ et il utilise le RTI PORTICO comme implémentation de HLA. Ce simulateur utilise d'anciennes versions de OMNet++ et de PORTICO et ne fonctionne pas avec les dernières versions de ces logiciels.

Veins Le projet de recherche Veins (*Vehicles in Network simulation*) a abouti sur le *framework* éponyme [37], un logiciel *open-source* de simulation de communication intervéhicule. Comme l'illustre la figure 2.8, il est composé du simulateur OMNet++ et de SUMO. Le framework MiXiM est nécessaire afin que OMNet++ puisse supporter les réseaux sans fil et mobiles.

Le code de VEINS est disponible et il existe un tutoriel d'installation ainsi qu'une FAQ sur le site. La documentation fournie ne concerne que l'utilisation du logiciel

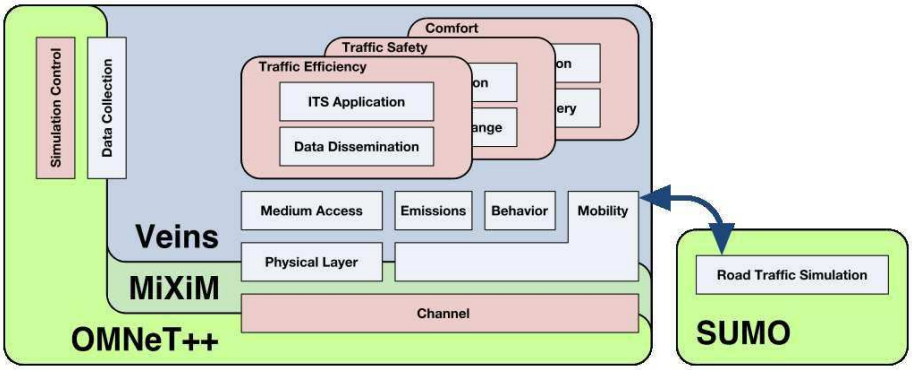


FIGURE 2.8 – L'architecture de Veins

et il n'est pas expliqué les moyens employés pour faire collaborer les simulateurs. L'étude du code sera nécessaire pour déterminer comment le couplage a été effectué.

Solution COSMO COSMO est une solution de cosimulation pour les réseaux sans fil d'intérieur [38] issue des travaux conjoints de personnels du Royal Institute of Technology de Stockholm et de l'université de Zhejiang. Cette solution utilise Matlab et le simulateur réseau OMNet++ mais elle ne réalise pas de manière effective un couplage de simulateurs. Le réseau sans fil est découpé en couches, elles-mêmes subdivisées en deux tâches différentes : *computation tasks* exécutées par Matlab et *control and communication tasks* par OMNeT++. Pour finir les modèles Matlab issus des *computation tasks* sont intégrés en tant que bibliothèques partagées dans OMNeT.

2.2.6 Bilan des solutions de multisimulation

La simulation et la multisimulation sont nécessaires à l'évaluation d'espaces intelligents qui constituent des systèmes complexes avec de nombreuses interactions. Il existe des solutions de multisimulation utilisant un cadre dédié comme HLA ou FMI. Il est aussi possible d'utiliser un bus logiciel ou d'effectuer un couplage ad hoc des simulateurs. Ces solutions sont néanmoins complexes, coûteuses en terme de temps et de personnel, rarement interopérables et finalement difficiles à maintenir. Il y a une nécessité à obtenir des solutions plus légères et réutilisables.

Deuxième partie

Définition et spécification d'un démonstrateur de multisimulation

Chapitre 3

Exemple de scénario visé

Le cas d'étude pour le développement d'une plate-forme de multisimulation sera fondé sur des drones Parrot. Ces drones sont des quadricoptères contrôlables par un réseau sans fil Wi-Fi. Le scénario voulu sera d'effectuer des vols en formation (*flocking*) composée de drones réels et de drones simulés. La coordination du vol de ces drones sera effectuée à l'aide d'information d'état provenant du réseau, comme la force du signal Wi-Fi reçu.

Les besoins pour réaliser ce scénario sont l'utilisation d'un simulateur de réseaux, d'un simulateur de drones et d'une application permettant le contrôle de drones, réels ou virtuels.

Chapitre 4

Simulateurs nécessaires

4.1 Simulateurs de réseaux

Un simulateur de réseaux est employé pour modéliser des réseaux informatiques, par exemple avant qu'ils ne soient déployés dans le monde réel. La simulation permet dans ce cas de comparer les performances de multiples configurations, et ainsi d'évaluer et résoudre des problèmes de fonctionnement. Cette méthode permet d'éviter des tests sur le terrain qui peuvent être potentiellement coûteux en terme de prix, de temps, ou de gêne occasionnée pour les utilisateurs. Les simulateurs réseaux sont aussi utilisés dans le milieu de la recherche, afin de connaître le comportement de protocoles nouvellement développés, comme l'amendement *IEEE 802.11p* (*WAVE, Wireless Acces in Vehicular Environnments*) du standard *IEEE 802.11* qui ajoute le support d'applications pour les systèmes de transports intelligents (*ITS*).

Ces simulateurs sont généralement fondés sur une exécution à évènements discrets. Les évènements sont stockés dans une file d'attente, ils sont triés par ordre chronologique et ils peuvent eux-mêmes générer un ou plusieurs autres évènements. Le temps de simulation avance jusqu'au *timestamp* du prochain évènement avant de l'exécuter.
Il existe de nombreux simulateurs, en version open source comme NS-3, OMNeT++, JiST/SWANS [39] ou bien des outils commerciaux comme OPNET.
Dans le cadre du démonstrateur, le simulateur devra pouvoir supporter les réseaux sans fil, les modèles de mobilité et les connexions ad hoc.

Une partie de ma mission a donc consisté à effectuer un choix parmi les différentes solutions existantes dont les principales caractéristiques sont présentées dans les sections suivantes.

4.1.1 OMNeT++

OMNeT++ est un *framework* et une bibliothèque de simulation orientée objet, en langage C++, modulaire et extensible (*Objective Modular Network Testbed in C++*) [40]. Il est disponible pour les systèmes d'exploitation Linux, Windows et Mac OS. Il a été développé par András Varga et il bénéficie du soutien d'une grande communauté qui met à disposition de nombreux modèles de simulation.

OMNeT++ est gratuit dans le cadre d'un usage académique ou non commer-

cial. Une version commerciale nommée *Omnest* est disponible et est utilisée par de grandes compagnies de l'industrie comme IBM, CISCO, Intel, Alcatel-Lucent, Thales, Orange. Cette version, outre l'accord d'une utilisation commerciale, apporte un support client dédié, une meilleure gestion de Windows et le support de HLA.

Description

OMNeT++ repose sur la programmation orientée objet, l'utilisation de modules et l'emploi d'un langage de description appelé *NED (Network Description)*.
Les composants de base sont appelés *simple modules*. Ils sont implémentés en C++ et ils sont contenus dans des bibliothèques de classes. Ils peuvent être utilisés tels quels ou bien être combinés pour créer de modules complexes nommés *compound module*. Des fichiers *NED* servent à décrire aussi bien les modules que le réseau dans son ensemble, par imbrication de sous modules. Les paramètres et la configuration de la simulation se font grâce à un fichier texte obligatoire appelé *omnetpp.ini*. Il est possible à partir de ce fichier de configurer plusieurs simulations avec différents paramètres à partir de la même structure de réseau, tout en n'ayant qu'à compiler qu'une seule fois. Les messages échangés entre les modules peuvent utiliser la classe de message de base (*cMessage)* ou alors être décrits dans des fichiers optionnels *msg*. La figure 4.1 illustre l'organisation des différents modules et fichiers nécessaires à une simulation.

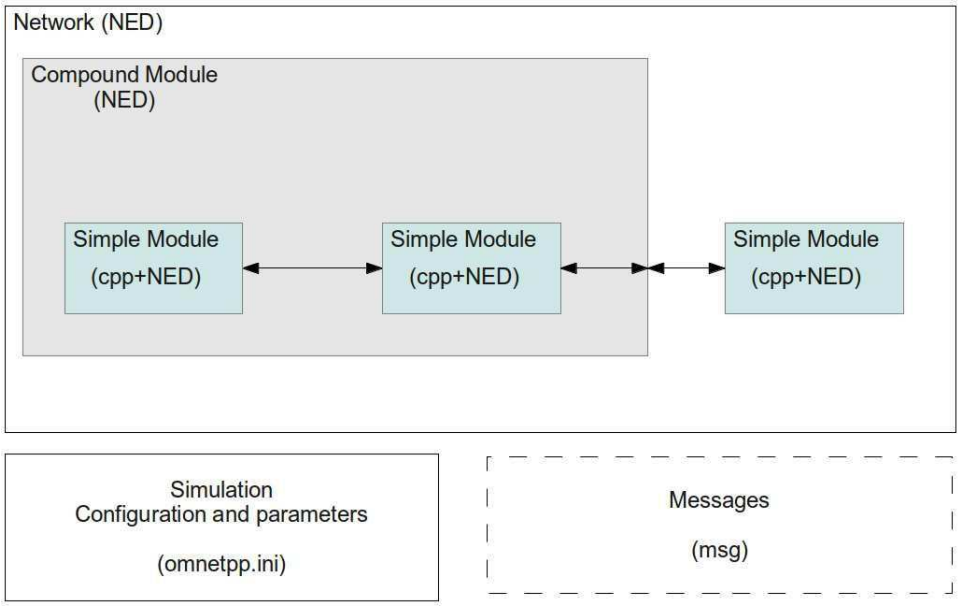


FIGURE 4.1 – Les composants d'OMNeT++.

L'exécution de la simulation peut s'effectuer en ligne de commandes (*CMDenv*) ou dans une interface graphique (*Tkenv*). *CMDenv* est une interface rapide conçue principalement pour exécuter des *batch*. *Tkenv* est en environnement permettant une exécution interactive de la simulation, l'utilisateur pouvant modifier les paramètres de la simulation et visualiser la représentation de réseau en fonctionnement. Il est conseillé de limiter l'utilisation de *Tkenv* lors des phases de développement ou pour des présentations, les animations graphiques ralentissant fortement la simulation. La

figure 4.2 illustre un exemple d'exécution avec *Tkenv*.

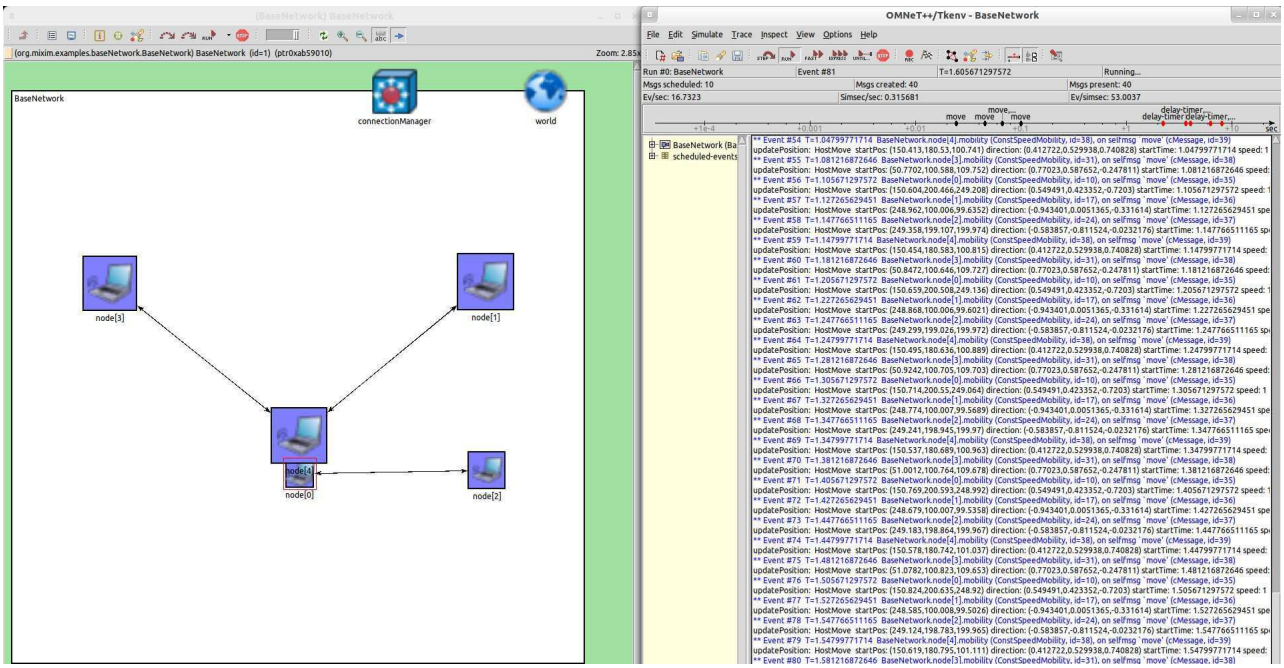


FIGURE 4.2 – Execution d'une simulation dans l'environnement graphique.

OMNeT++ permet de collecter et d'analyser les résultats de la simulation, sous forme scalaire ou vectorielle, en spécifiant les paramètres de l'enregistrement dans les modules de la simulation. Les fichiers scalaires contiennent des statistiques calculées sur l'ensemble de la simulation comme le nombre de paquets envoyés ou perdus, le débit crête, etc. Les fichiers vectoriels capturent le comportement au fil de l'exécution en associant un *timestamp* à une donnée, comme l'évolution de la longueur de la file d'attente. Ces données peuvent être visualisées dynamiquement lors de l'exécution de la simulation dans *Tkenv* ou être analysées en différé.

OMNeT++, actuellement à la version 4.2.2, est maintenu à jour régulièrement. La version 4 est une révision majeure du logiciel. Un *IDE (Integrated Development Environnment)* fondé sur Eclipse a été ajouté, des changements fondamentaux dans presque tous les composants ont été effectués et le langage *NED* a été révisé et étendu.

OMNeT++ dans sa version de base comprend l'essentiel des composants afin d'effectuer une simulation d'un réseau. Il existe différents *frameworks* complémentaires pour modéliser des réseaux plus complexes comme les réseaux sans fil, la propagation des ondes ou la mobilité de nœuds :

INET

Le *framework* INET contient les modèles pour de nombreux protocoles réseaux.Parmi d'autres, on peut citer les protocoles UDP, TCP, IP, le standard 802.11, MPLS, OSPF, etc. INET apporte le support pour les communications sans fil et mobiles, cependant les protocoles pour les communications ad hoc ne sont pas gérés.

INETMANET

Ce *framework* est fondé sur INET, il apporte des fonctionnalités supplémentaires pour les communications sans fil et mobiles en mode ad hoc (MANET signifie *Mobile Adhoc Network*), comme des protocoles de routage mobiles, des modèles de propagation et de mobilité.

MiXiM

Ce *framework* est destiné aux réseaux fixes ou mobiles sans fil et propose des modèles détaillés de propagation des ondes, d'interférences, de protocoles sas fil MAC ou de consommation d'énergie. MiXiM est centré sur la modélisation des *NICs (Network Interface Controller)* et un inconvénient est qu'il ne modélise pas les couches hautes comme la pile IP. Il existe néanmoins un module appelé *Mixnet* permettant d'utiliser ensemble MiXiM et INET afin de permettre par exemple l'utilisation d'un *NIC* de MiXiM avec les protocoles TCP/IP d'INET.

La figure 4.3 illustre les différences de modélisation d'un nœud du standard IEEE 802.11 entre MiXiM et INETMANET, celui-ci prenant en charge la couche transport.

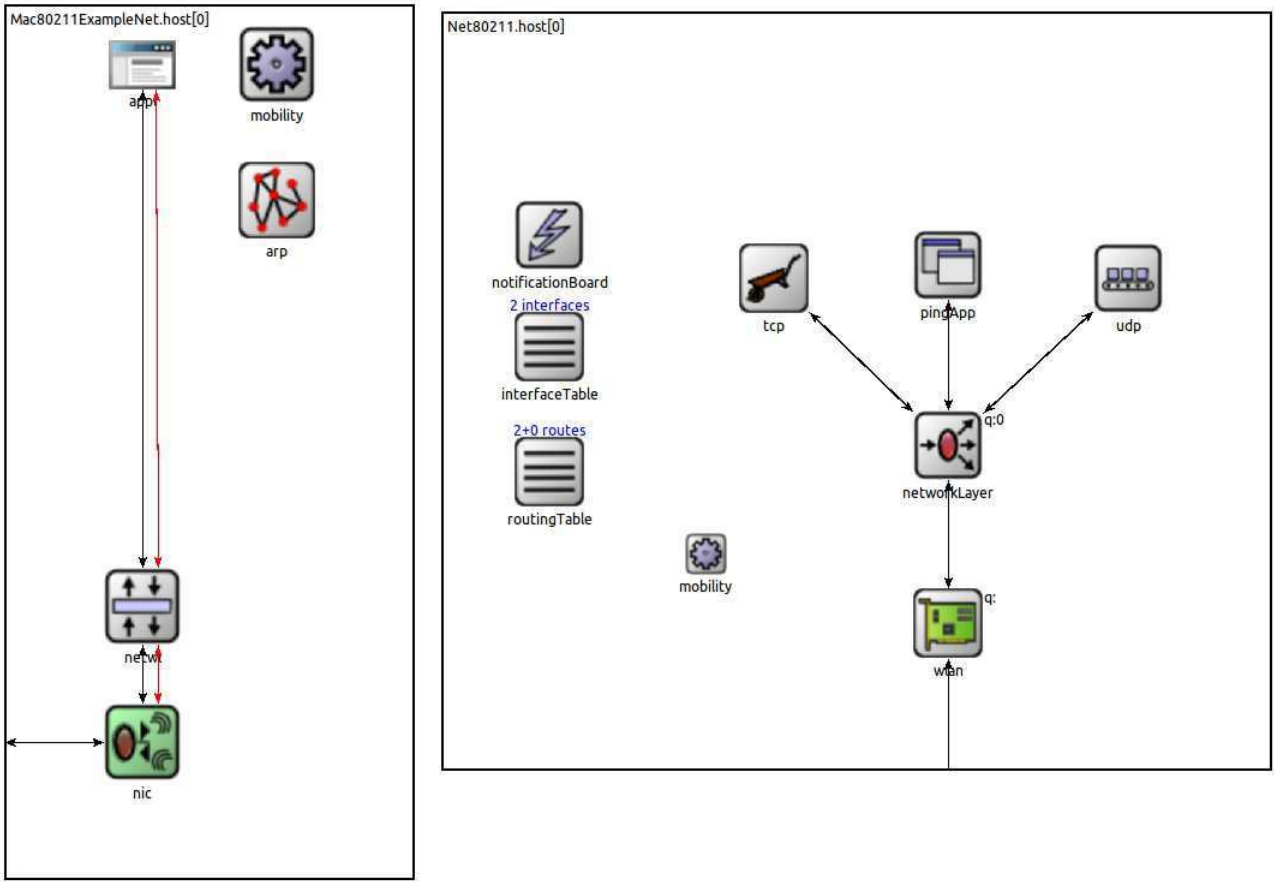


FIGURE 4.3 – Comparaison entre les composants d'un même noeud avec MiXiM (à gauche) et avec INETMANET (à droite).

OMNeT++ est un simulateur réseau gratuit pour une utilisation non commerciale, et il est soutenu par une large communauté. La documentation, intégrée dans l'IDE, comporte un tutoriel, un guide de développement en C++ et l'ensemble des classes de l'API est décrit par un document *doxygen*. Les projets HLA-OMNeT++ et Veins, décrits à la section 2.2.5, montrent les possibilités de couplage de OMNeT++ avec un autre simulateur. Cependant, HLA-OMNeT++, qui utilise une version antérieure d'OMNeT++, n'est pas compatible avec la dernière version d'OMNeT++ à cause des changements profonds de la version 4. Veins fonctionne avec le *framework* MiXiM qui, comme les autres *frameworks* cités précédemment, est en constante évolution. Les mises à jour et les corrections de *bugs* sont effectuées sur *github* par la communauté, et la documentation est obsolète ou très succincte, ce qui amène une grande difficulté de développement.

Une autre caractéristique intéressante d'OMNeT++ est sa capacité à fonctionner, pour la taille du réseau envisagé dans ce projet, en « temps réel » et à s'interfacer avec des éléments réels (émulation, *hardware-in-the-loop*).

OMNeT++ est le simulateur réseau qui a été retenu pour la réalisation de ce projet car il permet la simulation de réseaux Wi-Fi comprenant des nœuds mobiles, comme le sont des AR.Drones. Il est de plus possible de s'appuyer sur le projet Veins pour le couplage avec un autre simulateur.

4.1.2 NS-3

NS-3, pour *Network Simulator* version 3, est un logiciel libre de simulation à événements discrets [41]. Ce simulateur est un dérivé de REAL et son développement a commencé en 1989. Il est actuellement maintenu par une communauté de volontaires. NS est principalement utilisé pour la recherche ou l'enseignement en fournissant un environnement de simulation ouvert.

NS-3 est une bibliothèque fournissant un *core* de simulation et un ensemble de modèles. Cette bibliothèque est écrite en C++, il existe cependant des *bindings* Python et l'utilisateur peut donc interagir avec la bibliothèque à partir d'une application C++ ou Python, comme l'illustre la figure 4.4.

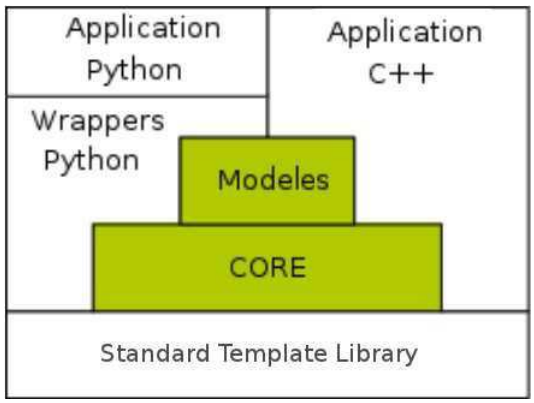


FIGURE 4.4 – Composition de NS-3.

NS-3 supporte les réseaux fondés sur IP ou non, bien que la majorité des utili-

sateurs l'utilise pour des simulations de réseaux sans fil avec IP, comme les réseaux Wi-Fi, WIMAX, LTE (couches 1 et 2), avec des protocoles de routage statique ou dynamique comme OLSR ou AODV. NS-3 comprend un *scheduler* temps réel autorisant l'interaction de simulation avec des systèmes réels. Une bibliothèque de classes *mobility* apporte le support de la mobilité des nœuds.

Bien qu'étant un logiciel gratuit, bien documenté et bénéficiant d'une large communauté, NS-3 souffre de quelques défauts. La version antérieure, NS-2, a été très populaire et a été largement utilisée. Malheureusement, à cause d'un développement entièrement en C++ de NS-3, les deux versions sont incompatibles et les modèles NS-2 doivent être redéveloppés pour fonctionner avec NS-3. Aussi, et contrairement à OMNeT++, il a été fait le choix de ne pas intégrer un IDE au logiciel. Le processus de conception d'une simulation consiste à coder en lignes de commandes et à inclure des outils de configuration et de visualisation à la demande. Le projet iTETRIS, décrit à la section 2.2.5, effectue un couplage entre NS-3 et SUMO. Cependant, aucune documentation ni code source ne sont disponibles pour étudier la manière utilisée. De plus, il n'existe pour l'instant pas de compatibilité avec HLA bien qu'un projet du GSOC 2012 (*Google Summer of Code*) propose de développer des interfaces afin de supporter HLA [42]. Le projet a commencé fin mai et se déroule jusqu'au 24 août 2012.

4.1.3 OPNET Modeler

OPNET Modeler est un logiciel de simulation commercial développé par OPNET, une société spécialisée dans les solutions de gestion des réseaux. OPNET Modeler est disponible pour les plates-formes Windows et Linux [43].

Les caractéristiques de base sont identiques à celles d'OMNeT++, OPNET est un simulateur à évènements discrets, offrant une orientation objet et disposant d'interfaces graphiques et d'outils pour la modélisation, l'exécution de la simulation et la visualisation des résultats. Les modèles sont décrits de façon hiérarchique, à l'instar de la structure des réseaux réels, comme l'illustre la figure 4.5.

Des suites optionnelles (*Modeler Wireless Suite* et *Modeler Wireless Suite for Defense*) apportent le support des réseaux sans fil, mobiles, ad hoc (*MANET)*, des modèles de propagation, d'interférence, etc. Il existe aussi un module optionnel amenant le support d'HLA.

Les clients d'OPNET comprennent des fournisseurs de services, des entreprises et des fabricants d'équipements réseaux (Oracle, Total, IBM, Cisco, Conexant, 3Com, BT, France Telecom, NTT DoCoMo, etc.).

OPNET propose une licence académique à but de recherche, mais celle-ci a une validité de six mois, et le renouvellement est soumis au respect de conditions très strictes :

- fournie à titre individuel, le travail partagé n'est pas autorisé ;
- obligation de créer une page web décrivant le projet ;
- obligation de fournir les rapports de progrès, aussi bien en version définitive que les brouillons ;

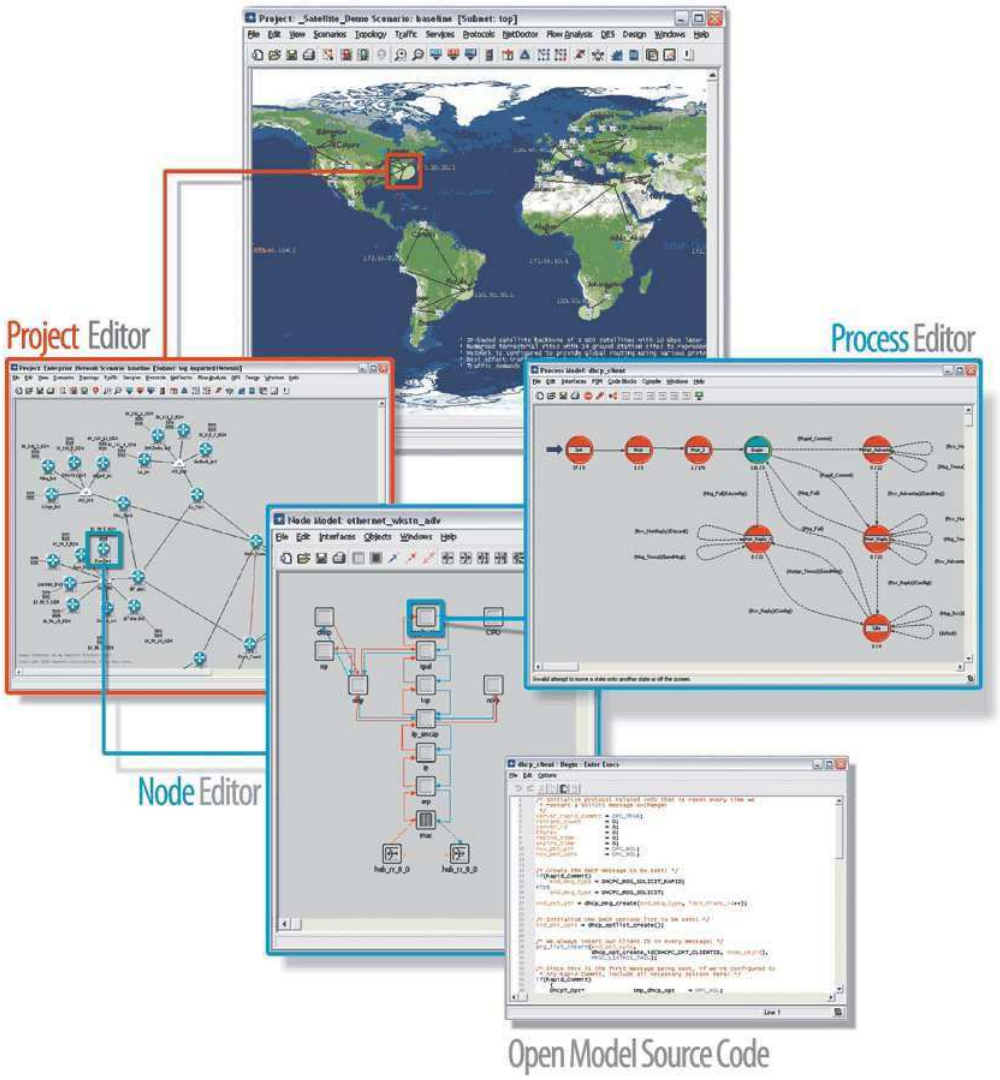


FIGURE 4.5 – Environnement de modélisation hiérarchique (source OPNET)

– obligation de fournir les modèles créés.

4.2 Simulateurs de drones

Il n'existe pas de simulateur spécifique pour les drones. Une plate-forme de simulation est en cours de développement depuis 2010 par une filiale de Capgemini en collaboration avec le cluster aquitain AETOS [44] mais il n'y pas encore de version disponible. Il est néanmoins possible d'utiliser une solution de simulation robotiques, à condition que celle-ci soit capable de gérer trois axes (il faut pouvoir tenir compte de l'altitude), ou même un simulateur de vol et d'utiliser un modèle correspondant à l'AR.Drone.

4.2.1 Player/stage

The Player Project (anciennement appelé Player/Stage Project) [45] vise à créer des logiciels gratuits pour la recherche dans le domaine de la robotique et des systèmes de capteurs. Le projet a démarré en 2000 à l'université de Los Angeles, Californie du Sud et actuellement deux logiciels sont développés.

Player Ce logiciel est une interface de contrôle de robot et peut être décrit comme étant une couche d'abstraction robotique. Le logiciel repose sur un fonctionnement client/serveur communiquant par *sockets* TCP. La partie serveur, installée sur le robot, fournit un ensemble d'interfaces prédéfinies et elle supporte une large variété de robots. La partie client est un ensemble de bibliothèques compatibles avec plusieurs langages de programmation, comme C, C++, Python et Rubis. Player est capable de gérer plusieurs connexions et d'effectuer du travail collaboratif.

Stage Stage est un simulateur de plate-forme robotique. Il permet de simuler dans un environnement basique en deux dimensions de multiples robots, l'échelle pouvant aller d'un à une centaine en même temps. Stage et Player peuvent collaborer, et il est ainsi possible pour les utilisateurs d'accéder à des robots ou des capteurs simulés à travers les interfaces de Player.

Gazebo Initialement, Gazebo, un simulateur de robot en trois dimensions, faisait partie du projet. Ce simulateur intègre un moteur physique ODE (*Open Dynamics Engine*) et un rendu graphique OpenGL. En 2011, Gazebo est devenu un projet indépendant financé en partie par Willow Garage, une entreprise développant des matériels et des logiciels pour des solutions de robotique personnelle. La figure 4.6 illustre la différence de rendu entre Stage et Gazebo.

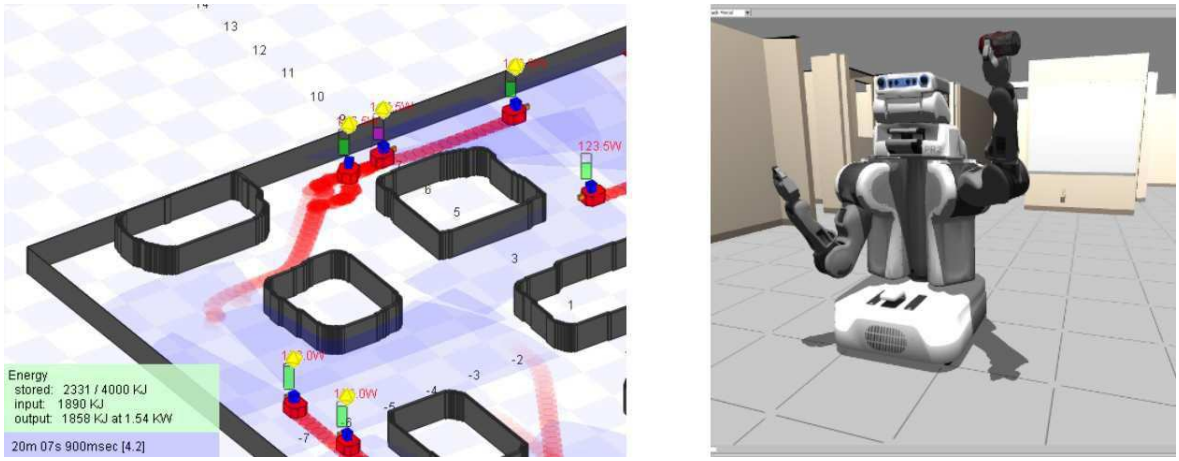


FIGURE 4.6 – Une simulation avec stage (à gauche) et avec Gazebo (à droite)

Player/Stage est très populaire dans le domaine de la recherche, il n'existe cependant pas de modèle de drone, plus particulièrement d'AR.Drone, déjà existant. Il a été décidé de ne pas utiliser ces logiciels dans le cadre de ce projet.

4.2.2 FlightGear

FlightGear est un simulateur de vol open source [46]. Il supporte les plates-formes Windows, Linux et MAcOs et il est développé par une communauté de volontaires. Le code source est entièrement disponible sous licence GNU-GPL.
Le but de FlightGear est de créer un *framework* de simulation propre à être utilisé dans des environnements académiques, de recherche ou d'entrainement au pilotage. Afin de rendre la simulation la plus fidèle possible, il propose trois modèles dynamiques de vol différents, il est aussi possible d'ajouter de nouveaux modèles ou des interfaces vers des modèles propriétaires. FlightGear comprend une base de données vaste modélisant de façon précise des milliers d'aéroports et il permet une représentation détaillée et réelle du ciel et des phénomènes météorologiques, avec un rendu extrêmement précis en trois dimensions, comme l'illustre la figure 4.7.

FIGURE 4.7 – Le rendu en trois dimensions d'un simulation FlightGear

FlightGear ne possède pas de modèle d'AR.Drone, mais il est possible de créer son propre modèle, celui-ci étant la description exhaustive tant au niveau de l'aspect graphique que du modèle physique et de la manière de le contrôler. La description d'un modèle utilise le langage XML et un langage de script propre à FlightGear, Nasal. La documentation et l'aide à l'utilisateur sous forme de wiki sont disponibles sur le site, et la communauté est très active et apporte de l'aide aussi bien aux utilisateurs qu'aux développeurs.

FlightGear est le simulateur choisi pour le projet de deuxième année des étudiants de l'ENSMN, comme décrit au paragraphe 5.2.4.

4.2.3 SAMOVAR

SAMOVAR est une plate-forme de simulation open source dédiée aux réseaux de capteurs et d'actionneurs sans fil (*WSAN, Wireless Sensors and Actuator Network*).

SAMOVAR est un projet de l'équipe TRIO du LORIA et les auteurs sont Lionel Havet et Adrien Guenard [47].

Les *WSANs* mélangent des capteurs et des actionneurs interconnectés par des réseaux sans fil pour accomplir des tâches distribuées, et composent ainsi des *CPS (Cyber Physical Systems)*.

SAMOVAR est fondé sur Matlab/Simulink et sur la bibliothèque TrueTime, un simulateur de systèmes de contrôle en réseau et intégré développé par l'université de Lund en Suède. Il propose des modèles de systèmes *WSAN* contenant à la fois des systèmes robotiques et des réseaux sans fil. Il permet une visualisation en trois dimensions dans des environnements réalistes, comme l'illustre la figure 4.8 représentant le vol d'un quadcopter dans une représentation virtuelle de la smartroom de SUPELEC.

FIGURE 4.8 – Simulation d'un quadcopter dans une représentation de la smartroom de SUPELEC

Le choix d'utiliser SAMOVAR a été fait en tenant compte des avantages suivants :

- un modèle de quadcopter est fourni avec les exemples, et même s'il ne représente pas un AR.Drone, il ne sera pas nécessaire de développer à partir de zéro ;
- il est conçu à partir de Matlab et il existe des interfaces rendant Matlab compatible avec HLA, comme MatlabHLA-TOOLbox [48], un logiciel gratuit fonctionnant avec le RTI CERTI de l'ONERA ;
- il existent aussi des interfaces FMI pour Matlab/Simulink, comme FMI Toolbox de Modelon ;
- la collaboration entre le LORIA et SUPELEC permet d'entrer en contact avec l'un des auteurs, Adrien Guenard, et ainsi d'avoir des réponses rapides en cas de difficultés.

Troisième partie

Conception, développement et évaluation du démonstrateur

Chapitre 5

Interface de contrôle d'AR.Drone

5.1 Présentation de l'AR.Drone et du logiciel existant

5.1.1 Description AR.Drone

FIGURE 5.1 – L'AR.Drone équipé de sa coque d'extérieur

L'AR.Drone est un quadricoptère développé par Parrot, société spécialisée dans les périphériques sans fil pour téléphone mobile. L'AR.Drone se veut un appareil au croisement de plusieurs domaines : le modélisme, le jeu vidéo et la réalité augmentée, comme l'illustre la figure 5.1. L'AR.Drone est un produit grand public disponible en grandes surfaces et magasins Hifi/TV/Vidéo. Son prix de vente est de 300 €. Alors que le premier AR.Drone est sorti en août 2010 en France, une deuxième version est disponible depuis le mois de mai 2012 et elle apporte quelques améliorations : caméra frontale à haute résolution (HD 720p à 30 i/s) et nouveau capteur magnétomètre ajoutant une fonction compas.

Les travaux effectués dans le cadre de ce projet ont été réalisés sur la première

version du drone. Afin d'utiliser la deuxième version des modifications de l'interface de contrôle seront nécessaires sur le traitement de la vidéo,l'encodage étant différent, et sans doute sur les NavData afin d'intégrer les informations issues du compas. Des tests en ce sens sont prévus en septembre.

Caractéristiques du drone

L'AR.Drone est un carré de 50 centimètres de côté et il pèse 380 grammes avec la coque d'extérieur. Une coque d'intérieur permettant de protéger les hélices en cas de choc est fournie avec le drone. Cette coque, illustrée par la figure 5.2, augmente fortement la trainée et modifie les comportements de vol du drone.

FIGURE 5.2 – La coque d'intérieur de l'AR.DRone

Il est équipé de moteurs électriques et d'une batterie lui assurant une autonomie d'environ douze minutes de vol. Le système électronique embarqué est composé d'un processeur ARM9, de mémoire DDR, d'interfaces Wi-Fi et USB et fonctionne sur un noyau Linux.

Le drone est équipé de capteurs permettant de connaître son état lorsqu'il est en vol : un accéléromètre à 3 axes, deux gyromètres et un altimètre à ultrason. Ces capteurs fournissent les informations d'attitude du drone suivant les angles d'Euler (tangage, roulis, lacet) ainsi que l'altitude. Les angles d'Euler sont illustrés par la figure 5.3.

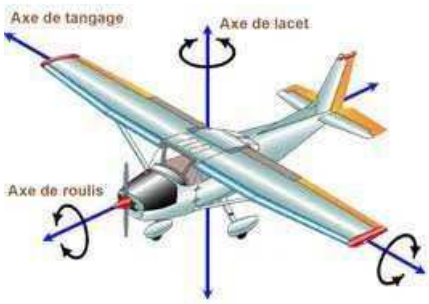


FIGURE 5.3 – Représentation de l'attitude d'un mobile

Le drone possède une caméra frontale et une ventrale permettant la diffusion et l'enregistrement des images sur un périphérique connecté au drone. La caméra verticale sert à la stabilisation du drone, même avec un vent léger. La caméra frontale est utilisée pour les fonctions de réalité augmentée telles que la détection d'ennemis

ou de balises. Cette caméra supporte une résolution QVGA (320x240 à 15 i/s) avec un champ de vision de 90 ° .

Parrot ne fournit pas de dispositif de contrôle pour son drone. Le pilotage s'effectue par un appareil mobile de type smartphone ou tablette par l'application AR.Free Flight. Cette application est disponible pour les systèmes d'exploitation iOS (Apple) ou Android (Google). L'application de contrôle permet de tirer parti de l'accéléromètre de l'appareil mobile et il suffit d'incliner celui-ci pour piloter l'AR.Drone. Un contrôle plus classique par boutons sur l'écran est aussi possible.

La connexion entre le drone et l'appareil mobile est effectuée par Wi-Fi en mode ad hoc. Le drone crée son propre réseau avec choix automatique du canal le moins encombré parmi trois et attribution d'adresse IP (rôle de serveur DHCP) à l'appareil mobile.

Les communications

L'application AR.Free Flight n'est disponible que pour des appareils mobiles mais Parrot fournit un SDK permettant le développement d'application pour contrôler le drone à partir d'un PC (systèmes Linux ou Windows) disposant d'une interface Wi-Fi. Le guide du développeur [49] disponible dans le SDK décrit précisément le protocole de communication. Celui-ci est fondé sur l'utilisation de la pile TCP/UDP/IP : trois ports UDP servent aux services de communications principaux et un port TCP est utilisé lors de transfert d'informations critiques, comme les mises à jour du firmware du drone. Ces différentes communications sont illustrées par la figure 5.4.

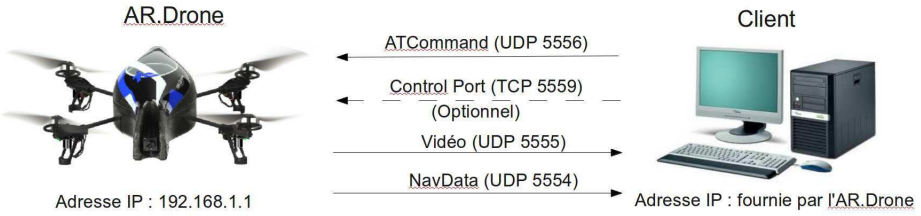


FIGURE 5.4 – Les communications avec l'AR.Drone

Les communications principales sont les données de navigation (*NavData*) et la vidéo reçues du drone, ainsi que les commandes envoyées au drone (*ATCommands*). Ces communications utilisent le protocole non connecté UDP, l'ordonnancement correct des paquets reçus n'est donc pas assuré. La solution employée par Parrot consiste en l'utilisation d'un ordonnanceur réalisé au niveau applicatif : un numéro de séquence est incorporé à chaque paquet transmis.

ATCommands Les commandes de déplacement du drone s'effectuent par le port UDP 5556. Celui-ci sert aussi à envoyer des commandes de configuration comme l'altitude ou les angles maximaux admissibles, l'envoi par le drone de données de navigation en mode réduit (ce mode est décrit dans le paragraphe ci-dessous), etc. Ces différentes commandes sont nommées *ATCommands*. Les commandes utiles pour le démonstrateur seront décrites à la section 5.2.

Les *ATCommands* doivent être envoyées environ 30 fois par seconde afin de permettre un contrôle fluide du drone.

NavData Les données de navigation sont envoyées par le drone via le port UDP 5554. Elles sont appelées *NavData* et sont constituées des informations concernant l'attitude du drone, son statut (posé, en l'air, batterie faible, mode *emergency*, ses vitesses de déplacement, la vitesse de rotation de ses moteurs, etc. Elles contiennent aussi les données utiles pour les fonctions de réalité augmentée, comme la présence d'une cible détectée et sa position.

Un mode appelé *NavDataDemo* consiste à envoyer seulement les informations d'état du drone, pour économiser de la bande passante ou de l'énergie. Ce mode est activable par l'utilisateur en envoyant la commande idoine.

Comme les *ATCommands*, ces données sont transmises environ 30 fois par seconde.

Un exemple d'application pour Linux contenu dans le SDK affiche une représentation graphique des informations contenues dans le flux *NavData*. La figure 5.5 illustre cet exemple.

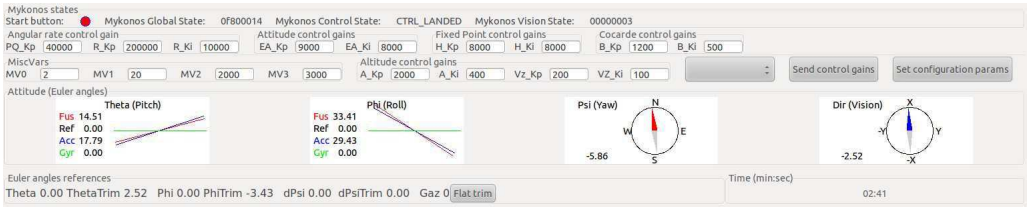


FIGURE 5.5 – Visualisation graphique des *NavData*

Les *NavData* sont transmises au format binaire et forment une structure composée de plusieurs blocs appelés *options*. Cette structure sera détaillée à la section 5.2.

Vidéo Le flux vidéo est transmis par le drone via le port UDP 5555. Le drone possédant deux caméras il est possible pour l'utilisateur de choisir la vidéo reçue (caméra horizontale ou caméra verticale). Le drone est de plus capable de fournir une vidéo PiP (*Picture in Picture*) composée d'une des deux vidéos au choix contenue dans l'autre, comme le montre la figure 5.6.

FIGURE 5.6 – Diffusion des vidéos en mode PiP

Parrot a fait le choix d'utiliser un codec propriétaire nommé P264. Ce codec est dérivé de la norme H.264 afin d'être adapté à la puissance de calcul de l'AR.Drone. Les principales différences se situent au niveau de la gestion de la compensation de mouvements et du codeur entropique.

Le guide du développeur du SDK décrit de manière détaillée le fonctionnement de ce codec.

Firmware

Parrot maintient à jour son drone en fournissant de nouvelles versions de *firmware*.

Lors de la connexion du drone avec un appareil de type iPhone ou iPad, l'application vérifie la disponibilité d'une mise à jour (si une connexion à l'internet est disponible), demande l'accord de l'utilisateur et installe la mise à jour sur le drone.

Il est aussi possible de télécharger la mise à jour sur un PC et de l'installer en établissant une connexion FTP avec le drone.

La version 1.7.4 du mois de mars 2011 a apporté une modification majeure dans le mode de connexion de la liaison Wi-Fi en supportant le mode *access point* en plus du mode *ad hoc*, qui était le seul mode supporté avant cette mise à jour. Le mode de connexion a une influence sur la fonctionnalité *PhoneAttached* d'ARDroneMines, fonctionnalité présentée au paragraphe « Liaison avec un téléphone » de la section 5.1.2.

Il est à noter que Parrot ne prévoit pas la possibilité d'installer une version de *firmware* antérieure à celle installée. Bien qu'il existe des méthodes pour le faire, celles-ci sont officieuses, s'apparentent à du *hacking* et remettent en cause la garantie de l'AR.Drone. L'application ARDroneMines a par exemple des problèmes de compatibilité avec les versions 1.7.4 et supérieures du *firmware*, et ne disposant que de drones étant dans cet état je n'ai pas pu disposer d'une interface pleinement fonctionnelle au début de mes travaux.

5.1.2 Présentation du logiciel ARDroneMines

Le logiciel ARDroneMines a été initialement développé dans le cadre d'un projet d'étudiants de l'École Nationale Supérieure des Mines de Nancy en 2011. Les auteurs, encadrés par Laurent Ciarletta, sont Paulin Andurand, Jeremie Hoelter, Mathieu Plachot et Baptiste Tissot [50].

ARDroneMines a été développé en Java, il permet de contrôler un AR.Drone depuis une machine équipée d'une interface Wi-Fi et il a été testé sur les systèmes d'exploitation Windows 7, Ubuntu et Mac OS. Les drones utilisés lors du développement de l'application étaient équipés d'un firmware 1.4.7, et les auteurs indiquent dans leur rapport qu'il leur a été impossible de faire fonctionner ARDroneMines avec des versions ultérieures de firmware.

La figure 5.7 illustre la fenêtre utilisateur de l'application, à partir de laquelle il peut contrôler le drone et visualiser la vidéo et les informations de navigation.

Le contrôle du drone s'effectue par des touches du clavier, il n'est cependant indiqué ni sur l'interface, ni dans la documentation quelles touches sont utilisées.

Les *NavData* sont représentées par deux blocs : les *Navdata Demo* en haut représentent l'état du drone (angles, altitude, vitesses et niveau de la batterie) alors que des informations plus détaillées fournies par les moteurs et les différents capteurs sont affichées en bas de la fenêtre.

FIGURE 5.7 – L'interface utilisateur d'ARDroneMines.

L'interface graphique utilise la bibliothèque ControlP5, une GUI (*Graphical User Interface*) pour Processing qui est un langage de programmation orienté multimédia. ControlP5 permet de créer facilement des boutons, des *sliders*, des champs de texte ou autres *controllers*.

Des boutons intégrés à l'interface permettent de régler la vitesse de déplacement du drone, de changer la caméra et donc la vidéo reçue, de désactiver le traitement vidéo et enfin de déclencher des scénarios (*TAG* et *DYNAMIQUE*). Ces scénarios seront décrits au paragraphe *Scénarios*.

ARDroneMines est une application multithreadée. Chaque canal de communication (ATCommands, NavData, vidéo) est géré par un *thread*, et le lancement d'un scénario s'effectue lui aussi dans un *thread* dédié.

Des informations sur le drone connecté sont affichées en haut de la fenêtre : son nom, son *id* et son adresse *IP*. ARDroneMines est en effet capable de gérer plusieurs drones en même temps.

Gestion Multi-drones ARDroneMines a été conçu de manière à pouvoir se connecter à plusieurs drones afin de réaliser des scénarios comme le suivi dynamique. Le logiciel utilise une bibliothèque *Processing* appelée *napplet* et qui permet la gestion de plusieurs fenêtres. Ainsi, la fenêtre affichée au lancement du programme, et qui reste ouverte jusqu'à la fermeture de celui-ci, ne comporte qu'un seul bouton *NEWDRONE* comme l'illustre la figure 5.8. Ce bouton permet d'instancier deux classes associant pour un même drone l'interface utilisateur (*ARDroneUserInterface*) et une entité drone (*ARDroneEntity*) qui va contenir toutes les informations concernant le drone, comme son nom, son *id*, son adresse IP, mais aussi les méthodes permettant d'effectuer les connexions (ATCommands, NavData, vidéo), les lancements de scénarios et la déconnexion.

FIGURE 5.8 – Créer une nouvelle entité drone.

La fenêtre suivante concerne la configuration des informations de connexion. Bien que les champs de texte soient préremplis, comme l'illustre la figure 5.9, il est nécessaire afin que l'adresse IP soit prise en compte de cliquer sur le champ *IP* et de valider par la touche *Entrée*.
La fonctionnalité *PhoneAttached* est décrite au paragraphe suivant.

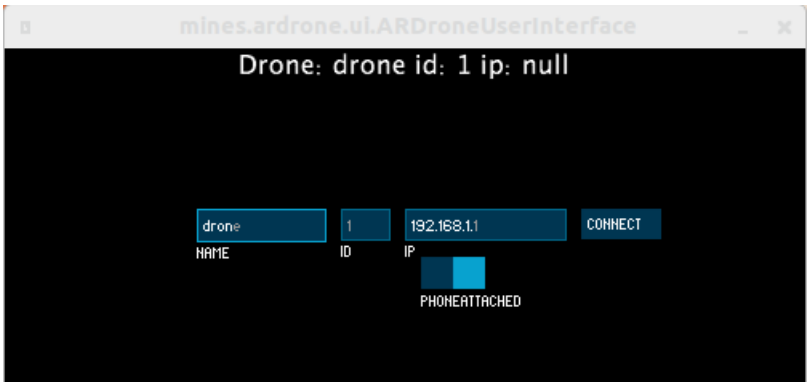


FIGURE 5.9 – La configuration de la connexion.

Liaison avec un téléphone Il a été prévu lors de la conception du logiciel la possibilité d'effectuer une géolocalisation du drone par un *smartphone* Android fixé à celui-ci. Une application a été développée permettant grâce au GPS intégré de récupérer les coordonnées du téléphone.

Cette fonctionnalité n'a pas été testée dans le cadre de ce projet car elle comporte plusieurs limitations :

- elle ne fonctionne pas en intérieur car elle dépend du GPS du téléphone ;
- une version d'Android supérieure à la version 2.1 afin de gérer les connexions ad hoc ou alors une version du firmware de l'AR.Drone supérieur ou égal à la version 1.7.4 pour fonctionner en mode *Access point* ;
- la capacité d'emport du drone est très faible, ainsi un essai a été effectué un fixant un téléphone d'un poids moyen de 120 grammes et le drone avait extrêmement de difficultés à s'élever et à se stabiliser.

Scénarios Deux types de scénarios ont été élaborés permettant un contrôle du drone sans intervention de l'opérateur humain.

Suivi de balise Le drone est capable de reconnaitre une balise, appelée *tag*, de différentes formes et couleurs. La coque d'extérieur, avec ses bandes de couleurs, est reconnue comme un *tag* et des autocollants supplémentaires (*tag 2D*) sont fournis et peuvent être collés sur la coque d'intérieur.
Les *tags* utilisés pour ce scénario, et illustrés par la figure 5.10, sont composés d'un rectangle de trois bandes de différentes couleurs, la principale étant la couleur de la bande du milieu. Le déroulement typique du scénario est celui-ci :

- le *tag* est détecté par la caméra frontale ;
- le drone calcule les coordonnées du *tag* en décalage vertical, horizontal et en éloignement ;
- ces coordonnées sont incluses dans les NavData ;
- le logiciel décide du déplacement à effectuer pour que le *tag* identifié soit toujours au centre de sa vision frontale, à distance fixe, un cercle rouge à l'emplacement du *tag* est inséré dans la vidéo affichée ;
- les commandes de déplacements sont envoyées au drone sous forme d'ATCommands.

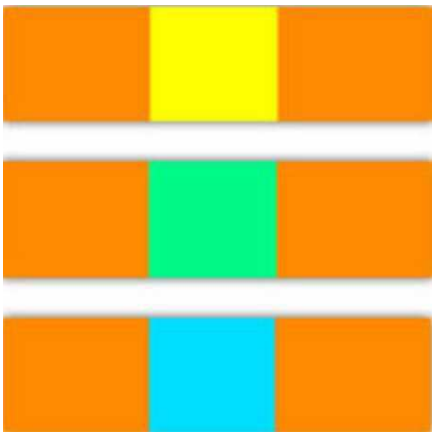


FIGURE 5.10 – Exemples de *tag 2D*

Suivi dynamique Le suivi dynamique consiste à définir un drone *leader* et à transmettre ses déplacements à un ou plusieurs suiveurs.
Le *leader* est désigné par l'*id* paramétré à « 1 » dans la fenêtre de connexion.

Le principe dans ce scénario n'est pas de dupliquer les commandes envoyées au *leader*, mais de récupérer son état (angles d'Euler et altitude) contenu dans les NavData, de transformer celui-ci en ATCommands et de les envoyer au(x) drone(s) suiveur(s). Il est ainsi possible, sans le faire décoller, de se servir du *leader* comme d'une télécommande. En le prenant en main et en l'inclinant, ses mouvements seront retransmis au(x) suiveur(s).

La méthode utilisée pour connecter plusieurs drones au même ordinateur par les auteurs d'ARDroneMines fait appel à un routeur Wi-Fi. Il sera développé dans un paragraphe ultérieur une méthode permettant de se passer de cet équipement.

Contrôle du drone Le premier constat, avant de pouvoir faire un essai en vol, a été que les touches du clavier utilisées pour piloter le drone n'étaient pas indiquées dans le rapport, il donc été nécessaire de faire des recherches dans le code source et comprendre comment celui-ci était articulé pour trouver les affectations des touches. Ensuite, et contrairement à la méthode employée par Parrot dans ses logiciels, le contrôle du drone s'est avéré difficile et peu intuitif. En effet, le drone se déplace lors de l'appui sur une touche et maintient son déplacement après le relâchement de celle-ci. Il faut donc appuyer sur la touche correspondante pour passer en mode stationnaire. De plus, la disposition des touches n'est pas vraiment ergonomique et elles ne sont pas regroupées en blocs, comme c'est le cas dans un jeu vidéo.

Vidéo La bibliothèque Processing utilisée par l'interface utilisateur permet l'affichage de la vidéo en temps réel. La gestion du flux vidéo est assurée par des classes fournies par Parrot. Un bouton de type *toogle* dans l'interface utilisateur permet de terminer (ou de réactiver) le *thread* vidéo. Cette fonctionnalité est utile, car le décodage en temps réel est consommateur de ressources processeur, surtout lors de connexions avec plusieurs drones.
La vidéo envoyée par le drone, et donc affichée dans l'interface utilisateur, est permutable grâce à un bouton *NEXTCAMERA* (vue de la caméra frontale, verticale ou *PiP*). Cependant, lors des essais effectués avec drone équipé d'un firmware 1.7.4, cette possibilité n'était pas fonctionnelle.

Informations de navigation Les informations de navigation sont affichées dans l'interface utilisateur et comprennent l'état du drone et des informations sur le *tag* détecté, les paramètres moteurs, etc. Cependant, je ne disposais que de drones équipés de *firmwares* ultérieurs à la version 1.4.7, aucune de ces informations n'était affichée, et il était rapporté des erreurs de connexion pour le *thread* NavData. Les auteurs d'ARDroneMines indiquent dans leur rapport un possible changement de structure concernant les NavData.
Les scénarios vus précédemment utilisant les NavData, ils n'étaient pas non plus fonctionnels.

Limitations Le logiciel ARDroneMines montre certaines limitations dans son utilisation, outre le problème de compatibilité avec le firmware 1.7.4 de l'AR.Drone.

Il existe un mode *emergency* à l'activation duquel les moteurs se coupent et le drone ne répond plus à aucune commande. Le drone active ce mode automatiquement pour prévenir tous dommages lorsque ses angles deviennent trop importants,

le drone indique le passage en mode *emergency* en allumant en rouge ses diodes situées sous les hélices. Avec l'application de contrôle Parrot, l'utilisateur peut aussi activer et désactiver ce mode par une commande dédiée. Il n'est pas possible avec ARDroneMines de sortir de ce mode, ni de l'activer ou d'avoir une information indiquant dans quel mode est le drone. Lors des essais de fonctionnement du logiciel ARDroneMines effectués dans un volume limité de la taille d'un bureau, le drone est souvent passé dans ce mode lors de collisions avec les murs. La seule solution étant alors de le débrancher électriquement pour le faire sortir de ce mode.
Lorsque le niveau de batterie devient faible, le drone passe en mode *low battery*. Ce niveau est calculé avec une marge de sécurité pour que le drone puisse se poser lorsqu'il est en vol. Le drone en lui-même n'indique visuellement pas qu'il est en mode *low battery*, ses diodes étant toujours allumées, mais il ne répond plus aux commandes. Ce mode peut aussi s'activer lorsque le drone est au sol, en attente. ARDroneMines ne permettant pas de visualiser que le drone est dans ce mode, j'ai fréquemment perdu du temps lors de mes recherches et mes modifications dans le code, car lorsque je voulais tester une nouvelle amélioration et que le drone ne répondait plus aux commandes, je pensais que le problème venait de ces changements, et non pas que le drone était en *low battery*. Il a aussi été constaté une dérive du drone lorsque celui-ci était mis en vol stationnaire. Au lieu de rester à la verticale du point d'arrêt, il glissait lentement et il n'y avait aucune compensation de mouvements. Lorsqu'il est en vol stationnaire, il arrive aussi de manière erratique que le drone ne réponde plus à aucune commande. Il faut dans ce cas relancer ARDRoneMines.
Il existe un *bug* graphique intermittent à la génération de la fenêtre d'interface utilisateur. Celle-ci est alors figée, le programme doit être relancé.
Lorsque les scénarios furent fonctionnels, il s'est avéré qu'il était impossible de recommencer un scénario une fois celui-ci arrêté. La remise en position *on* du *toggle* provoquait des erreurs et l'application devait être relancée.
La couleur principale du *tag* à détecter peut être choisie parmi trois : verte, jaune ou bleue. La reconnaissance est effectuée par la caméra frontale et est très dépendante des conditions de luminosité, une couleur étant plus facilement reconnue qu'une autre suivant les conditions d'éclairage. Cependant, la couleur choisie est codée en *dur* dans ARDroneMines et l'interface utilisateur ne permet pas d'en changer.
Les vitesses de déplacement du drone ne sont pas homogènes. Par exemple, la montée, la descente ou la rotation à droite et à gauche s'effectuent de façon très lente, alors que les déplacements avant/arrière ou latéraux se font à vitesse normale. La hauteur maximale de vol est aussi fixée à trois mètres par un codage en dur, et l'utilisateur ne peut modifier ce niveau. Ces limites diminuent l'expérience de vol.
Il arrive aussi que des problèmes de connexion se produisent, par exemple en lançant l'application alors que la liaison Wi-Fi n'était pas complètement établie ou simplement parce que des obstacles gênent la transmission radio. Des erreurs sont alors reportées par l'application, mais il n'est pas possible de réinitialiser la phase logicielle de connexion. La solution consiste alors à relancer l'application.
Le logiciel est livré sans documentation, que ce soit pour les utilisateurs ou pour des développeurs amenés à travailler sur le code. Le rapport fourni explique les principes de base, mais ne détaille pas le fonctionnement des classes et des méthodes.

5.2 Solution JaSMinDrone

5.2.1 Méthodologie de travail

La difficulté lors du travail sur l'interface de contrôle pour les AR.drones a été d'être confronté à un manque de documentation. Le rapport des étudiants des Mines ne traitait que des grandes lignes du logiciel et le code source n'était pas documenté. Le SDK de Parrot souffre lui d'une grande complexité organisationnelle, il est constitué de multiples dossiers, sous dossiers et fichiers sans explications sur la structure de l'ensemble. De même, la documentation est elle aussi très sommaire. En dernier lieu, les langages utilisés sont différents : Java pour ARDroneMines, langage C pour le SDK.

La démarche effectuée a souvent consisté en la capture des informations échangées entre le drone et l'ordinateur et en la comparaison avec les différentes interfaces à ma disposition, ARDroneMines et le SDK, mais aussi avec l'application Android développée par Parrot.

Le logiciel *Wireshark* a été utilisé pour capturer les communications entre la machine et le drone. *Wireshark* est un analyseur de protocoles réseaux permettant de capturer de façon interactive le trafic traversant une interface réseaux. Il est possible d'appliquer des filtres, comme le type de protocole, le numéro de port ou les adresses IP source et destinataire. Le logiciel permet d'analyser les flux des couches transports afin de visualiser en clair les données échangées. Il est aussi capable de capturer le trafic d'un réseau sans fil, si l'interface Wi-Fi permet de passer en mode *monitor*.

Il a ensuite fallu explorer le code source afin de comprendre son fonctionnement, effectuer des diagrammes de séquences pour visualiser l'enchainement des classes instanciées et utiliser des points d'arrêt dans Netbeans, l'*IDE* utilisé, pour identifier les méthodes incorrectes. Le diagramme de séquence, illustré à la figure 5.11, explique le processus de lancement des *threads* pour le contrôle du drone, la gestion de la vidéo et des NavData. La méthode connect() de la classe ARDroneEntity crée trois *threads*, chaque *thread* créant à son tour un gestionnaire de *socket* s'occupant des communications par le port UDP adéquat.

Les deux paragraphes suivants mettront en œuvre cette méthodologie et permettront d'expliquer de manière détaillée le fonctionnement des *threads* NavData et Control. Les figures 5.12 et 5.13 illustrent les modifications et ajouts apportés à ARDroneMines par JaSMinDrone.

5.2.2 Corrections des limitations d'ARDroneMines

Problème de réception des NavData Les figures 5.14, 5.15 et 5.16 illustrent une capture *Wireshark* lors d'une connexion avec l'Ar.Drone. Un filtre est appliqué afin de ne visualiser que le trafic sur le port UDP 5554 correspondant aux NavData. L'adresse IP 192.168.1.1 correspond au drone, l'adresse 192.168.1.2 au *smartphone* Android et l'adresse 192.168.1.3 à l'ordinateur.
Pour amorcer la transmission, les trois logiciels envoient un paquet à destination du drone. Comme expliqué dans le guide du SDK, le drone transmet ensuite les NavData, à une cadence d'environ trente paquets par seconde. Il apparait sur ces captures que le drone répond directement au *smartphone* avec l'application An-

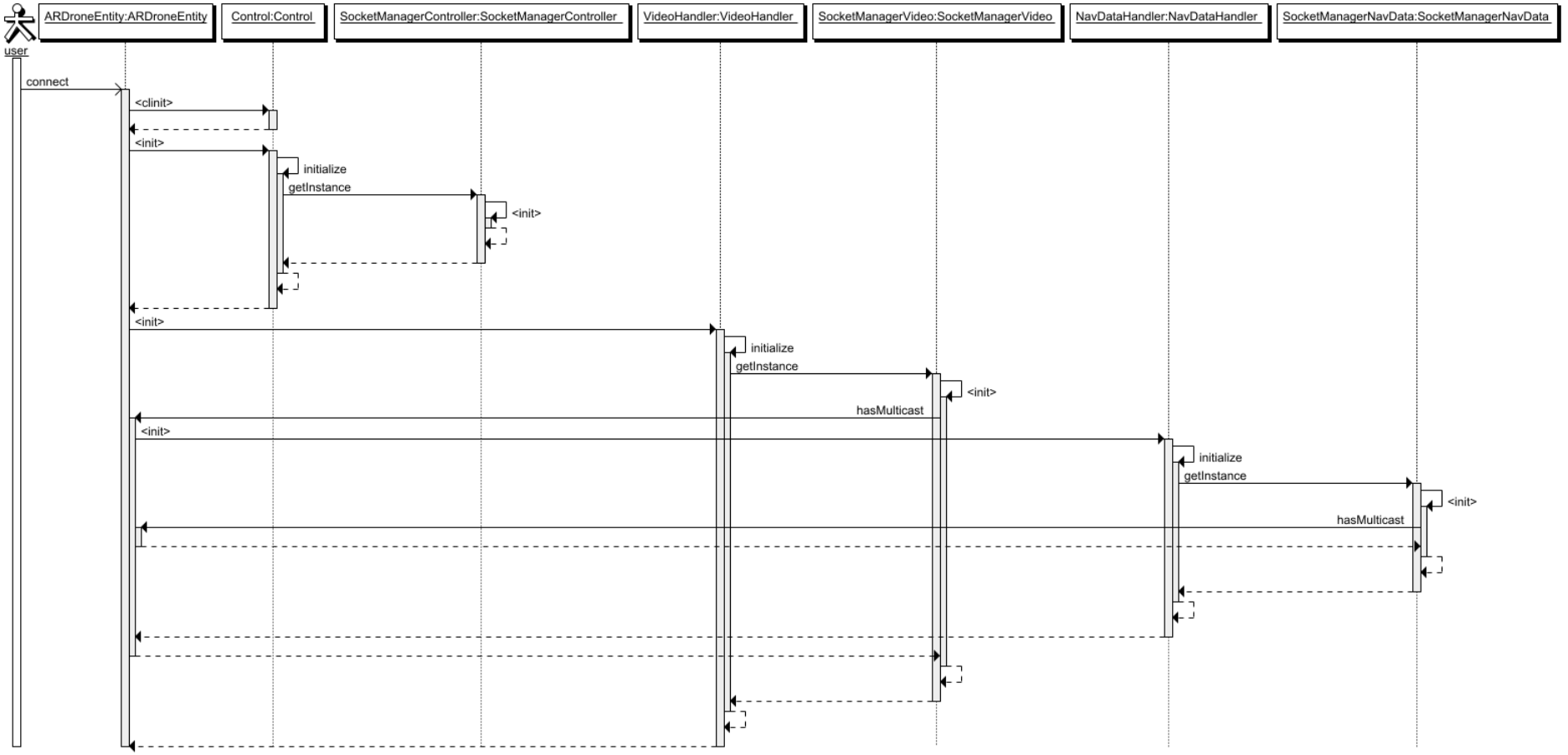


FIGURE 5.11 – Le diagramme des séquences des *threads* principaux : Control, Video et NavData

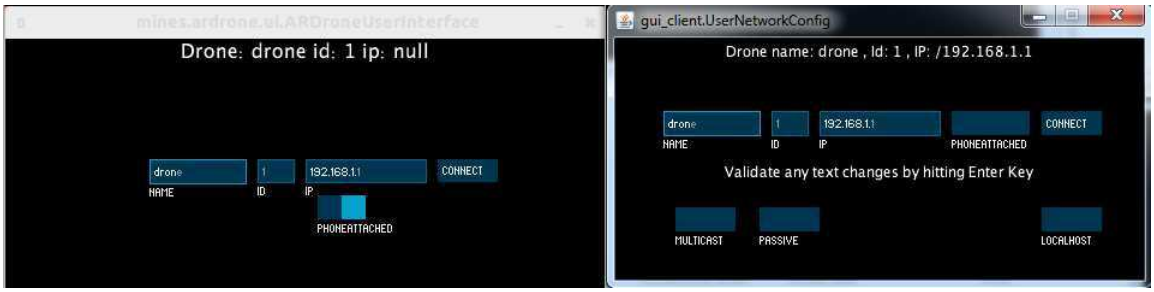


FIGURE 5.12 – Les interfaces de configuration du réseau d'ARDroneMines (à gauche) et de JaSMinDrone (à droite)

FIGURE 5.13 – Les interfaces utilisateurs d'ARDroneMines (à gauche) et de JaSMinDrone (à droite)

droid (adresse destinataire 192.168.1.2) mais qu'il transmet à un groupe *multicast* d'adresse 224.1.1.1 avec les deux autres logiciels.

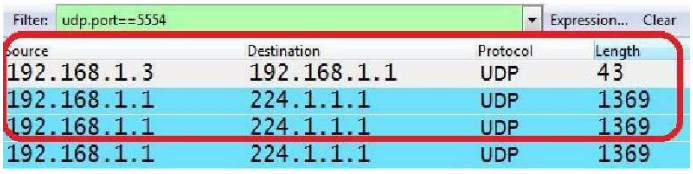


FIGURE 5.14 – Capture de paquet avec l'interface ARDroneMines

En s'intéressant au contenu du paquet envoyé au drone pour initier la connexion, on s'aperçoit que celui-ci diffère suivant les logiciels, comme l'illustre la figure 5.17 :

FIGURE 5.15 – Capture de paquet avec l'exemple du SDK Parrot

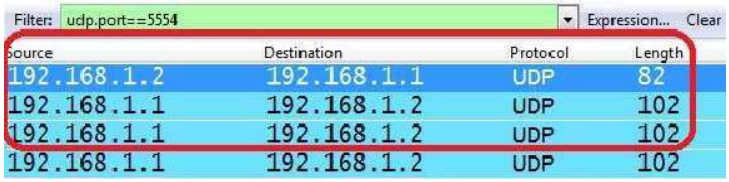


FIGURE 5.16 – Capture de paquet avec l'application Android

- l'application Android envoie 4 octets contenant la valeur 01 00 00 00 ;
- ARDroneMines envoie un seul octet de valeur 0A ;
- le SDK envoie 4 octets de valeur 02 00 00 00.

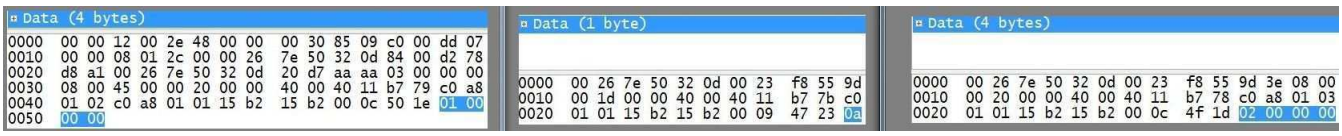


FIGURE 5.17 – Comparaison du paquet envoyé au drone par Android (à gauche), ARDroneMines (au milieu) et le SDK (à droite)

Il semble que le problème de compatibilité ne provienne pas d'un changement de structure des NavData, mais du contenu du paquet initial envoyé au drone.

Pour trouver la classe et la méthode initiatrices de l'envoi du paquet, il a été nécessaire de réaliser un diagramme de séquence et d'étudier le fonctionnement complet de l'ensemble traitant les NavData.

La méthode d'analyse des NavData, reçues sous forme de *BytesBuffer*, sera traitée au paragraphe 6.1.4. Je vais décrire ici le fonctionnement général de l'ensemble, de la réception à la visualisation, des NavData, afin de pouvoir appréhender au mieux le problème de compatibilité avec ARDroneMines.

Dans la phase d'initialisation, le *thread* NavDataHandler va constituer la structure nécessaire à la réception, par la création d'un gestionnaire de *socket* (SocketManagerNavData), à l'analyse (NavDataParser) et à la mise à jour des informations (par des *listeners)*. Le paquet NavData reçu se compose de plusieurs blocs d'informations, chaque bloc forme une structure d'attributs élémentaires. Chaque structure est associée à une classe : NavDataDemo, NavDataPWM, NavDataGyrosOffsets, etc.).

La mise à jour et la mise à disposition des NavData décodées fonctionnent sur le principe du lecteur/rédacteur. Lorsque le *parser* décode un bloc, il fait appel au *listener* correspondant à ce bloc. Le *listener* mettra à jour les attributs dans la structure.

Le diagramme de séquence, représenté par la figure 5.18, présente un exemple de ce principe pour un bloc d'informations, l'attitude du drone. La classe AtittudeListener met à jour les informations de la structure NavDataDemo, par des méthodes set(). Ces informations sont les différents angles d'Euler ainsi que l'altitude. Les intéressés par ces informations, typiquement l'interface utilisateur ou le scénario de suivi dynamique, peuvent ensuite les lire par des méthodes get().

Le diagramme de séquence montre aussi la méthode recherchée concernant le paquet envoyé au drone et provoquant une réponse *multicast*. La méthode tickleNavdataPort de la classe NavDataHandler crée un paquet d'un octet et appelle le gestionnaire de *socket* pour l'envoyer au drone.

```
private void tickleNavdataPort() {

            byte[] buf = new byte[1];
            buf[0] = "\n";
            DatagramPacket packet=new DatagramPacket(buf, 1, drone.
                getInetAddress(), Global.NAVDATA_PORT);
            socketManagerNavData.navdataSocketSendPacket(packet);
    }
```

Le problème de compatibilité avec ARDroneMines a été résolu en modifiant cette méthode. En créant un buffer de 4 octets contenant les valeurs 01 00 00 00, le drone répond à l'adresse IP de l'ordinateur.

```
private void tickleNavdataPort() {

            byte[] buf = {0x01, 0x00, 0x00, 0x00};
            DatagramPacket packet = new DatagramPacket(buf, 4, drone.
                getInetAddress(), Global.NAVDATA_PORT);
    socketManagerNavData.navdataSocketSendPacket(packet);
    }
```

Il est à noter que la version d'ARdroneMines originelle fonctionnait correctement avec les anciens *firmwares* d'AR.Drone, Parrot a du introduire une contrainte à ce niveau dans la dernière version du *firmware*. Il a aussi été essayé d'envoyer un paquet contenant les valeurs 02 00 00 00, comme le fait le SDK. Le drone répond alors en multicast. Cela m'a poussé à développer une fonctionnalité multicast, qui sera détaillée dans un paragraphe ultérieur.

Modification de la gestion des touches Pour améliorer l'ergonomie de contrôle du drone, il fallait modifier l'affectation des touches du clavier et la gestion de ces touches. Le but était que le drone se déplace à l'appui d'une touche et s'arrête au relâchement de celle-ci, comme il le fait avec les applications fournies par Parrot. La méthode employée a été la même que pour le problème des NavData.

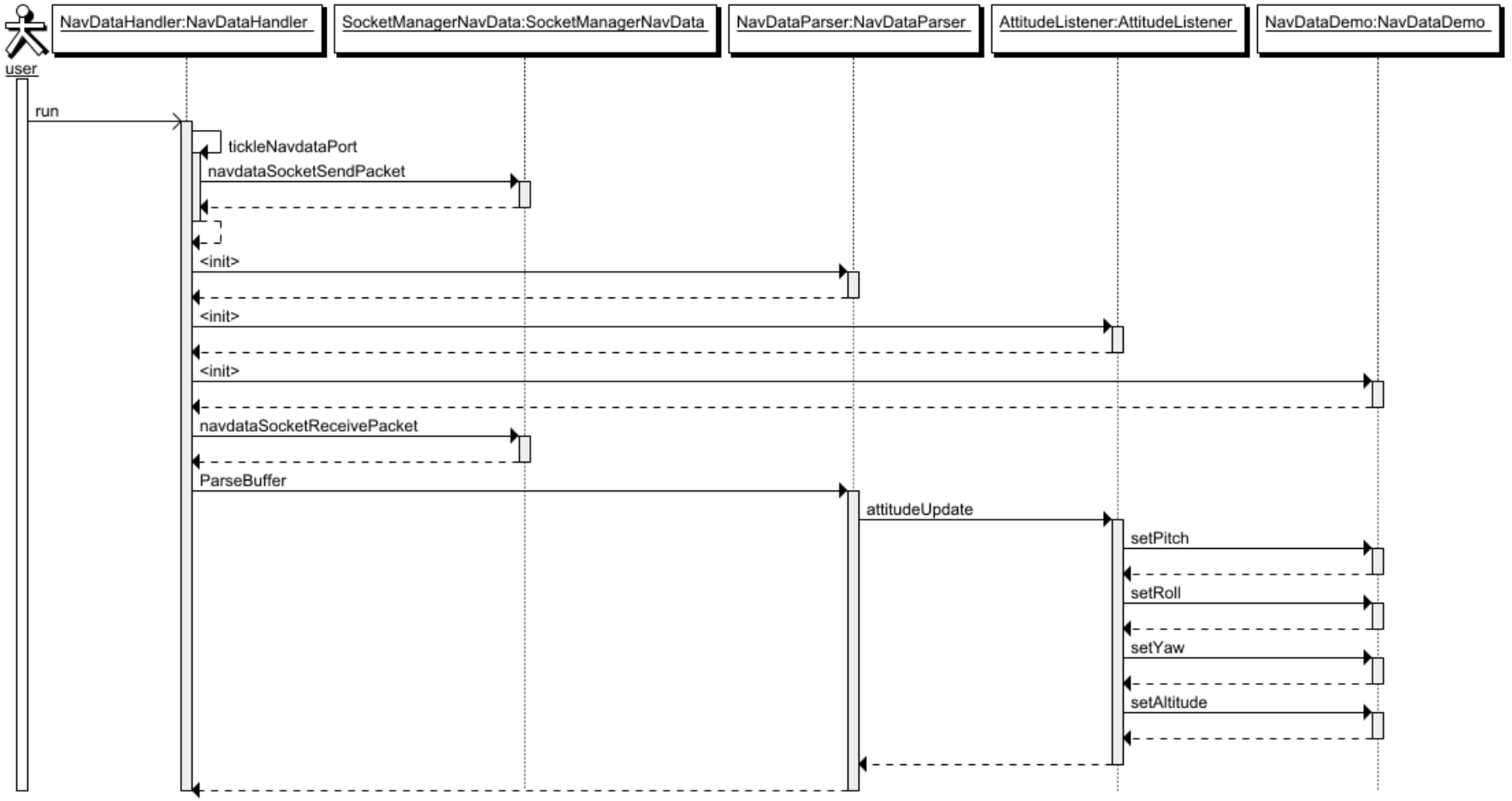


FIGURE 5.18 – Le diagramme des séquences pour le *thread* navdata

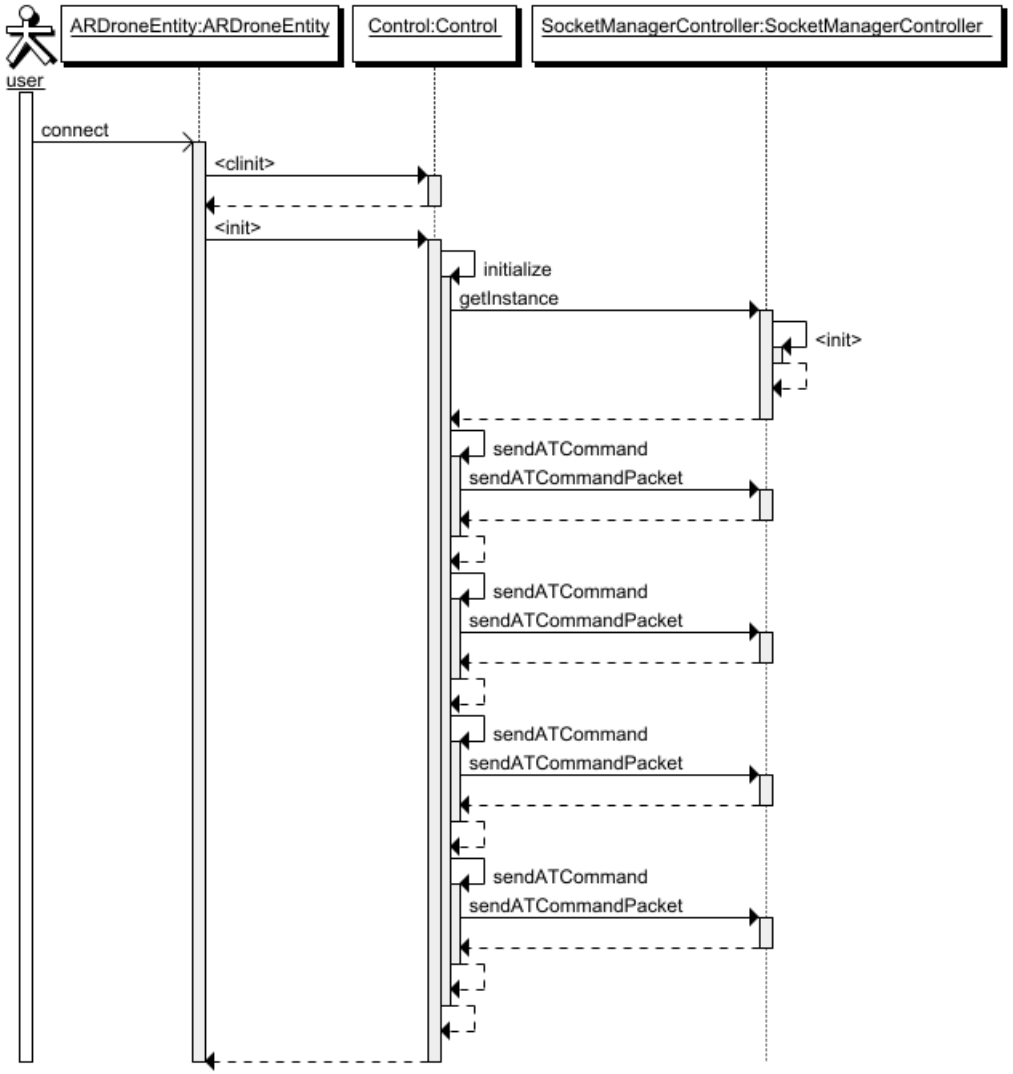


FIGURE 5.19 – Le diagramme des séquences pour le *thread* navdata

La figure 5.19 représente le diagramme de séquence pour le *thread* Control. Celui-ci, après avoir instancié un gestionnaire de *socket*, lui transmet un paquet ATCommand qui sera envoyé au drone. En analysant le code de la classe Control, il apparait deux phases distinctes :

- une phase d'initialisation, les commandes envoyées concernent la configuration du drone, comme la désactivation du mode NavDataDemo, le type de cible à détecter, sa couleur et l'activation de la vidéo ;
- une phase s'exécutant en boucle, envoyant une commande toutes les vingt millisecondes conformément au guide du SDK, et ce jusqu'à la déconnexion.

Le listing suivant est un extrait de la méthode run() qui s'exécute en boucle.

```
public void run(){

    if(this.atCommand!=null){
        sendATCommand(this.atCommand);
        sleep(20);
        if(!this.continuance){
            this.atCommand=null;
            continue;
        }
        continue;
    }

    if(drone.landing){
        sendATCommand("AT*PCMD="+(seq++)+",1,0,0,0,0"
                      +CR+"AT*REF="+(seq++)+",290717696");
        sleep(20);

    }else{
         sendATCommand("AT*PCMD="+(seq++)+",1,0,0,0,0"
                       +CR+"AT*REF="+(seq++)+",290718208");
         sleep(20);
    }
}
```

Des tests conditionnels sont effectués et trois commandes distinctes sont envoyées :

- une commande de déplacement si celle-ci existe ;
- une commande indiquant au drone de se poser ou de rester au sol ;
- une commande indiquant au drone de décoller ou de rester en vol stationnaire s'il est déjà en vol.

C'est par l'état du booléen *continuance* que le drone maintient son déplacement même après avoir relâché une touche, l'appel à la méthode de vol stationnaire (*hover*) le met dans l'état faux, et le déplacement s'arrête.

La figure 5.20 représente le flux UDP envoyé au drone après analyse par Wireshark. On distingue la phase d'initialisation, jusqu'à l'activation de la vidéo (commande « general :video_enable","TRUE" ») puis les commandes envoyées en boucle, alternativement les commandes AT*REF et AT*PCMD.

FIGURE 5.20 – Le flux UDP des ATCommands analysé par Wireshark

La signification de ces commandes est détaillée dans le guide du SDK. Le premier argument après le signe égal correspond au numéro de séquence, incrémenté à chaque commande, et qui fait office d'ordonnanceur logiciel. AT*REF prend deux valeurs distinctes et indique le statut du drone, actuel ou voulu, entre posé ou en l'air. Cette commande permet ainsi de faire décoller ou atterrir le drone. La commande AT*PCMD est la commande de déplacement. Outre le numéro de séquence, elle comprend cinq arguments. Le premier est toujours positionné à un, les quatre suivants correspondent respectivement aux mouvements latéraux gauche-droite, avant-arrière, montée-descente et pivoter à gauche-droite. La valeur d'un déplacement est exprimée par un pourcentage de la vitesse maximale que le drone peut prendre. Cette valeur est codée sous la forme d'un flottant signé selon le format IEEE754. Ainsi, pour indiquer un déplacement vers l'avant à 80% de la vitesse maximale, soit -0,8 dans le repère du drone, l'ATCommand correspondante sera : AT*PCMD=xx, 1 , 0, -1085485875, 0, 0.

Dans ARDRoneMines, lorsque l'utilisateur appuie sur une touche pour passer une commande, c'est l'interface utilisateur, la classe ARDroneUserInterface, qui traite l'évènement par la méthode keyPressed(). Celle-ci contient une série de *if else* imbriqués qui, en fonction de la touche appuyée, appelle la méthode correspondante de la classe Control. Les modifications apportées pour améliorer l'ergonomie ont changé l'organisation de cette méthode. Les changements ont consisté en :

- suppression des *if else* imbriqués et remplacement par un *switch case* ;
- ajout d'un argument d'entrée *KeyEvent* et affectation du *KeyCode* à un entier pour effectuer la fonction *switch case* ;
- affectation des touches Z,S,Q et D pour les déplacements, à la manière d'un jeu vidéo ;

- ajout de commandes, comme *Emergency*, *Trim*, *Re-initialize* ou *DroneState* ;
- ajout d'une méthode keyReleased() qui appelle la méthode de vol stationnaire au relâchement d'une touche. Cet ajout s'est accompagné de modifications dans la méthode run() de la classe Control comme la suppression du booléen Continuance.

L'ajout d'une commande est accompagné par la création de la méthode correspondante dans la classe Control. Ainsi la commande *Emergency* permet d'entrer ou de sortir du mode d'urgence. Elle appelle la méthode emergency() suivante :

```
public void emergency() {
        ATconstruct = false;
        setATCommand("AT*REF=" + (seq++) + ",290717952", false);
    }
```

La commande *Trim* permet d'effectuer une remise à zéro des centrales inertielles lorsque le drone est posé sur un sol plat, de niveau.

La commande *Re-initialize* permet de relancer la connexion au drone après la survenue d'une erreur réseau.

D'autres commandes dédiées aux développeurs ont été ajoutées comme l'obtention du masque d'état du drone ou la remise à zéro du numéro de séquence.

Correction du mode stationnaire En vol stationnaire, le drone présente une légère dérive, et ne reste finalement pas sur place. La commande envoyée par ARDroneMines dans la méthode hover() indique au drone de n'effectuer aucun mouvement. En étudiant la description de la commande ATPCMD dans le guide du SDK, il apparait que le premier argument, toujours positionné à « 1 » par ARDRoneMines, correspond au maintien sur place quand il a pour valeur zéro. Dans ce mode, le drone effectue automatiquement de légères corrections afin de supprimer la dérive.

Ainsi le problème de la dérive du drone a été supprimé en modifiant le code de la méthode hover() de AT*PCMD=xx,**1**,0,0,0,0 à AT*PCMD=xx,**0**,0,0,0,0.

Suppression de la perte de contrôle Le drone est équipé d'une fonction de surveillance des communications appelée *watchdog*. Lorsque le drone ne reçoit aucun trafic pendant plus de cinquante millisecondes, il entre dans un mode dans lequel il ne tient plus compte des commandes reçues. Pour sortir de ce mode, la commande AT*COMWDG doit lui être envoyée. La solution pour éviter les pertes de contrôle intermittentes qui arrivaient avec ARDroneMines a été d'ajouter la commande AT*COMWDG aux commandes envoyées en boucle dans la méthode run() de la classe Control.

Reprise sur erreur réseau Les exceptions réseau ne sont pas gérées dans ARDroneMines, et lors d'un problème de connexion le logiciel doit être relancé. Pour remédier à cela, il a été ajouté des booléens navDataError et videoError dans la classe ARDroneEntity. Ces booléens sont positionnés par les blocs *try catch* dans les méthodes de réception de paquet des classes NavDataHandler et VideoHandler. L'interface utilisateur teste ces booléens et lorsqu'une exception apparait, un

message est affiché prévenant l'utilisateur d'un problème réseau, comme l'illustre la figure 8.4 à l'annexe 8.

Des méthodes ont été ajoutées aux classes ARDroneEntity, NavDataHandler et VideoHandler, et permettant de réinitialiser les connexions lorsque l'utilisateur appuie sur la touche correspondante.

Correction des lancements de scénarios Les scénarios de suivi de *tag* et de suivi dynamique sont effectués par des *threads* dédiés. Ces *threads* sont créés dans la méthode setup() de la classe ARDroneUserInterface. Cette méthode est exécutée une seule fois à l'instanciation de cette classe. Au démarrage d'un scénario, une méthode démarre le *thread* : flightScenario.start(). Lorsque l'utilisateur arrête un scénario, une méthode est appelée et tue le *thread* : flightScenario.kill(). Il y a alors un problème lorsque le scénario doit être de nouveau lancé, le *thread* demande à être lancé, mais il n'existe plus, ayant été tué auparavant. La solution a été de déplacer les fonctions créant les *threads* dans le code appelé lors d'un lancement ou de l'arrêt d'un scénario, comme le montre le listing suivant :

```
private void Tag(boolean theFlag) {
        if (theFlag) {
            TagFollowing = new TagFollowing(drone);
            drone.setTagDetection(TagFollowing);
            drone.startFlightScenario(TagFollowing);
            System.out.println("TagFollowing begins");
        } else {
            drone.stopFlightScenario(TagFollowing);
            System.out.println("TagFollowing ends");
        }
    }
```

Changement de caméra Le changement de caméra dans ARDroneMines s'effectue par un bouton affiché sur la fenêtre utilisateur. Les éléments interactifs de cette fenêtre sont créés à partir de la bibliothèque ControlP5. Un bouton est associé à une méthode contenue dans la même classe que le bouton et qui sera appelée lors de l'appui sur celui-ci. Le listing suivant détaille la création du bouton butNextCamera et la méthode NextCamera() associée :

```
butNextCamera = controlP5.addButton("NextCamera", 0, 10, 285, 60, 19);
controllersMainPage.add(butNextCamera);

public void NextCamera() {
            drone.getController().setNextCamera();
    }
```

Le bouton est créé et placé aux coordonnées précisées, puis il est affiché sur la fenêtre principale. En appuyant sur le bouton, l'utilisateur appelle la méthode setNextCamera() de la classe Control, cette méthode envoie la commande AT*ZAP au drone. En cherchant dans le SDK, il apparait que cette commande est obsolète à partir du *firmware* 1.7.4 et a été remplacée par la commande AT*CONFIG associée

au paramètre « video :video_channel ».
La modification a été apportée à la méthode et le changement de caméra est fonctionnel. Il a été décidé de supprimer le bouton NextCamera de l'interface utilisateur afin de laisser de la place pour un bouton NavDataDemo qui sera décrit ultérieurement. Le changement de caméra s'effectue par la touche « c » du clavier.

Interface graphique Il apparait de façon intermittente un *bug* graphique à la création de la fenêtre utilisateur : le texte affiché est dédoublé et il n'y a pas d'actualisation des informations. Ce problème est illustré par la figure 5.21. Les essais effectués afin de localiser et de corriger ce problème n'ont pas été concluants : séparation des fenêtres de configuration réseaux et utilisateur, mise en place de mécanismes permettant la création et le lancement des *threads* avant l'affichage de la fenêtre utilisateur, réduction du nombre de polices de caractères chargées par les fenêtres, mise en place de point d'arrêt et utilisation du mode débogage de Netbeans. Il semble que ce *bug* apparaisse plus fréquemment avec un système Linux, comme Ubuntu 11.10 utilisé pour ce projet, et il a été moins souvent constaté sur les systèmes Windows.

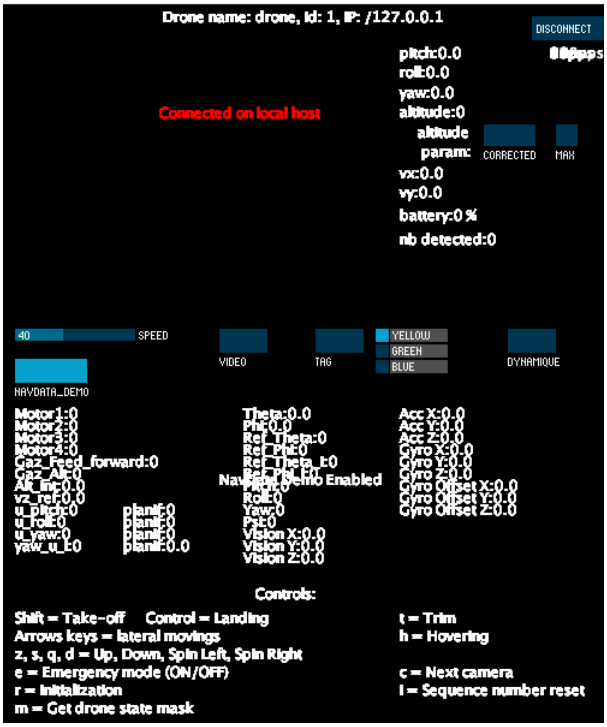


FIGURE 5.21 – Le problème intermittent d'affichage de l'interface graphique

Les bibliothèques ControlP5 et NApplet utilisées pour la création des fenêtres sont en version bêta, et il est possible que le *bug* rencontré provienne de l'une de ces bibliothèques. Une solution serait de programmer l'interface graphique avec Swing, une bibliothèque incluse dans Java.

5.2.3 Fonctionnalités ajoutées

Affichage d'informations Des informations ont été ajoutées à la fenêtre utilisateur, comme l'affectation des touches de commandes en bas de la fenêtre. Des

messages sont aussi affichés en surimpression sur la vidéo pour indiquer que le drone est en mode *emergency* ou *low battery*. Ces ajouts sont illustrés par les différentes figures de l'annexe 8.

Connexion multidrones L'AR.Drone, en tant qu'*access point*, crée son propre réseau Wi-Fi caractérisé par un SSID et une fréquence correspondant à un canal libre. Il est nécessaire d'effectuer une manipulation afin de connecter plusieurs drones au même ordinateur. Le principe consiste à choisir un drone comme *leader* et à récupérer ses informations de connexion, SSID et fréquence, avec la commande *iwconfig*. Puis, à se connecter au second drone et à le configurer depuis un terminal *telnet* pour le passer en mode *managed*, changer le SSID et la fréquence pour ceux du *leader* et à choisir une adresse IP différente de celles du *leader* et de l'ordinateur. Cette configuration n'est pas permanente et doit être réalisée après chaque mise sous tension d'un drone.

Il est à noter qu'en tant qu'*access point*, tout le trafic transite par le drone *leader*, y compris les communications du second drone vers l'ordinateur. Les essais effectués avec deux drones n'ont pas été sujets à ralentissement ou congestion de trafic, mais le drone *leader* peut être surchargé en cas de connexion avec plus de drones.

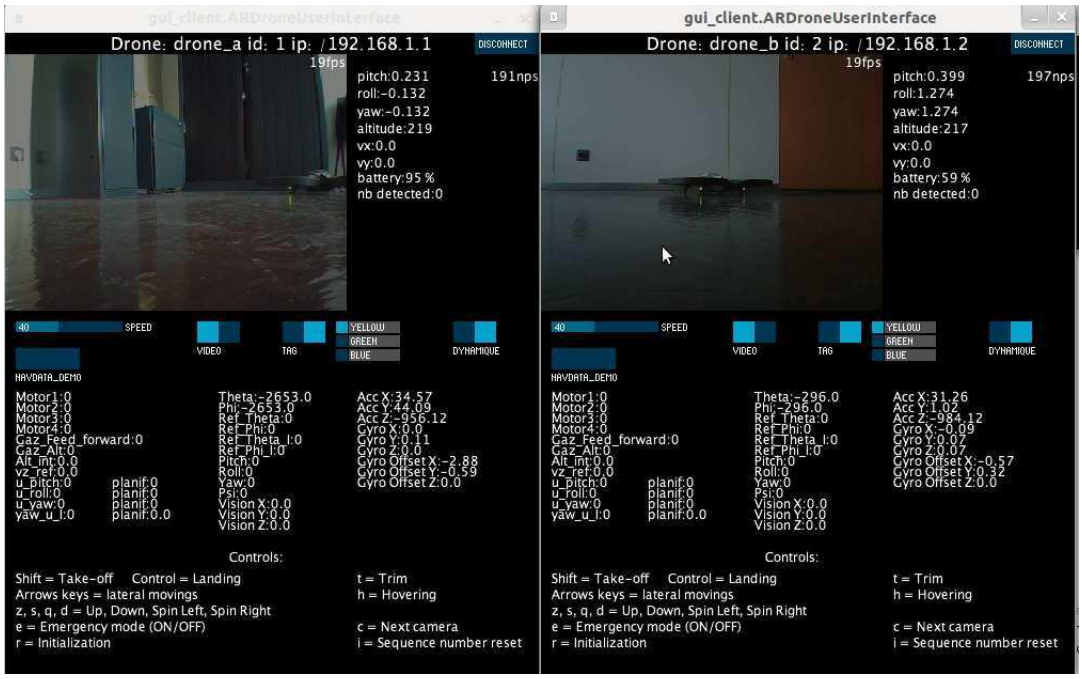


FIGURE 5.22 – Connexion de deux AR.Drones avec JaSMinDrone

Afin de pouvoir différencier les drones, il convient de leur donner un nom et un *id* différents dans JaSMinDrone, comme l'illustre la figure 11.16.

Multicast Il a été vu que le drone pouvait envoyer les NavData à une adresse de groupe multicast. La question de l'intérêt de cette possibilité m'a poussé à étudier le fonctionnement du deuxième flux de communication envoyé par le drone, la vidéo. Il apparait que l'organisation de la gestion de la vidéo par ARDroneMines est simi-

laire à celle des NavData, en plus simplifiée : une classe VideoHandler, un *listener* ImageListener et un gestionnaire de *socket* pour la vidéo. La classe VideoHandler est elle aussi chargée d'amorcer la connexion vers le drone en envoyant un paquet lui signifiant de commencer à émettre le flux vidéo. La méthode tickleVideoPort est identique à la méthode tickleNavdataPort de la classe NavDataHandler décrite antérieurement. Il est à noter que, pour la classe VideoHandler, le contenu du paquet créé était correct et qu'il n'a pas fallu le modifier (01 00 00 00). Le test a été fait de remplacer le contenu par les valeurs 02 00 00 00 et une capture Wireshark a montré que le drone envoyait la vidéo la aussi à l'adresse multicast 224.1.1.1.
Les deux flux d'informations pouvant être émis par le drone à une adresse multicast, une modification d'ARDRoneMines a été envisagée enfin de permettre d'une part une connexion en multicast par un client actif, et d'autre part la réception de ces informations par un ou des clients passifs.
La première étape a consisté à modifier les gestionnaires de *sockets* NavData et vidéo afin d'avoir un fonctionnement en multicast, comme le montre l'extrait de la méthode ci-dessous :

```
socketMulticast = new MulticastSocket(Global.VIDEO_PORT);
InetAddress multiGroup = InetAddress.getByName(Global.MULTICAST_GROUP);
socketMulticast.setInterface(drone.getMultiInterfAdress());
socketMulticast.joinGroup(multiGroup);
socketMulticast.setSoTimeout(3000);
```

Ce code consiste à créer un *socket* multicast lié au port vidéo (défini dans la classe Global), à définir une adresse de groupe multicast (définie elle aussi dans la classe Global) et à rejoindre ce groupe. L'ordinateur sur lequel j'effectuais les essais étant doté de plusieurs interfaces réseaux, la fonction setInterface() permet de spécifier sur quelle interface établir la connexion.
Les essais montrant que la connexion en multicast était fonctionnelle, la deuxième étape a consisté à modifier l'interface de connexion réseau afin permettre à l'utilisateur de choisir un fonctionnement classique ou en multicast, et dans ce cas un champ de texte est ajouté pour indiquer l'adresse de l'interface réseau sur laquelle établir la connexion. Cette fenêtre est illustrée par la figure 5.23. Les gestionnaires de *socket* ont aussi été modifiés afin de pouvoir gérer les deux cas.

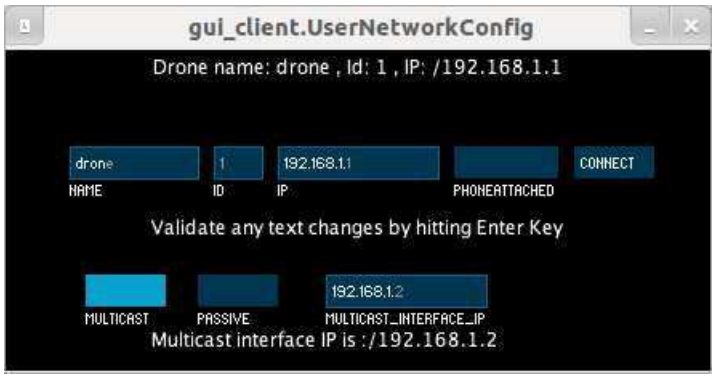


FIGURE 5.23 – La configuration en multicast

La dernière étape a consisté à créer un client multicast passif, c'est-à-dire un mode dans lequel seule la réception des NavData et de la vidéo, émises en multicast par un drone, est possible, le client passif ne pouvant pas envoyer de commandes. Il a fallu pour cela modifier la méthode Connect() afin de ne pas exécuter le *thread* Control dans ce mode, il a fallu aussi modifier l'interface utilisateur pour désactiver la gestion des touches du clavier et ne pas afficher les boutons inutiles. La figure 5.24 illustre la fenêtre utilisateur dans ce mode.

FIGURE 5.24 – La fenêtre utilisateur en mode multicast passif

Pour profiter de ce mode, un client actif doit être connecté et avoir activé le mode multicast. Le client passif, exécuté sur une autre machine, peut alors effectuer du *monitoring* ou bien faire un traitement vidéo de manière déportée.
Une vérification est effectuée lors de la configuration réseau pour qu'il ne soit pas possible d'avoir plusieurs drones en multicast, lors d'une connexion multidrones, comme l'illustre la figure 5.25.

Choix de la couleur de la balise à détecter Le but est de pouvoir choisir dynamiquement la couleur de la balise à détecter pour le scénario suivi de *tag*.

Pour cela, trois boutons de types radio (un seul peut être activé en même temps) ont été ajoutés à l'interface utilisateur. Ces boutons sont associés à une méthode

FIGURE 5.25 – Un seul drone est autorisé à se connecter en multicast

ajoutée à la classe Control. Cette méthode envoie une commande AT*CONFIG avec comme paramètre la couleur choisie, comme l'indique le listing suivant :

```
public void tagColor(int tagColorChosen) {
        switch (tagColorChosen) {
            case 1: {
                System.out.println("Tag Color is Yellow");
                sendATCommand("AT*CONFIG=" + (seq++) + ",\"detect:
                    enemy_colors\",\"1\""); //yellow
                break;
            }
            case 2: {
                System.out.println("Tag Color is Green");
                sendATCommand("AT*CONFIG=" + (seq++) + ",\"detect:
                    enemy_colors\",\"2\""); //Green
                break;
            }
            case 3: {

                System.out.println("Tag Color is Blue");
                sendATCommand("AT*CONFIG=" + (seq++) + ",\"detect:
                    enemy_colors\",\"3\""); //blue
                break;
            }

        }
    }
```

Grâce à cette méthode, il est possible de changer la couleur à détecter, et ce même pendant la réalisation d'un scénario.

Séparation de l'interface graphique L'interface graphique a été séparée du *moteur d'exécution*, celui-ci ayant été regroupé dans un *package* nommé JaSminLib. L'interface graphique constitue le *package* JaSMinDrone et elle utilise la bibliothèque JaSMinLib comme l'illustre la figure 5.26.

Cette séparation a permis la création d'une classe nommée Autopilot et située

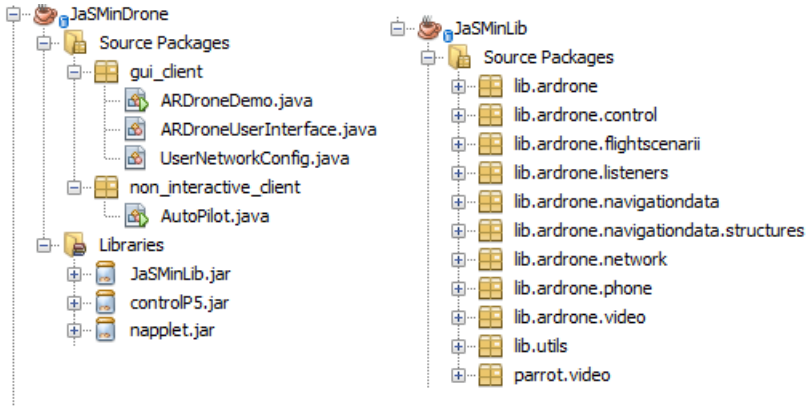


FIGURE 5.26 – La séparation de l'interface graphique et de la bibliothèque

dans le *package* « non_interactive_client ». Cette classe permet de programmer le vol du drone, à la manière d'un script, sans avoir recours à l'interface utilisateur. Cela permet la reproduction à l'identique d'un vol, pour effectuer par exemple la calibration d'un drone simulé avec un drone réel. Cette classe est représentée à l'annexe 9 et les commandes permettent d'indiquer un temps de déplacement et le pourcentage de la vitesse maximale, comme l'illustre le script suivant :

- décollage ;
- avancer pendant deux secondes ;
- rester sur place pendant une seconde ;
- reculer pendant une seconde à soixante pour cent de la vitesse maximale ;
- etc.

Il a fallu créer les méthodes correspondantes à ces déplacements dans la classe ARDroneEntity, qui est le seul objet connu d'Autopilot, comme le code des méthodes suivantes :

```
public void goForward(int time) {
        System.out.println("\nGo forward during " + ((float) time / 1000) + "
            s");
        getController().forward();
        try {
            Thread.sleep(time);
        } catch (Exception e) {
        }
        getController().hover();
    }

public void goForward(int time, int speed) {
        setSpeed(speed);
        goForward(time);
        System.out.println("at " + (speed) + "% of speed\n");
        setSpeed(40);
    }
```

Avant d'exécuter le script, la classe teste l'état du drone pour connaitre son état de batterie et savoir si le mode *emergency* est enclenché et prévient l'utilisateur de

l'un ou l'autre, le choix de sortir du mode *emergency* lui étant aussi proposé. Ces remontées d'informations sont nécessaires, car, sans interface utilisateur, il n'est pas possible de savoir pourquoi le drone n'accomplit pas le script prévu.

Encapsulation des méthodes La majorité des méthodes d'ARDroneMines étaient déclarées en tant que *public*, même quand cela n'était pas nécessaire. L'encapsulation a été revue afin de déclarer en tant que *private* les méthodes qui n'avaient pas à être accessibles d'autres classes.

Connexion à un simulateur Dans le but de coupler JaSMinDroneavec un simulateur s'exécutant sur la même machine, un bouton a été ajouté à l'interface de configuration des informations réseaux. Celui-ci configure automatiquement l'adresse IP en adresse *local host* (127.0.0.1) et change le port UDP source utilisé par JaSMinDrone pour les ATCommands. Conformément au guide du SDK, les ports UDP utilisés par ARDRoneMines sont identiques pour la source et pour la destination, comme l'illustrent les captures effectuées par Wireshark 5.14. Cela pose un problème au niveau des ATCommands pour la connexion en local à un simulateur. Le simulateur essaie dans ce cas d'ouvrir le port UDP 5556 pour recevoir les ATCommands, mais celui-ci étant déjà ouvert par ARDroneMines, il en résulte une erreur. En configuration *localhost*, le port UDP utilisé pour les ATCommands est le port 7556. La figure 5.27 illustre le changement effectué à l'interface de configuration réseau. La configuration en multicast n'est pas possible en mode *localhost* et les champs de configuration ne sont alors pas affichés.

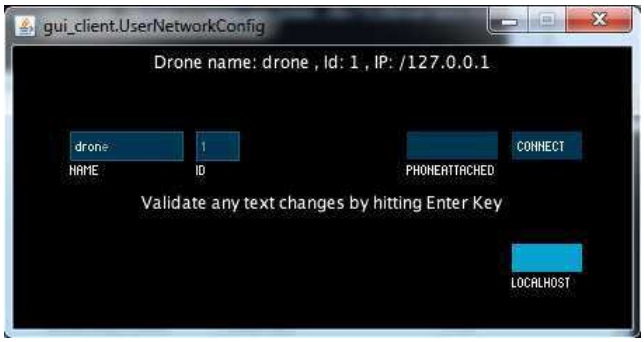


FIGURE 5.27 – La configuration en mode *localhost*

La vidéo n'étant pas prévue d'être traitée lors d'une connexion avec un simulateur, le *thread* Video n'est pas exécuté lors de la méthode Connect() afin d'économiser des ressources. Un message est affiché à la place de la vidéo prévenant l'utilisateur de la connexion en *local host*.

NavDataDemo En activant le mode NavDataDemo, les NavData envoyées par le drone ne contiennent plus que les informations d'état du drone. Cela permet d'économiser de la bande passante, le paquet envoyé ne faisant que 66 octets contre 1369 octets en mode complet. Un bouton ajouté à l'interface utilisateur permet d'activer ou de désactiver ce mode, en envoyant la commande AT*CONFIG associée au paramètre « general :navdata_demo ». Un message d'information est alors affiché à la place des NavData, comme l'illustre la figure 5.28.

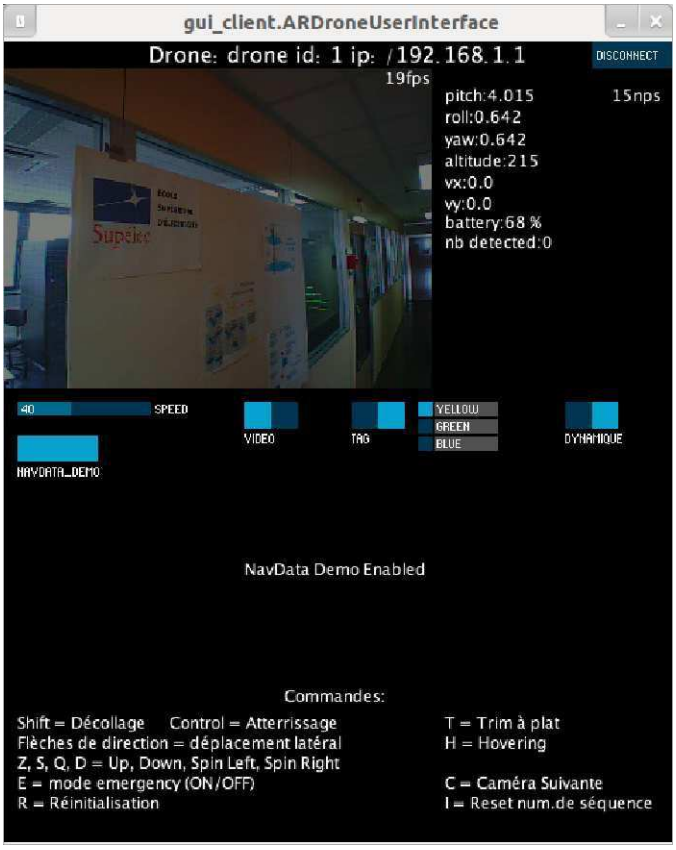


FIGURE 5.28 – Le mode NavDataDemo

Fonctionnalités concernant l'altitude Deux boutons ont été ajoutés à l'interface utilisateur concernant l'altitude du drone.

Le premier permet de sélectionner l'altitude maximale que le drone peut atteindre entre les valeurs 3 et 10 mètres, en envoyant la commande AT*CONFIG associée au paramètre « control :altitude_max ».

Le second permet d'effectuer une correction sur l'altitude affichée par l'interface utilisateur. L'altitude est mesurée par le drone par un capteur ultrason. Lorsque le drone se déplace et qu'il est incliné, la mesure est faussée par l'angle d'inclinaison du drone. En activant le mode *corrected*, une fonction *cosinus* suivant les angles de roulis et tangages est appliquée sur la valeur reçue, en effectuant une conversion des degrés aux radians.

Gestion du masque d'état du drone Les NavData contiennent un masque d'état du drone, DRONE_STATE, représenté sous la forme d'un mot de 32 bits. Ce mot contient des informations comme l'activation du mode *emergency*, l'état de la batterie ou des communications. La signification de chacun des bits de ce mot est présentée à l'annexe 10. Il est prévu dans ARDroneMines l'exploitation de ce mot par l'analyseur des NavData et par un *listener* dédié. Cependant, une confusion a été faite avec un autre masque d'état, CONTROL_STATE, situé dans le bloc des NavDataDemo. Ce masque contient les informations de vol du drone, comme posé, en l'air ou en vol stationnaire, et c'est vers lui que pointait le *parser* d'ARDRone-Mines.
La correction a été faite en changeant le nom du *listener* existant en ControlStateLis-

tener pour lever toute ambiguïté, en créant un nouveau *listener* StateMaskListener et en modifiant le code du *parser* afin de prendre en compte ces modifications.
L'affichage par la fenêtre utilisateur des informations *emergency* et *low battery* est effectué en testant l'état de ce masque d'état.
Il est à noter aussi que le SDK fournit des informations contradictoires sur la signification de certains bits, comme annoté à l'annexe 10.

5.2.4 Accompagnement des utilisateurs et des développeurs

Documentation pour les utilisateurs Une documentation à l'intention des utilisateurs de JaSMinDrone a été rédigée. Celle-ci explique les fonctionnalités de JaSMinDrone et des captures d'écran illustrent les explications. Cette documentation, rédigée en anglais, est à l'annexe 11.

Documentation pour les développeurs Une documentation à l'intention de développeurs souhaitant reprendre, améliorer ou maintenir le logiciel a été rédigée. Les diagrammes de séquences réalisés au cours de ce projet illustrent les explications. Cette documentation, rédigée en anglais, est à l'annexe 12.

Javadoc Le code source a été documenté à l'aide de l'outil Javadoc, qui permet de générer les documentations d'un code Java. Cette documentation a été réalisée grâce à l'*IDE* Netbeans utilisé dans ce projet.

Encadrement de projets étudiants

Projet SUPELEC Le projet de conception des étudiants de deuxième année de SUPELEC avait pour but d'effectuer un couplage entre JaSMinDrone et SAMOVAR [51]. Les différentes étapes de ce couplage seront décrites à la section 6.1.
Au cours de ce projet, j'ai présenté aux étudiants le drone et le logiciel JaSMinDrone. Je les ai rencontrés régulièrement afin d'effectuer des bilans sur l'avancée du projet, et lors de ces réunions il leur était expliqué en détail le fonctionnement de la partie de JaSMinDrone ou du drone concernée par l'étape en cours : ATCommands, NavData. Je leur ai aussi fourni un exemple de code effectuant la liaison par *socket* entre JaSMinDrone et Matlab ainsi que le décodage des ATCommands.

Projet ENSMN Le projet des étudiants des Mines de Nancy consistait à réaliser un simulateur de vol pour l'AR.Drone [52], en utilisant ARDroneMines et le simulateur de vol FlightGear. Ils ont développé un *proxy* permettant de faire communiquer les deux logiciels. Cependant, le projet a buté sur des limitations de FLightGear en mode connecté et il s'est avéré impossible de reproduire tous les mouvements du drone. Il est indiqué en conclusion de leur rapport qu'il serait envisageable de laisser FlightGear s'occuper de l'aspect graphique de la simulation et de laisser une autre application faire le rendu physique.
Mon rôle dans ce projet a consisté à fournir une version fonctionnelle d'ARDroneMines et de présenter aux étudiants le fonctionnement et les changements appliqués à ARDroneMines.

Diffusion du logiciel Il est envisagé de diffuser ce logiciel sous licence GPL. Cependant, certaines parties du code sont peut-être soumises à des droits d'auteurs. Des démarches sont en cours afin d'obtenir l'autorisation des auteurs.

Chapitre 6

Couplage des composants

6.1 JaSMinDrone/SAMOVAR

6.1.1 Architecture de la solution

Le but est de réaliser un couplage entre JaSMinDrone et SAMOVAR afin de contrôler un drone virtuel à partir de la même interface que pour un drone réel, et d'afficher dans cette interface les informations d'état concernant ce drone virtuel. Il n'est pas prévu de traiter dans cette solution l'affichage de la vidéo, d'une part car la vue du monde virtuel proposée par SAMOVAR est effectuée à partir d'un point fixe et ne suit pas les déplacements du drone. Il n'est pas prévu une vue « vu par le drone ». D'autre part, la vidéo devrait être encodée au format propriétaire P264 et, bien que ce format soit décrit dans le guide du SDK, le développement d'un encodeur demanderait trop de temps dans le cadre de ce projet.

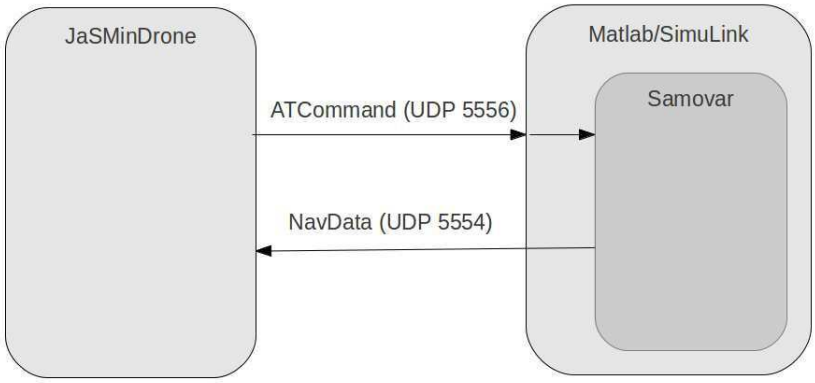


FIGURE 6.1 – L'architecture de couplage entre JaSMinDrone et SAMOVAR

La figure 6.1 illustre l'architecture de ce couplage. SAMOVAR est fondé sur Matlab/SimuLink, la première étape consiste à effectuer une liaison entre JaSMinDrone et Matlab, puis vers SAMOVAR, enfin les informations de SAMOVAR devront être fournies à JaSMinDrone.

Le modèle utilisé pour cette solution est le modèle Quadrotor, fourni avec SAMOVAR, et illustré à la figure 6.2. Le bloc Quadrotor modélise la physique de vol du drone, le bloc Keyboard Control transforme les commandes effectuées par les touches du clavier en consigne suivant les angles d'Euler et l'altitude. Les autres

blocs concernent le rendu en trois dimensions du drone et de son environnement.

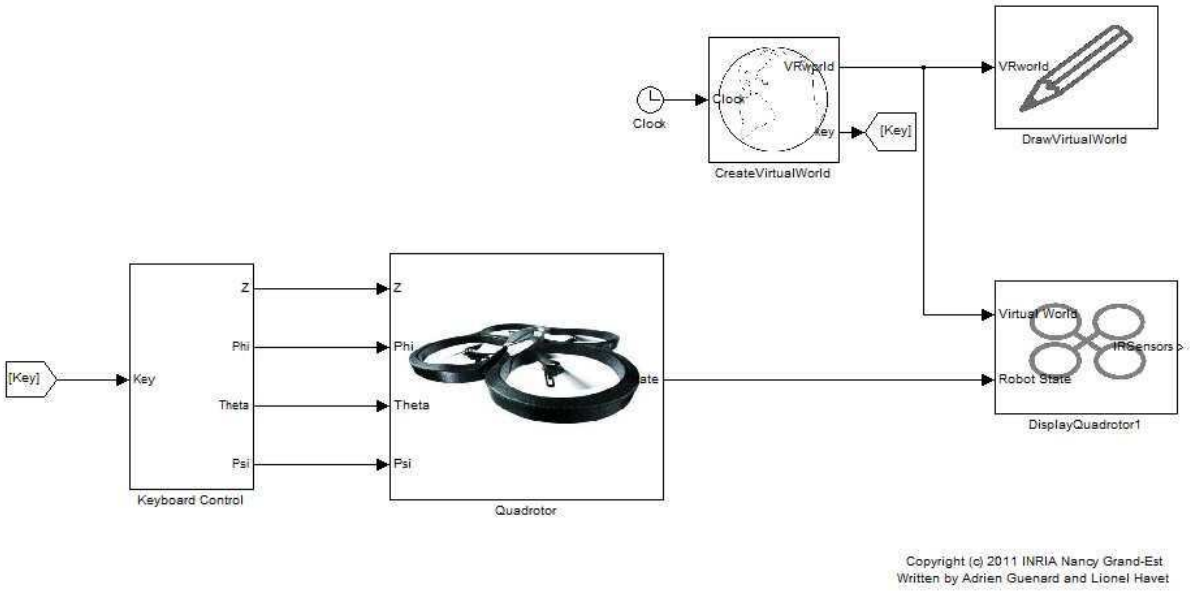


FIGURE 6.2 – Le modèle de quadcopter fourni avec SAMOVAR

6.1.2 Lien de JaSMinDrone vers Matlab

La liaison entre JaSMinDrone et Matlab consiste à recevoir les ATCommands envoyées par JaSMinDrone sur le port UDP 5556. Pour cela, les étudiants ont utilisé JUDP [53], une application disponible sur le site de Mathworks et utilisant les interfaces Java de Matlab.

Cependant, ils ont été confrontés à un problème empêchant d'effectuer la connexion. Comme indiqué dans le paragraphe « Connexion à un simulateur » 5.2.3, le port UDP 5556 était utilisé par JaSMinDrone et ne pouvait être lié à JUDP, l'erreur remontée étant *BindException : Adresse déjà utilisée*. Je leur ai fourni une version de JaSMinDrone intégrant une fonctionnalité de connexion en *local host*. J'ai aussi conçu, à titre d'exemple, une fonction Matlab utilisant JUDP et les fonctions régulières pour afficher en clair dans Matlab la signification des commandes reçues de JaSMinDrone, comme l'illustre la figure 6.3.

Cela leur a permis de créer leur propre fonction établissant la communication et traduisant les commandes reçues de JaSMinDrone.

6.1.3 Interaction entre matlab et SAMOVAR

SAMOVAR est composé de blocs imbriqués, comme l'illustre la figure 6.4. Le bloc Quadrotor est composé de blocs effectuant des tâches plus élémentaires, ces blocs étant eux-mêmes composés de fonctions.

La communication établie entre JaSMinDrone et Matlab, le travail suivant a été de développer un moyen permettant de transmettre les commandes reçues à SAMOVAR. Leur méthode a été de remplacer le bloc Keyboard Control, et d'utiliser la

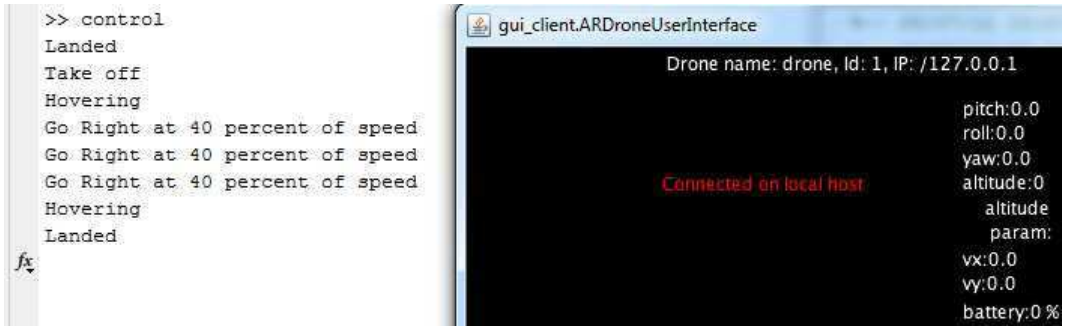


FIGURE 6.3 – L'affichage en clair dans Matlab des commandes reçues de JaSMin-Drone

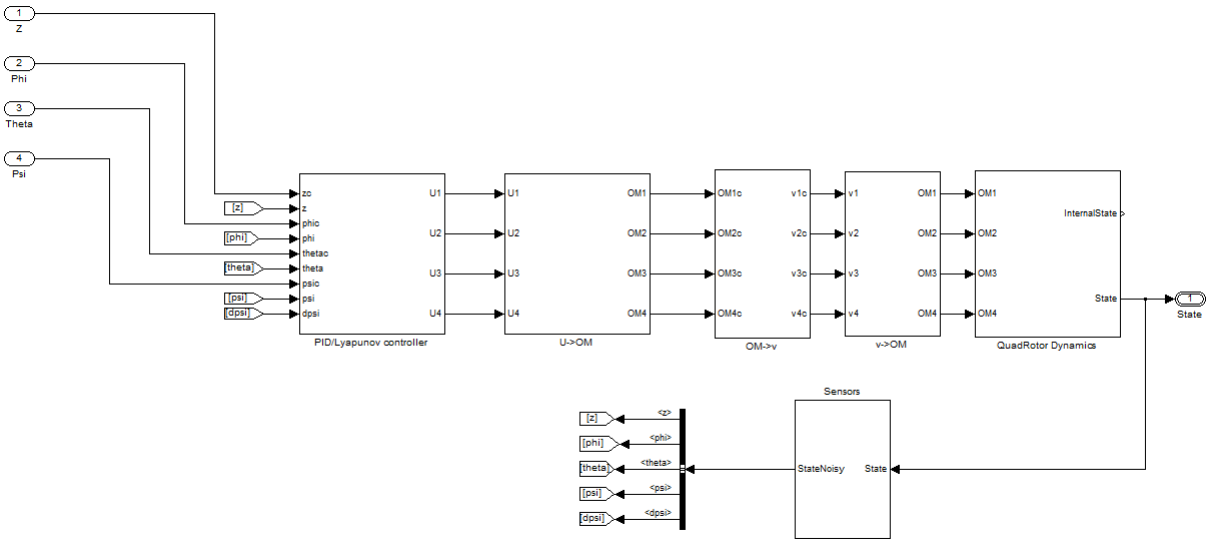


FIGURE 6.4 – Le contenu du bloc Quadrotor

fonction précédente pour créer un bloc *Embedded Matlab Function*, ce type de bloc permettant de faire appel à une fonction Matlab externe à SimuLink.
Cependant, dans un tel bloc, une fonction externe est exécutée en boucle, totalement et de manière séquentielle, sans mémoire d'état. Ainsi, à chaque fois que l'application JUDP est appelée, celle-ci crée un *socket* puis le ferme à la fin de la réception d'un paquet, comme l'illustre la capture Wireshark représentée par la figure 6.5. Le délai induit provoque des erreurs dans l'exécution du bloc. Leur solution a consisté à insérer un compteur jouant un rôle de temporisation permettant l'exécution complète de JUDP avant que celle-ci ne soit rappelée. La figure 6.6 illustre ce bloc qui remplace le bloc Keyboard Control dans le modèle du drone.

Afin d'adapter le comportement du drone virtuel avec celui du drone réel, des changements ont été effectués dans les caractéristiques du modèle contenues dans le fichier « init.m ». Les coefficients des moments d'inertie ont notamment été multipliés par 10 afin que la stabilisation du drone après un mouvement soit proche de la réalité.

Notre travail a permis de rendre fonctionnel le contrôle du drone simulé dans

Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	UDP	334	Source port: 57133
127.0.0.1	127.0.0.1	UDP	334	Source port: 60302
127.0.0.1	127.0.0.1	UDP	334	Source port: 56847
127.0.0.1	127.0.0.1	UDP	334	Source port: 48523
127.0.0.1	127.0.0.1	UDP	334	Source port: 32900
127.0.0.1	127.0.0.1	UDP	334	Source port: 39066
127.0.0.1	127.0.0.1	UDP	334	Source port: 60895
127.0.0.1	127.0.0.1	UDP	334	Source port: 60159

FIGURE 6.5 – Capture Wireshark représentant la création d'un nouveau *socket* à chaque transmission de paquet

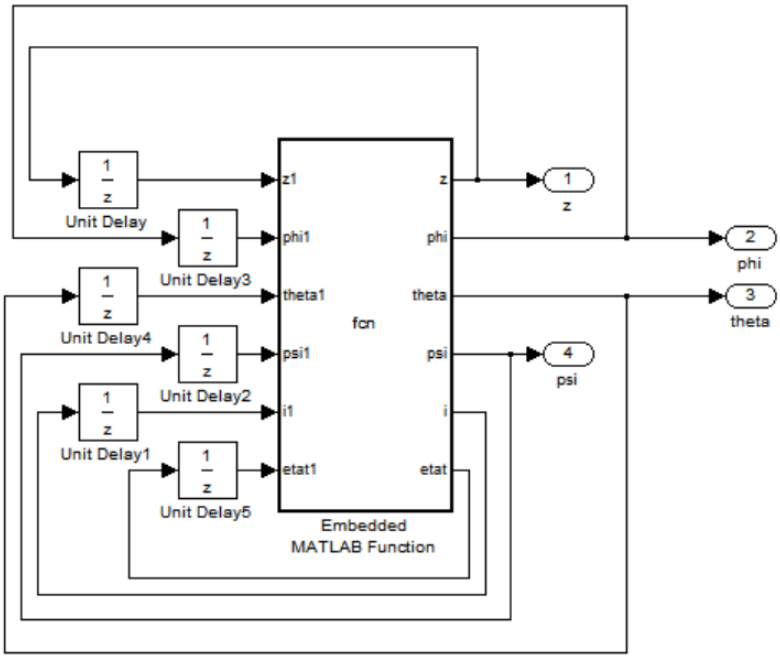


FIGURE 6.6 – Le bloc permettant la réception et le traitement des commandes de JaSMinDrone

SAMOVAR à partir de l'interface utilisateur de JaSMinDrone.

6.1.4 Lien de SAMOVAR vers JaSMinDrone

Afin de permettre la transmission des informations d'état du drone simulé vers JaSMinDrone, il est nécessaire de comprendre la structure des NavData ainsi que le traitement effectué par JaSMinDrone.

Les NavData sont transmises sous forme de paquet UDP, chaque paquet contenant plusieurs blocs d'informations. La composition de ce paquet est illustrée par la figure 6.7.

Le paquet se décompose ainsi :

– un *header*, dont la valeur est 0x55667788, sert à vérifier qu'il s'agit bien d'un paquet NavData ;

Header	Drone State	Numéro de séquence	Flag vision	Block 1 (Option)			Block ...	Checksum
				id	Taille	Données		
32 bits	32 bits	32 bits	32 bits	16	16	...	...	64 bits

FIGURE 6.7 – La composition d'un paquet NavData

- le champ *Drone state*, dont la signification a été traitée au paragraphe « Gestion du masque d'état du drone » de la section 5.2.3 ;
- le numéro de séquence, servant à l'ordonnanceur logiciel ;
- le champ *Vision flag* n'est pas utilisé avec cette version de *firmware* ;
- suivent un ou plusieurs blocs nommés ici option et contenant les informations proprement dites ;
- un bloc *Checksum* permet de vérifier l'intégrité des données reçues.

Le traitement des paquets dans JaSMinDrone est effectué par la classe NavDataParser, sous la forme de *ByteBuffer*. Le *parser* vérifie la valeur du *header* et le numéro de séquence. Si ces valeurs sont correctes, il répartit les options, identifiées par leurs *id*, dans les structures correspondantes : NavDataDemo, NavDataGyrosOffsets, NavDataPWM, etc. Ces structures sont ensuite traitées afin d'extraire les informations correspondantes aux attributs élémentaires : altitude, angles de roulis, de tangage, etc. La figure 6.8 illustre la décomposition d'un paquet capturé par Wireshark, et notamment le bloc NavDataDemo, bloc qui sera à transmettre par SAMOVAR à JaSMinDrone. Les données sont représentées sous la forme *little endian*, les octets de poids faible en premier. La valeur du *cheksum* n'est pas vérifiée par JaSMinDrone.

```
88 77 66 55            Header (0x55667788), entier (uint32)
54 04 80 0f            Drone State, entier (uint32)
b2 05 00 00            Sequence Number (0x05B2, 1458 décimale), entier (uint32)
01 00 00 00            Vision Flag, entier (inutilisé)
00 00 94 00            Début données: 00 00 -> data id (uint16) (0: navdata demo),
                                      94 00 -> taille (uint16) (148 octets, y compris data id et taille)
00 00 02 00                    Control State (uint32)
47 00 00 00                    Batterie (uint32) (0x47, 71%)
00 00 cc c2                    Théta, flottant (float32) (0xC2CC0000)
00 00 e8 c2                    Phi, flottant (float32) (0xC2E80000)
00 82 e5 47                    Psi, flottant (float32) (0x47E58200)
d6 00 00 00                    Altitude, entier (int32) (0xD6, 214 décimale)
00 00 00 00                    Vx, flottant (float32)
00 00 00 00                    Vy, flottant (float32)
00 00 00 00                    Vz, flottant (float32)
d6 9c 00 00                    Frame number, entier (uint32) (inutilisé)
00 00 00 00                    ...
```

FIGURE 6.8 – La décomposition d'un paquet NavData capturé par *Wireshark*

Concernant la communication entre SAMOVAR et JaSMinDrone, les informations à transmettre sont donc : le *header*, un numéro de séquence croissant et le bloc NavDataDemo. Celui-ci doit contenir son *id*, la taille des données et les champs altitude, thêta, psi et phi (les angles d'Euler). Pour finir, les données doivent être représentées en *little endian*.

La mise en application de ces informations par les étudiants a été similaire au lien de communication JaSMinDrone vers SAMOVAR : utilisation de l'application JUDP, d'un bloc *Embedded Matlab Function* relié aux sorties du bloc Quadrotor et d'un compteur pour pallier la création/destruction du socket.

Les figures 6.9 et 6.10 illustrent le travail final du projet de conception des étudiants de Supélec.

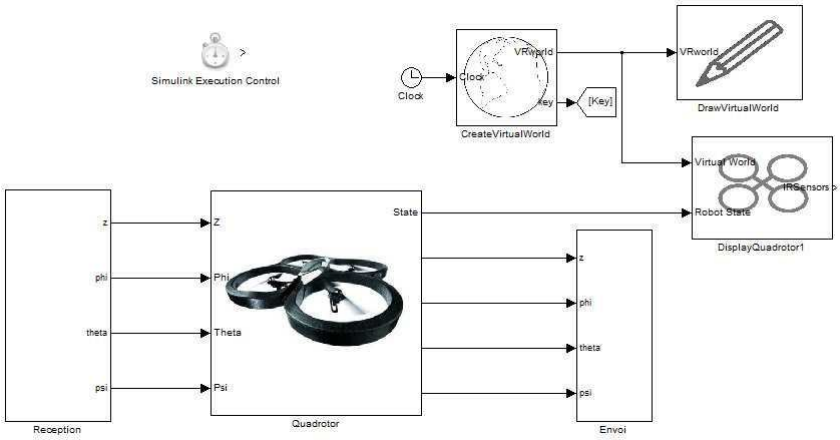


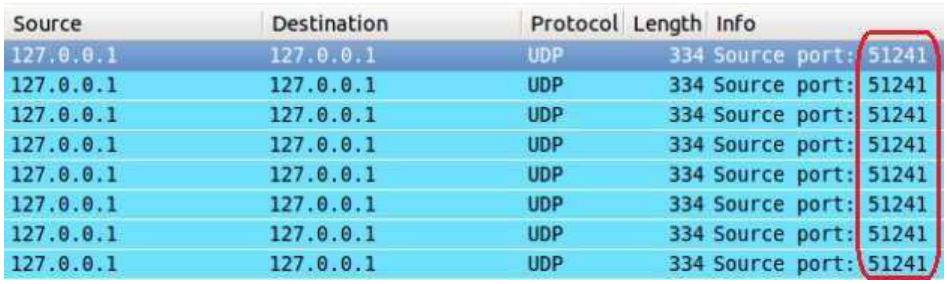
FIGURE 6.9 – Le modèle SAMOVAR permettant le couplage avec JaSMinDrone

FIGURE 6.10 – Contrôle par JaSMinDrone d'un drone simulé par SAMOVAR

6.1.5 Amélioration par l'utilisation de S-Function

Les limitations de la solution fournie par les étudiants viennent de l'utilisation de l'application JUDP dans des blocs *Embedded Matlab Function*. Les *sockets* utilisés sont créés et détruits à chaque envoi ou réception de paquet.
La solution a été apportée par Adrien Guénard en utilisant les S-Function de Simu-Link. Ces fonctions permettent d'utiliser des variables d'état accessibles par toutes les fonctions et qui sont gardées en mémoire lors de l'exécution d'une simulation. La variable correspondant au *socket* a été définie comme étant « globale », ce qui évite la création d'un nouveau *socket* à chaque transmission d'un paquet, comme l'illustre la capture Wireshark représentée par la figure 6.11.

Il subsistait néanmoins un problème dans le code, et lorsque le drone simulé était décollé, le fait de vouloir le déplacer le collait au sol, passant son altitude à zéro. En recherchant dans le code de la S-Function recevant les ATCommands, il s'est avéré que la variable définissant le drone comme étant en l'air ou posé était initialisée à chaque traitement d'une commande de déplacement, et le drone se posait. J'ai modifié le code afin d'initialiser cette variable après la création du *socket*. Après cette modification, le comportement du drone est redevenu normal.



Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241
127.0.0.1	127.0.0.1	UDP	334	Source port: 51241

FIGURE 6.11 – Capture Wireshark représentant l'utilisation d'un seul *socket* après l'utilisation de S-Function

6.1.6 Vol en formation de drones simulés

SAMOVAR contient un modèle permettant le vol en formation de drones simulés. L'utilisateur dirige un drone *leader* et un algorithme permet le maintien de six autres drones aux alentours du *leader*.
J'ai modifié le code du *leader* afin d'ajouter les S-Function précédentes. La figure 6.12 illustre les modifications apportées. Le bloc « Réception » est relié au PID gérant les déplacements, le bloc « Envoi » est relié aux paramètres d'états du drone.

Cette solution permet le contrôle du *leader* et la réception de ses informations d'état, comme l'illustre la figure 6.13.

6.2 JaSMinDrone/OMNet/SAMOVAR

6.2.1 Étude de la solution Veins

L'étude de la solution Veins [37] permet de comprendre la manière d'effectuer un couplage entre OMNet++ et un autre simulateur.

La solution Veins est composée d'une version modifiée du *framework* MiXiM et d'un script en langage Python. Ce script crée un *proxy* permettant la communication par TCP entre SUMO et OMNet++. Les modifications apportées à MiXiM consistent en l'ajout de modèles spécifiques aux réseaux mobiles ad hoc, et plus particulièrement les *VANET* (*Vehicular Ad-Hoc Network*).
Veins utilise TraCI (*Traffic Control Interface)*, un composant de SUMO permettant la transmission des valeurs des objets de SUMO. Des modules ont été ajoutés à MiXiM afin de gérer les communications avec TraCI en passant à travers le proxy TCP. Un module permet la gestion du scénario à dérouler et de la connexion au proxy (module *TraciScenarioManager*) et un module effectue le déplacement des nœuds en fonction de la position reçue de SUMO (module *TraCIMobility*).

Veins étant basé sur MiXiM, la modélisation des nœuds n'est effectuée que jusqu'à la couche réseau. J'ai donc décidé d'utiliser le *framework* INETMANET afin de pouvoir simuler la communication entre deux drones en utilisant la pile TCP/IP. La démarche adoptée va consister à établir d'abord un couplage unidirectionnel entre SAMOVAR et OMNet++ pour un seul drone, puis pour deux drones. Enfin, une communication bidirectionnelle sera effectuée.

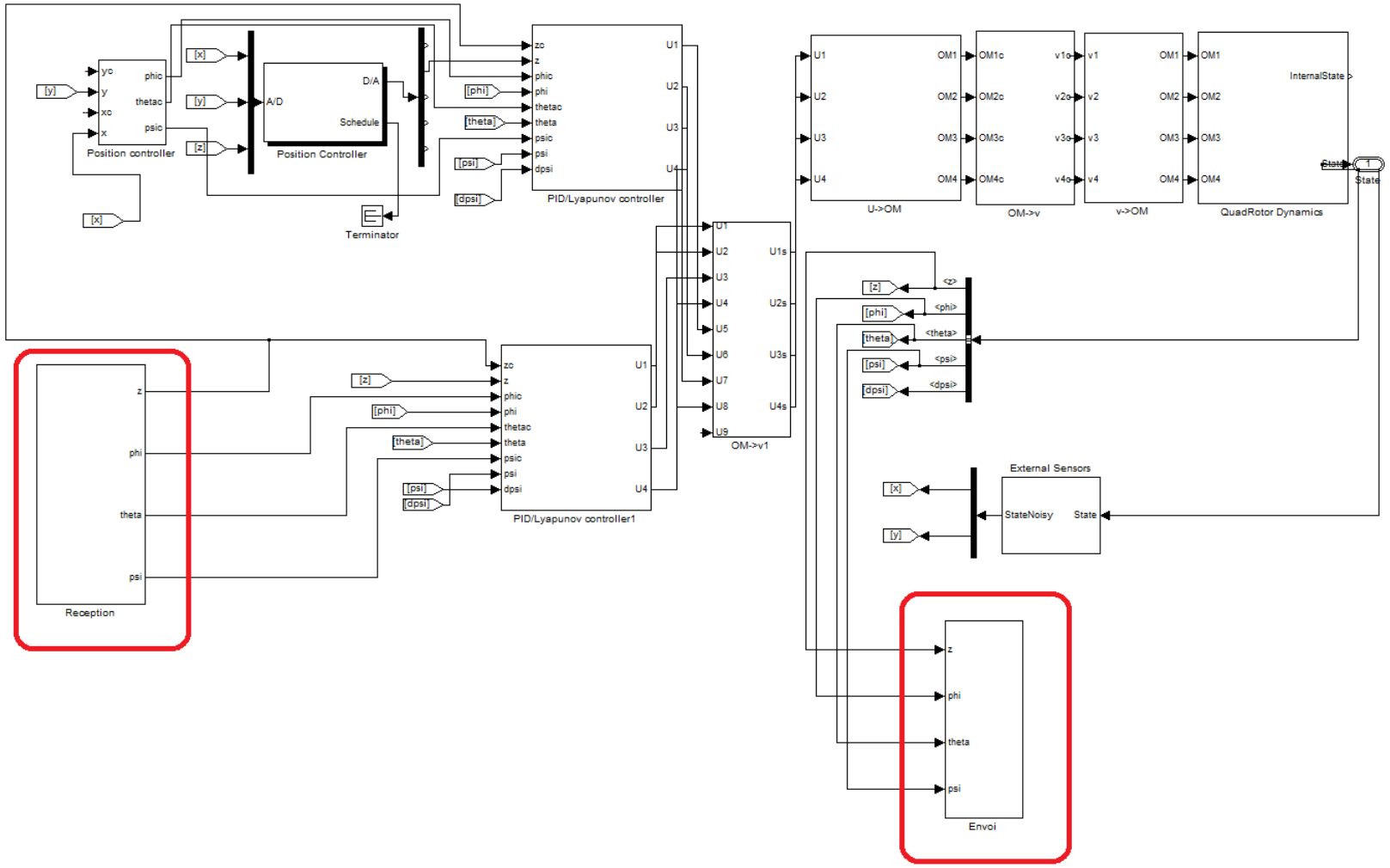


FIGURE 6.12 – Les blocs Réception et Envoi ajoutés au code du *leader*

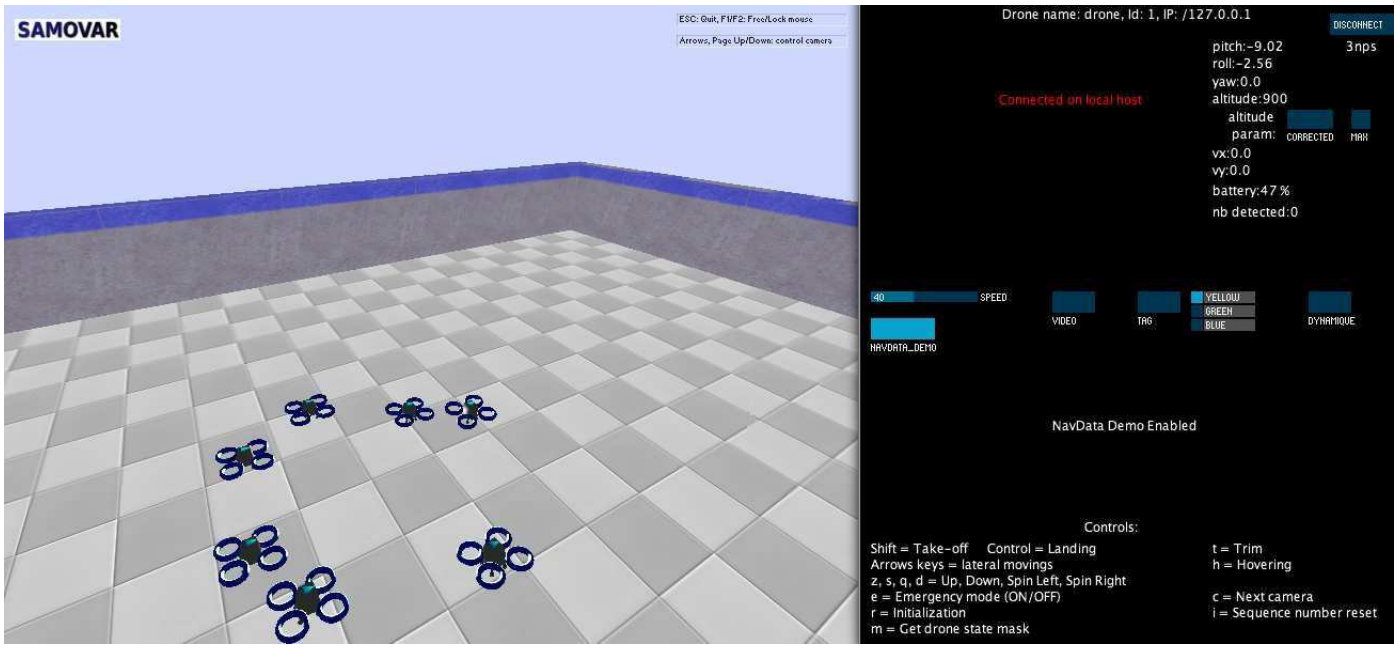


FIGURE 6.13 – Un vol en formation de drones simulés, le *leader* étant contrôlé par JaSMinDrone

6.2.2 Communication de SAMOVAR à OMNet++

Pour mettre au point et étudier des protocoles de réseaux ad hoc de nombreux modèles de mobilité ont été développés pour être utilisé dans les simulateurs réseaux. Nous souhaitons ici que la position des nœuds dans le simulateur réseaux reflète les déplacements des drones en fonction des ordres qu'ils ont reçus et de leurs capacités physique. Cette information est délivrée par le modèle de drone développé dans SAMOVAR. Conformément au chapitre 2 et plutôt que de développer un nouveau modèle physique dans OMNet++, nous allons récupérer les informations fournies par SAMOVAR.

L'architecture retenue pour établir une communication de SAMOVAR à OMNet++ est illustrée à la figure 6.14. Il s'agit d'établir une liaison par *socket* TCP en passant par un proxy jouant le rôle de serveur.

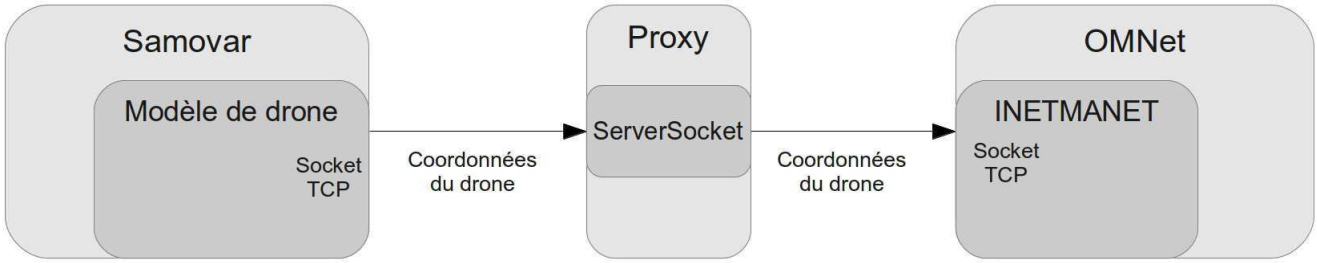


FIGURE 6.14 – Architecture de la solution permettant la communication entre SAMOVAR et OMNet++

Le proxy serveur Développé en Java, ce proxy serveur est une classe permettant d'établir la communication entre SAMOVAR et OMNet++ et de transmettre de manière transparente — aucune manipulation n'est effectuée sur les données — les informations de coordonnées du drone à OMNet++. Ceci est effectué en deux étapes.

D'abord, un *ServerSocket*, à l'écoute des demandes de connexions sur le port TCP 5560, crée un *socket* pour SAMOVAR puis pour OMNet++.

Ensuite, deux flux sont créés : un flux entrant depuis SAMOVAR et un flux sortant vers OMNet++. Il suffit ensuite de lire le flux entrant dans un message et d'écrire ce même message dans le flux sortant, en faisant une boucle tant qu'on reçoit des informations de SAMOVAR.

L'ordre de lancement des programmes a de l'importance, car le premier à demander une connexion au proxy sera l'émetteur des informations, le second le récepteur. Il convient donc d'exécuter SAMOVAR avant OMNet++ pour cette architecture.

L'utilisation de ce proxy permet l'exécution des différents composants (proxy, SAMOVAR, OMNeT++) sur un même machine ou bien de manière réparties sur des machines différentes.

Le modèle SAMOVAR Le modèle à créer doit pouvoir délivrer les informations concernant la position du drone au proxy par un *socket* TCP. Ce modèle sera fondé sur le modèle issu du couplage JaSMinDrone/SAMOVAR.

Le premier travail a consisté à récupérer les coordonnées du drone afin de connaitre sa position à tout instant. En étudiant la composition du bloc Quadrotor, il apparait qu'un sous-bloc nommé QuadRotor Dynamics génère l'état du drone et le distribue par un bus au sous-bloc Sensors. Les coordonnées du drone sont situées dans ce bus, mais le sous-bloc Sensors ne les utilisant pas, elles ne se retrouvaient pas en sortie de ce sous-bloc. La solution a consisté établir un *by-pass* entre les connecteurs d'entrée et de sortie du sous-bloc Sensors pour les variables de position x, y et z, puis à connecter ces variables au connecteur de sorties du bloc supérieur Quadrotor. Ce travail est illustré par la figure 6.15.

Il a fallu ensuite concevoir une S-Function effectuant la connexion au proxy et lui envoyant les coordonnées du drone. La S-Function créée, nommée SendTCP, utilise la bibliothèque *tcpip* incluse dans le module *Instrument Control Toolbox* de Matlab.

Cette S-Function se décompose en plusieurs fonctions. Il est indiqué dans la fonction *setup* le nombre de variables d'entrée, de sortie ainsi que le nombre de paramètres disponibles par la boite de dialogue graphique. Dans sa fonction d'initialisation, elle crée un *socket*, déclaré en tant que variable globale, en effectuant une demande au proxy serveur :

```
function InitConditions(block)

global socketSendTCP

port = block.DialogPrm(1).Data;
```

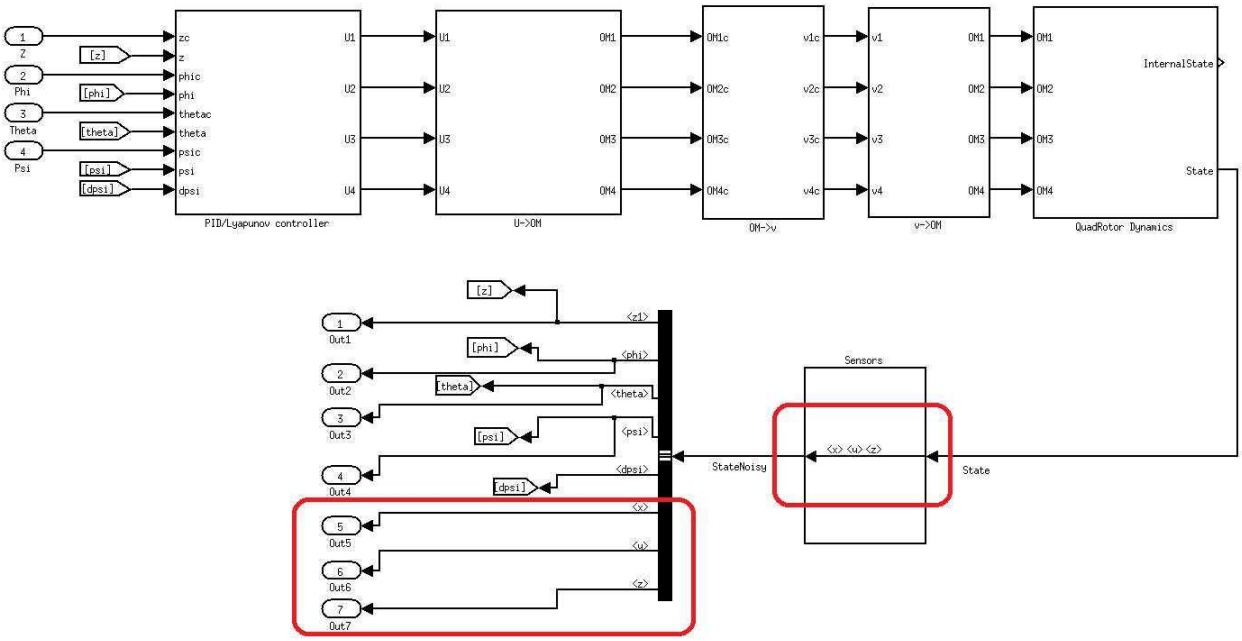


FIGURE 6.15 – Récupération des coordonnées du drone dans le bus *state*, *by-pass* à l'intérieur du sous-bloc Sensors et ajout au connecteur de sortie

```
host = block.DialogPrm(2).Data;

socketSendTCP = tcpip(host, port);
set(socketSendTCP, 'Timeout', 60);

fopen(socketSendTCP);

%endfunction
```

Les paramètres *host* et *port* sont déclarés en tant que *DialogPrm*. Cela évite de coder ces paramètres en dur et permet de changer leurs valeurs par la boite de dialogue graphique.

La fonction *update* est exécutée à chaque itération de la simulation. Elle récupère les variables d'entrées de la S-Function, ici les coordonnées du drone, et les transmet au proxy à travers le *socket* créé précédemment.

```
function Update(block)

global socketTCP

x1=round((block.InputPort(1).Data)+250); % Input 1 = x drone 1
y1=round(250-(block.InputPort(2).Data)); % Input 2 = y drone 1
z1=round(block.InputPort(3).Data); % Input 3 = z drone 1
```

```
strx1 = int2str(x1);
stry1 = int2str(y1);
strz1 = int2str(z1);

mssgAller=['x' strx1 'y' stry1 'z' strz1 'a'];
fprintf(socketTCP, mssgAller);

%endfunction
```

Un calcul est effectué sur les coordonnées afin d'adapter le repère de SAMOVAR à celui d'OMNet++ et que les nœuds dans OMNet++ soient placés au même endroit que les drones dans SAMOVAR. Des séparateurs — les lettres x, y, z et a — sont insérés dans la chaîne envoyée afin de permettre la séparation des coordonnées par OMNet++.

Le modèle créé permettant l'envoi des coordonnées d'un drone est illustré par la figure 6.16.

INETMANET Afin d'effectuer le couplage avec SAMOVAR, deux étapes ont été nécessaires : la création d'un modèle de mobilité effectuant la connexion avec le proxy serveur et la mise à jour de la position des nœuds, et la création d'un réseau utilisant ce modèle de mobilité.

Modèle de mobilité Différents modèles de mobilité sont fournis avec le *framework* INETMANET. Aucun modèle ne permettant la connexion à un programme extérieur, j'ai décidé de concevoir un nouveau modèle.

Un modèle de mobilité définit un module OMNet++ et il est ainsi utilisable par chaque type de nœud modélisé. Le type de modèle utilisé par les nœuds est spécifié par l'utilisateur dans le fichier de configuration de la simulation « omnetpp.ini ».

Un modèle est constitué de trois fichiers :

- un fichier NED permet d'indiquer les paramètres de mobilité que l'utilisateur pourra modifier dans le fichier « omnetpp.ini » et de fixer des valeurs par défaut ;
- un module correspond à une classe C++, les méthodes et les attributs d'un modèle sont situés dans des fichiers « cc » et « h ».

L'utilisation d'un langage objet permet de ne pas avoir à écrire un modèle en ne partant de rien et de pouvoir utiliser un modèle déjà existant. Ainsi, le modèle CouplageMobility que j'ai créé étend le modèle de base MovingMobilityBase, comme l'illustre le fichier NED suivant :

```
simple CouplageMobility extends MovingMobilityBase
{
    parameters:
```

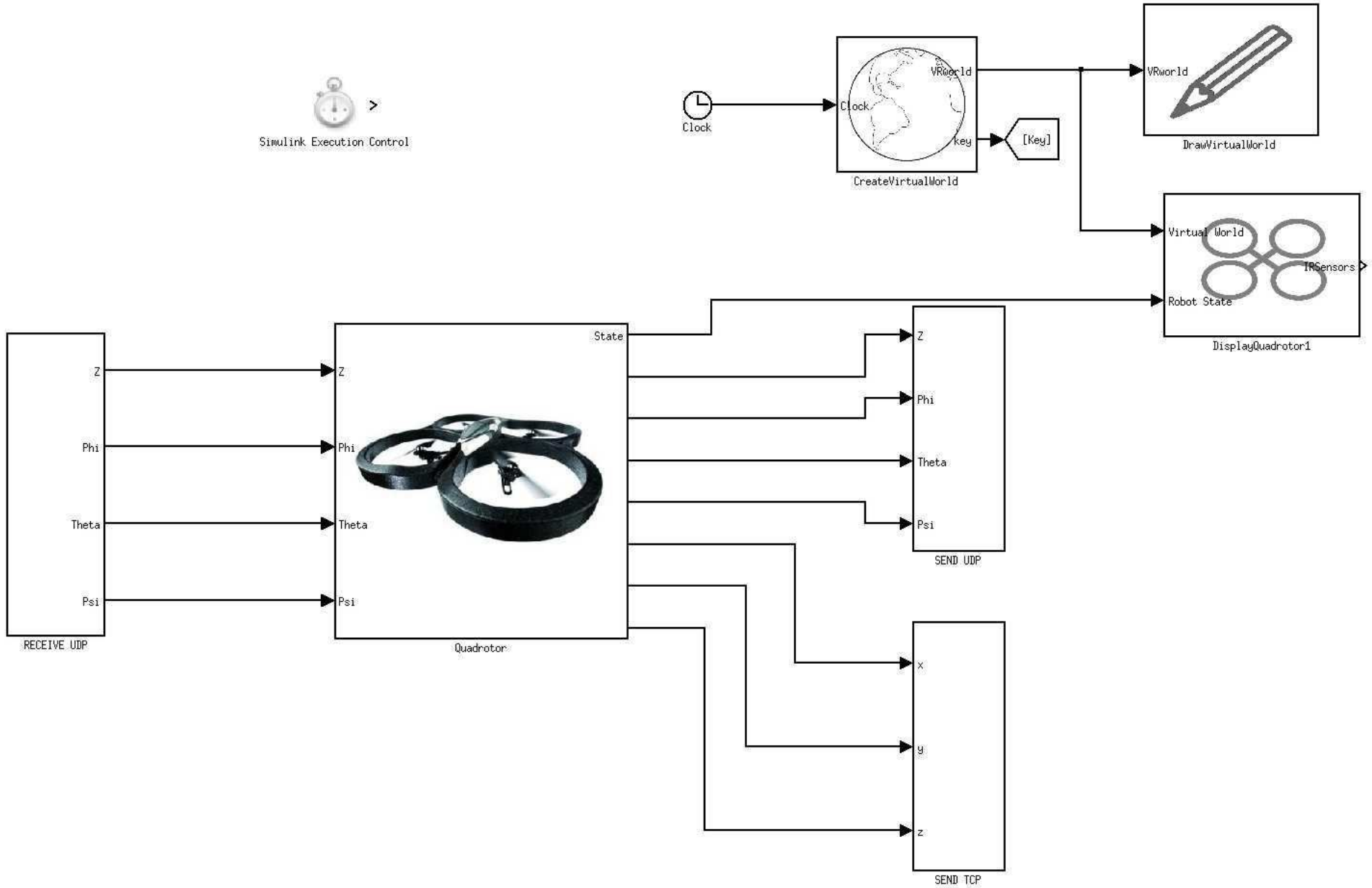


FIGURE 6.16 – Le modèle d'un drone intégrant le couplage avec JaSMinDrone et permettant d'envoyer sa position par *socket*

```
        double initialX @unit(m) = default(uniform(this.constraintAreaMinX,
            this.constraintAreaMaxX));
        double initialY @unit(m) = default(uniform(this.constraintAreaMinY,
            this.constraintAreaMaxY));
        double initialZ @unit(m) = default(uniform(this.constraintAreaMinZ,
            this.constraintAreaMaxZ));
        bool initFromDisplayString = default(true);
        int droneID; // ID to be used in the simulation for each managed drone
        int hostNumber; // number of drones involved in the simulation;
        string host = default("localhost"); // Server hostname
        int port = default(5560);   // Server port
        @class(CouplageMobility);
}
```

Les paramètres disponibles pour l'utilisateur sont le nombre de nœuds, leurs ID et leurs positions initiales, ainsi que les paramètres de connexion au proxy serveur.

Les fichiers « cc » et « h » doivent au minimum contenir les méthodes initialize(), indiquant les attributs utilisés par le modèle et les méthodes appelées à l'initialisation, et finish() permettant d'arrêter la simulation proprement.

Dans le modèle de mobilité créé, la demande de connexion au proxy serveur et la création d'un *socket* sont effectuées par une méthode connect(), appelée à l'initialisation. Cette méthode utilise les paramètres *host* et *port* définis dans le fichier NED. Le modèle étant exécuté par chaque nœud et afin d'éviter des erreurs lors de la création de multiples *sockets*, un contrôle est effectué sur l'ID du drone, et seul le nœud ayant l'ID 0 fait appel à cette méthode.

Pour recevoir les informations venant du proxy serveur, j'ai développé une méthode receiveSamMessage() renvoyant une chaîne de caractères contenant les coordonnées du drone :

```
std::string CouplageMobility::receiveSamMessage() {

    char bufferReception[80];
    memset(&bufferReception, 0, sizeof(bufferReception));

    ::recv(MYSOCKET, bufferReception, sizeof(bufferReception), 0);

    return std::string(bufferReception);

}
```

Enfin, la mise à jour des coordonnées des nœuds est effectuée par la méthode move(). Cette méthode appelle la méthode receiveSamMessage() précédemment définie et analyse la chaîne de caractères reçue afin de séparer les paramètres x, y et z du drone :

```
int posPoint1= mssg.find('x',0);
int posPoint2= mssg.find('y',posPoint1+1);
int posPoint3= mssg.find('z',posPoint2+1);
int posPoint4= mssg.find('a',posPoint3+1);

std::string mssgX1 = mssg.substr(posPoint1+1,(posPoint2-(posPoint1+1)));
const char * posX1 = mssgX1.c_str();
std::string mssgY1 = mssg.substr(posPoint2+1,(posPoint3-(posPoint2+1)));
const char * posY1 = mssgY1.c_str();
std::string mssgZ1 = mssg.substr(posPoint3+1,(posPoint4-(posPoint3+1)));
const char * posZ1 = mssgZ1.c_str();
```

Ces paramètres sont ensuite convertis au format « double » afin d'être compatibles avec la fonction de mise à jour des coordonnées, *lastPosition* :

```
double targetX1 = (atof(posX1));
double targetY1 = (atof(posY1));
double targetZ1 = (atof(posZ1));

lastPosition.x = targetX1;
lastPosition.y = targetY1;
lastPosition.z = targetZ1;
```

Définition du réseau La seconde étape a consisté à créer un réseau utilisant le modèle de mobilité précédent.

Un réseau est défini dans OMNet++ par un fichier NED et par un fichier de configuration. Le fichier NED décrit les composants du réseau. J'ai ici choisi d'utiliser des composants déjà existants dans INETMANET car ils correspondaient aux besoins de la simulation :

- le nœud de type *AdhocHost* contient une interface sans fil et supporte les protocoles IPv4, TCP, UDP et ICMP ;
- le module *ChannelControl* reçoit les informations de mouvement des nœuds et détermine les distances de communication et d'interférence ;
- le module *IPv4NetworkConfigurator* effectue la configuration IP des nœuds du réseau.

Le listing suivant représente le code du fichier Couplage.ned du réseau créé :

```
network Couplage
{
    parameters:
        int numHosts;
    submodules:
        host[numHosts]: AdhocHost {
            parameters:
```

```
            @display("i=block/drone");
        }
        channelControl: ChannelControl {
            parameters:
                @display("p=60,50");
        }
        configurator: IPv4NetworkConfigurator {
            config=xml("<config><interface hosts='*' address='192.168.x.x'
                netmask='255.255.0.0'/></config>");
            @display("p=140,50");
        }
}
```

OMNet++ permet une représentation graphique des réseaux décrits par un fichier NED, le réseau CouplageSam est illustré par la figure 6.17.

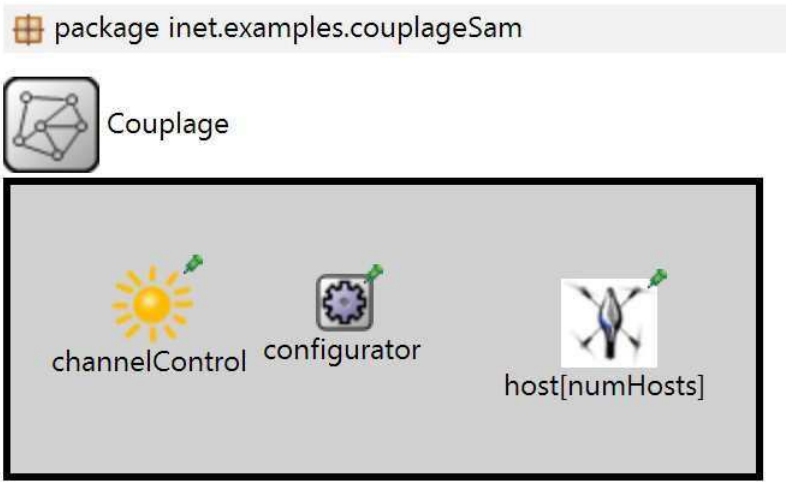


FIGURE 6.17 – Représentation graphique du réseau CouplageSam

Enfin, le fichier de configuration omnetpp.ini contient les paramètres utiles à la simulation comme le nombre de nœuds, les paramètres de la couche physique (fréquence, puissance, etc.), les paramètres du protocole ICMP permettant d'effectuer des pings et les paramètres du modèle de mobilité.

Les paramètres suivants de mobilité ont été définis :

```
# mobility
**.host*.mobilityType = "CouplageMobility"
**.host*.mobility.host = "localhost"
**.host[*].mobility.port = 5560
**.host*.mobility.updateInterval = 50ms
```

Ces paramètres permettent d'utiliser le modèle de mobilité créé, de définir un intervalle de mise à jour des informations et de se connecter au proxy serveur.

Il est possible de définir plusieurs configurations différentes de réseau et de sélectionner quelle configuration exécutée. Afin d'effectuer le couplage avec SAMOVAR simulant un seul drone, la configuration suivante a été faite :

```
[Config CouplageUnNoeud]
description = "host1 moves"
*.numHosts = 1
**.mobility.hostNumber = 1
*.host[0].pingApp[*].destAddr = ""
**.host[0].mobility.initialX = 250m
**.host[0].mobility.initialY = 250m
**.host[0].mobility.droneID = 0
```

L'ensemble de ces composants (l'interface de contrôle, le proxy serveur, le modèle SAMOVAR, le modèle de mobilité et le réseau OMNet++) permettent d'effectuer une simulation dans laquelle un drone simulé est piloté par JaSMinDrone et dont la représentation sous la forme d'un nœud ad hoc dans OMNet++ suit les mouvements décidés par l'utilisateur. La figure 6.18 illustre le fonctionnement d'une telle simulation.

Passage à deux drones Afin de pouvoir exécuter un scénario dans lequel les nœuds représentant des drones effectuent un échange de pings, les composants doivent être modifiés.

Il n'est pas possible dans SAMOVAR de changer le modèle dynamiquement. Un nouveau modèle contenant deux drones doit être développé.

Les changements consistent à ajouter deux blocs supplémentaires, un bloc Quadrotor et un bloc DisplayQuadRotor, ainsi qu'à modifier la S-Function afin que celle-ci prenne en compte les entrées supplémentaires que représentent les coordonnées du deuxième drone. Le message envoyé est lui aussi modifié et de nouveaux séparateurs permettent de délimiter les différentes coordonnées, comme le représentent le listing suivant :

```
strx1 = int2str(x1);
stry1 = int2str(y1);
strz1 = int2str(z1);

strx2 = int2str(x2);
stry2 = int2str(y2);
strz2 = int2str(z2);

mssgAller=['x' strx1 'y' stry1 'z' strz1 'a' strx2 'b' stry2 'c' strz2 'f'];
fprintf(socketTCP, mssgAller);
```

Ce deuxième drone n'est pour l'instant pas prévu pour être contrôlé, aussi les entrées du bloc Quadrotor ne sont pas connectées, le drone restera immobile pendant la simulation.

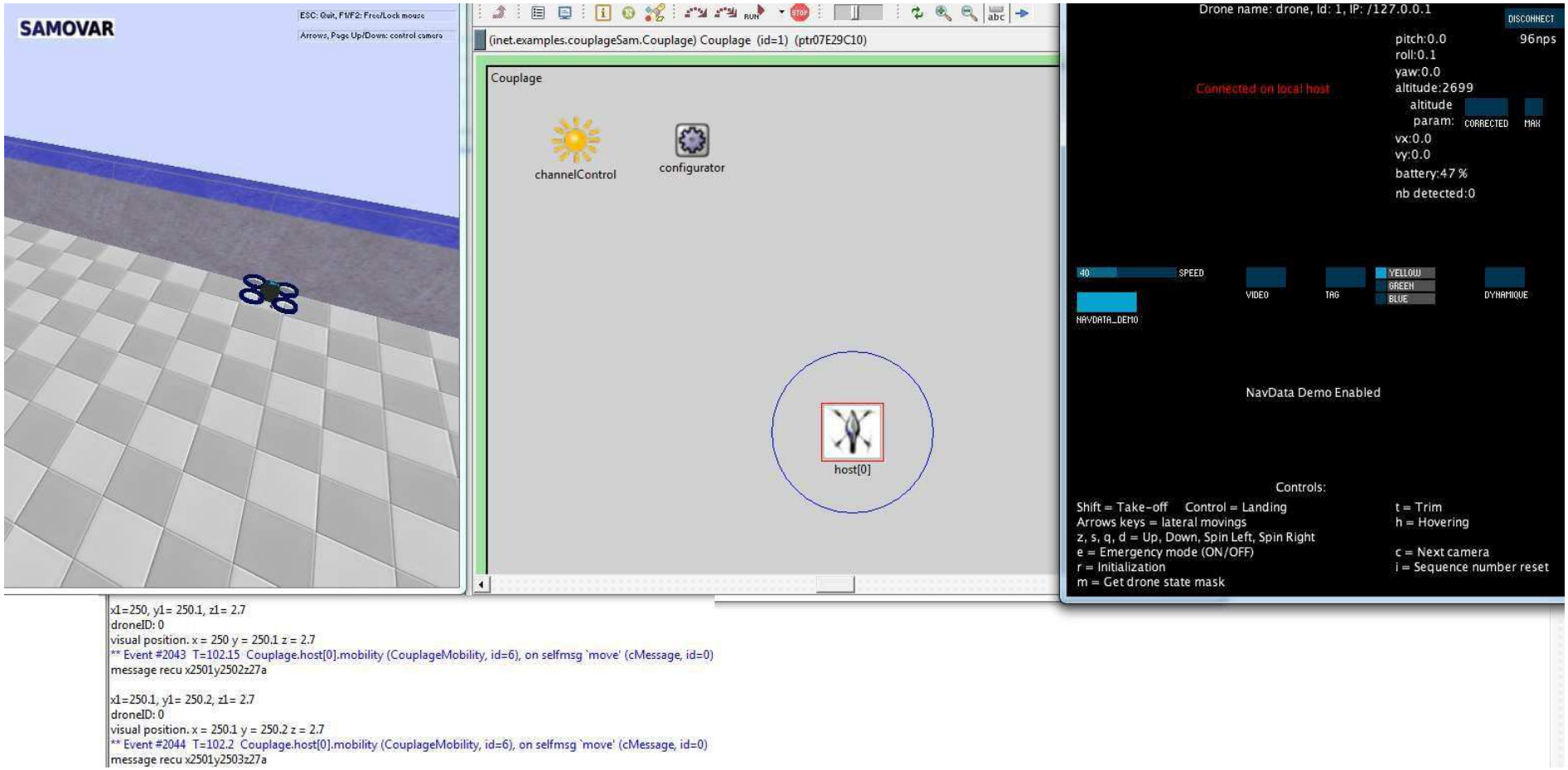


FIGURE 6.18 – Le couplage de l'ensemble des composants développés permet le contrôle d'un drone simulé et du nœud le représentant

Le modèle complet permettant la simulation de deux drones et effectuant le couplage avec les composants précédents est illustré par la figure 6.19.

Le fichier de configuration omnetpp.ini permet de réaliser plusieurs scénarios de simulation de manière simple. Ainsi, il suffit de rajouter les lignes suivantes pour dérouler une simulation comportant deux nœuds, le nœud ayant l'ID 0 effectuant régulièrement un ping vers le nœud 1 :

```
[Config CouplageDeuxNoeuds]
description = "host0 pinging host1"
*.numHosts = 2
**.mobility.hostNumber = 2
*.host[0].pingApp[*].destAddr = "host[1]"
**.host[0].mobility.initialX = 200m
**.host[0].mobility.initialY = 250m
**.host[0].mobility.droneID = 0
**.host[1].mobility.initialX = 300m
**.host[1].mobility.initialY = 250m
**.host[1].mobility.droneID = 1
```

Le modèle de mobilité doit lui aussi être modifié afin de tenir compte de la nouvelle structure des informations reçues. Les changements consistent en l'ajout des lignes suivantes dans la méthode move() :

```
if (droneID == 0)
    {
        std::string mssg = receiveSamMessage();

                'suite du code initial'

     }

if (hostNumber == 2)
     {
      int posPoint5= mssg.find('b',posPoint4+1);
      int posPoint6= mssg.find('c',posPoint5+1);
      int posPoint7= mssg.find('f', posPoint6+1);

      std::string mssgX2 = mssg.substr(posPoint4+1,(posPoint5-(posPoint4
          +1)));
      const char * posX2 = mssgX2.c_str();
      std::string mssgY2 = mssg.substr(posPoint5+1,(posPoint6-(posPoint5
          +1)));
      const char * posY2 = mssgY2.c_str();
      std::string mssgZ2 = mssg.substr(posPoint6+1,(posPoint7-(posPoint6
          +1)));
```

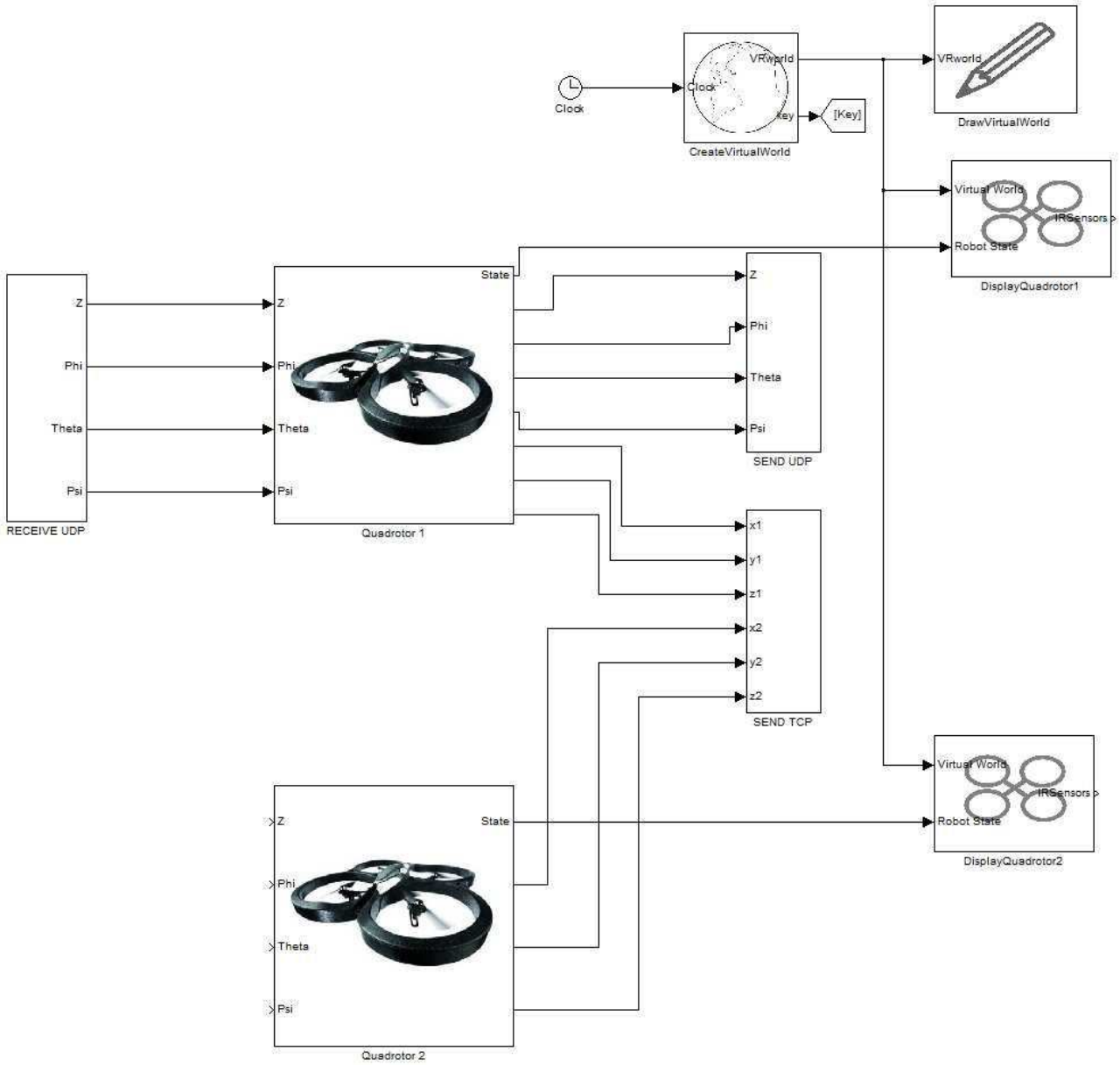


FIGURE 6.19 – Le modèle SAMOVAR comportant deux drones et permettant le couplage avec OMNet++

```
        const char * posZ2 = mssgZ2.c_str();

        ::targetX2 = (atof(posX2));
        ::targetY2 = (atof(posY2));
        ::targetZ2 = (atof(posZ2);
        }

if (droneID == 1)
    {
     lastPosition.x = ::targetX2;
     lastPosition.y = ::targetY2;
     lastPosition.z = ::targetZ2;
    }
```

La difficulté est que seul le nœud ayant l'ID 0 a connaissance du *socket* et donc seul lui peut appeler la méthode receiveSamMessage(). Les coordonnées du deuxième drone subissent les mêmes opérations que les coordonnées du premier drone. Enfin la mise à jour des coordonnées concernant le deuxième drone ne doit s'effectuer que lorsque celui-ci appelle la méthode move(). Cette condition est réalisée par le test sur l'ID égal à 1.

Ces modifications permettent d'effectuer une simulation mettant en scène deux drones représentés dans OMNet++ par deux nœuds, initialement éloignés et hors de portée de réception l'un de l'autre. Le premier nœud effectue régulièrement des pings vers le deuxième nœud, lorsque l'utilisateur déplace par l'interface de contrôle le premier drone, et donc le nœud correspondant, et que les nœuds sont à portée de réception, toutes les étapes du protocole ICMP permettant un ping (émission et réception d'une réponse) sont simulées par OMNet++.
Le déroulement d'une telle simulation est illustré par la figure 6.20. Le cercle bleu entourant les nœuds représente la portée de réception.

6.2.3 Communication de OMNet++ à SAMOVAR

Un cas d'étude de la plate-forme AETOURNOS est le vol de drones en formation. Pour établir et maintenir la cohérence de la formation, les drones peuvent utiliser des informations visuelles ; le projet veut étudier l'utilisation d'autres informations, issues du réseau. L'idée est que les drones tendent à adopter des positions leur permettant de maintenir la communication radio entre eux. Pour cela, ils pourraient utiliser des informations sur la puissance de signaux reçus. Cependant cela n'a pas été possible pour ce premier démonstrateur ; à la place les drones s'envoient des requêtes ping et la valeur du RTT (*Round-Trip delay Time*) sert à évaluer la distance les séparant. L'nevoi de ping et le calcul du RTT existent déjà dans le simulateur réseau OMNeT++, il s'agit donc maintenant d'établir une communication dans le sens OMNeT++ vers SAMOVARpour le transfert de cette information. Pour cela, les différents composants présentés la section 6.2.2 doivent être modifiés.

Proxy serveur bidirectionnel La modification du proxy serveur a pour but de rendre la communication bidirectionnelle. Pour cela, deux nouveaux flux sont créés,

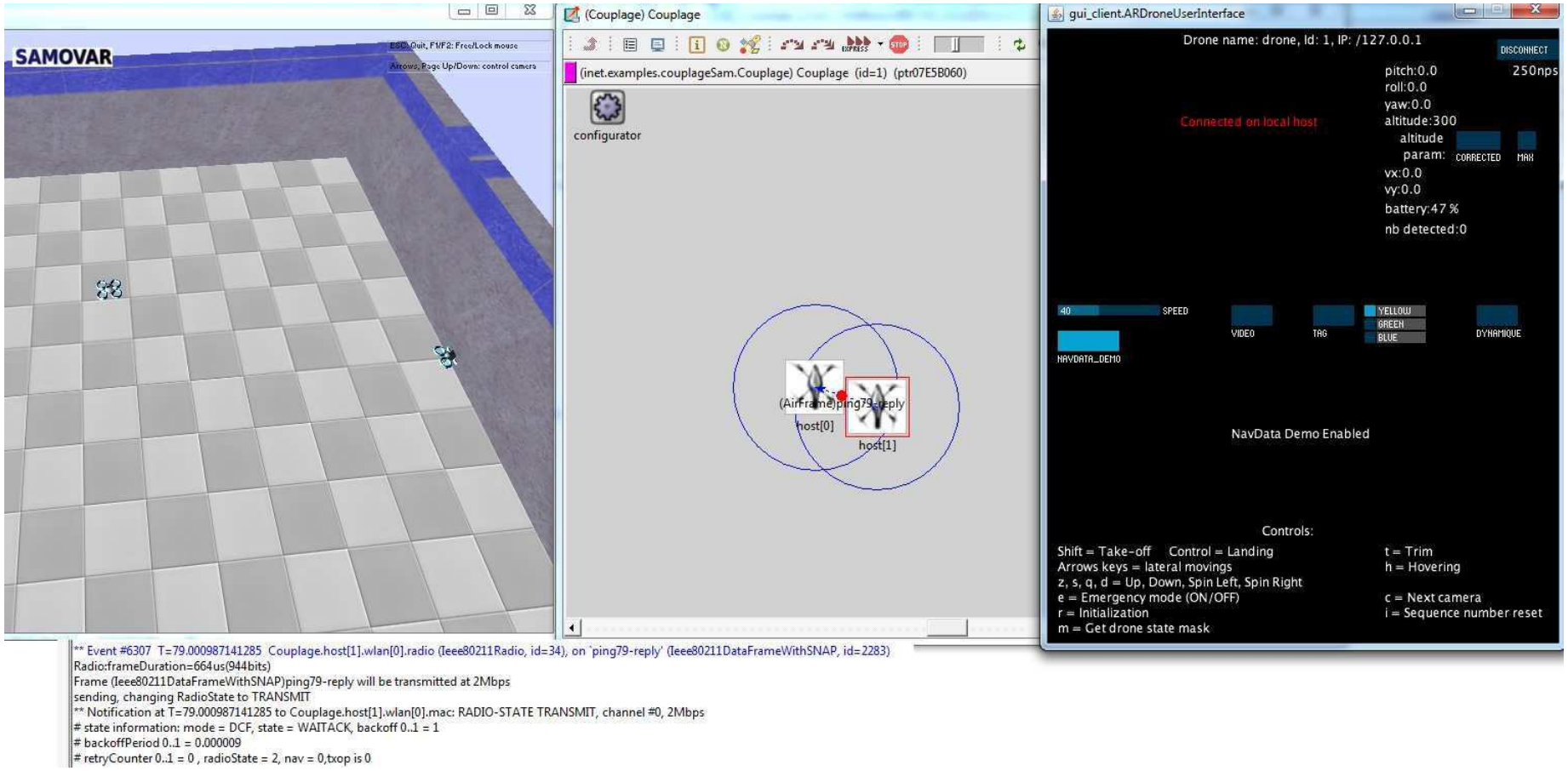


FIGURE 6.20 – Simulation mettant en jeu deux drones représentés par deux nœuds s'échangeant des messages ping

un entrant et un sortant. Ces flux sont insérés dans la boucle *while* et permettent la transmission d'un message retour.

Le proxy serveur étant maintenant bidirectionnel, l'ordre de lancement des composants n'a plus d'importance.

INETMANET L'objectif initial était d'utiliser une information comme la puissance du signal reçu afin de maintenir en formation un groupe de drones. Cependant, le module *ChannelControl*, chargé des calculs concernant la couche physique, ne permet pas d'obtenir cette information. J'ai donc décidé d'utiliser une solution alternative.

Le nœud utilisé dans la simulation modélise le protocole ICMP et permet d'effectuer des requêtes ping. Le module concerné est le module PingApp et il permet d'obtenir l'information RTT (*Round-Trip delay Time*), c'est-à-dire le temps écoulé entre l'envoi d'une requête et à la réception de la réponse.

Afin de fournir le RTT au modèle de mobilité, j'ai utilisé le *Notification Board*, une fonctionnalité basée sur le modèle patron/observateurs. Les modules intéressés par une catégorie d'information s'abonnent auprès du *Notification Board* et ils reçoivent les mises à jour lorsqu'un autre module change l'état de cette information. La méthode processPingResponse() de la classe PingApp a été modifiée afin d'inclure le code suivant :

```
if (notificationBoard == NULL) {
   std::cout << "RTT: notificationBoard est null !" << endl;
} else {
   if (!notificationBoard->hasSubscribers(RTT_CHANGED)) {
      std::cout << "RTT: il n'y a pas d'abonne " << endl;
   } else {
      RttInfo *details = new RttInfo();
      details->setRtt(rtt.raw());

      cout << "RTT: poste une notification avec RTT = " << (rtt * 1000)
          << endl;
      notificationBoard->fireChangeNotification(RTT_CHANGED, details);
    }
}
```

Ce code indique au *Notification Board* qu'un changement a été effectuée dans la catégorie « RTT_CHANGED » et fournit la valeur du RTT.

Afin d'abonner le modèle de mobilité à cette information, la méthode initialize() de la classe CouplageMobility a été modifiée et les lignes de code suivantes ont été ajoutées :

```
if (notificationBoard == NULL) {
```

```
        std::cout << "drone " << droneID << ": notificationBoard est null
            !" << endl;
    } else {
        std::cout << "drone " << droneID << ": je m'abonne" << endl;
        notificationBoard->subscribe(this, RTT_CHANGED);
    }
```

Pour réagir aux mises à jour de la catégorie par le *Notification Board*, il faut aussi inclure dans le modèle de mobilité une méthode receiveChangeNotification() :

```
void CouplageMobility::receiveChangeNotification(int category, const
    cObject *details) {

  if (category == RTT_CHANGED) {

        RttInfo *info = (RttInfo *)(details);
        double rtt = (info->getRtt() / 1e6);
        std::cout <<"valeur du RTT : " << rtt << endl;

        rttint = static_cast<int>(rtt);

  } else {
        std::cout << "notification de categorie " << category << " recue
            " << endl;
  }
}
```

Cette méthode reçoit les mises à jour de la valeur du RTT. Cette valeur est convertie en microsecondes et elle est placée au format *integer* dans la variable « rttint ».

Enfin, une méthode sendSamMessage() a été développée afin d'envoyer la valeur du RTT par le *socket* TCP au proxy serveur :

```
void CouplageMobility::sendSamMessage() {
    if (droneID == 0) {
        std::string BufferEmission;

        std::ostringstream os_stream;
        os_stream << rttint;
        rttstr = os_stream.str();

        BufferEmission = rttstr + char(13);

        ::send(MYSOCKET, BufferEmission.c_str(), BufferEmission.length(), 0);

    }
}
```

Cette méthode convertit au format *string* la variable « rttint » et ajoute à la fin de la chaîne un caractère « retour chariot » — char(13) — afin que le proxy serveur détecte la fin du message reçu.

Ces modifications effectuées permettent d'établir une communication entre OMNet++ et le proxy serveur et fournissent une information représentant un délai de communication entre deux nœuds simulés.

SAMOVAR La dernière étape consiste à modifier le modèle de SAMOVAR pour que celui-ci reçoive la valeur du RTT et la traite afin d'exécuter un scénario.

La S-Function effectuant la connexion avec le proxy serveur a été modifiée afin de pouvoir recevoir des données et les lignes suivantes ont été ajoutées :

```
mssgRetour = fscanf(socketTCP, '%c');

rtt = str2double(mssgRetour);

block.OutputPort(1).Data = rtt;
```

Ces commandes permettent de recevoir des données au format « caractères ». Ces données sont ensuite converties au format « double » et passées à un connecteur de sortie ajouté au bloc correspondant à la S-Function.

Finalement un nouveau bloc associé à la S-Function ReceiveRTT a été ajouté. Ce bloc prend comme entrée la valeur du RTT venant du *socket* TCP et ses sorties permettent de contrôler le deuxième drone de la simulation.

Le modèle final permettant d'établir un couplage avec OMNet++ est illustré par la figure 6.21.

Le code de la S-Function ReceiveRTT est fondé sur la S-Function ReceiveUDP du drone contrôlé par l'utilisateur et il doit être adapté suivant le scénario envisagé.

La figure illustre l'exécution d'une simulation mettant en œuvre l'ensemble des composants réalisant un couplage bidirectionnel entre SAMOVAR et OMNet++.

Exemple de scénarios simples Un algorithme à inclure dans la S-Function ReceiveRTT permet des scénarios de type « Attraction/Répulsion ».

Dans le scénario attraction, les drones sont initialement situés à portée de réception l'un de l'autre. L'utilisateur ne peut contrôler le drone *leader* que dans une direction sur l'axe des abscisses. Lorsque les drones ne sont plus à portée de réception, la valeur du RTT devient nulle. Le drone *suiveur* reçoit cette information, il décolle et effectue un déplacement sur l'axe des abscisses tant que la valeur du RTT est nulle. Le code est alors le suivant :

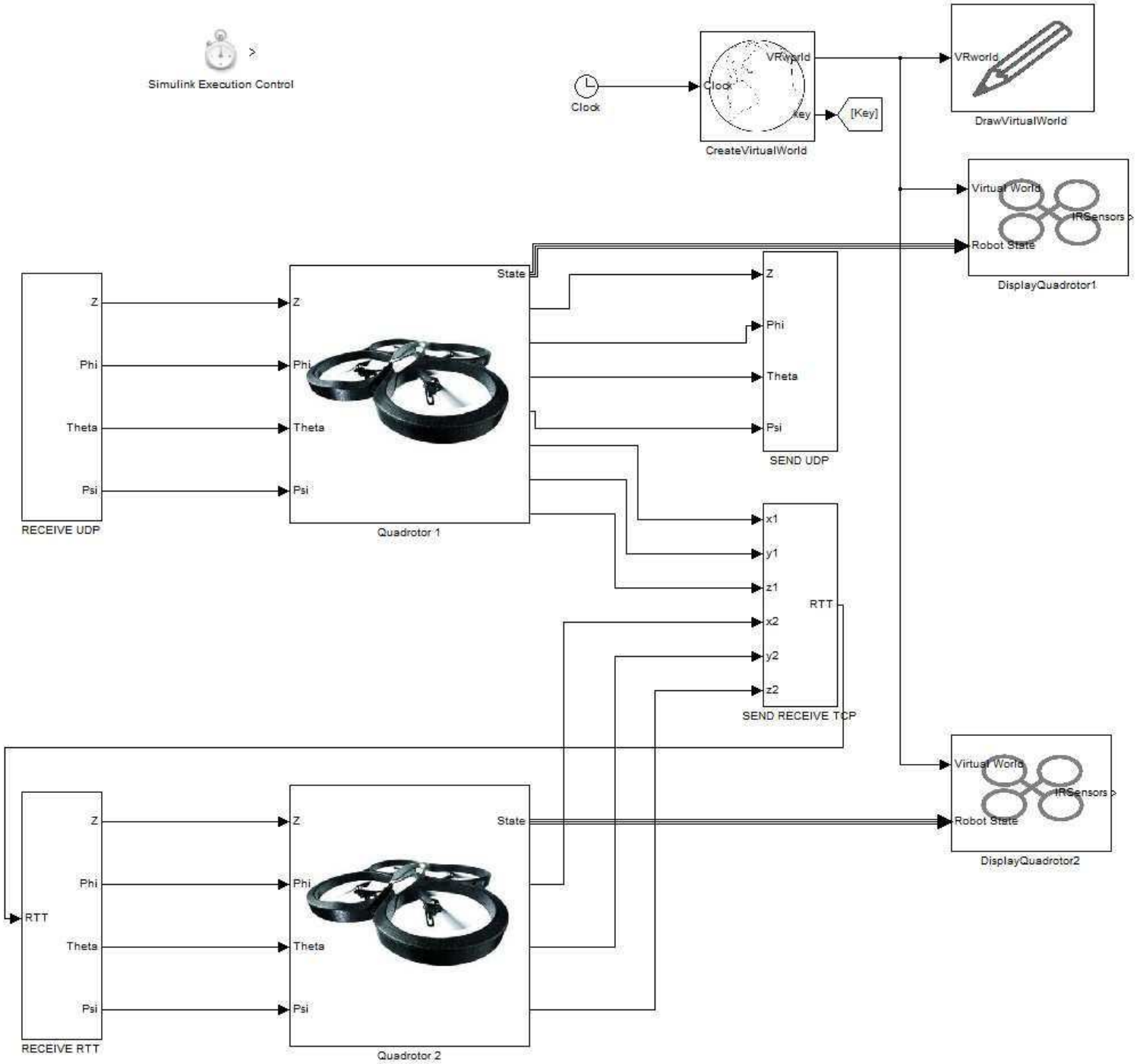


FIGURE 6.21 – Le modèle de SAMOVAR permettant un couplage bidirectionnel avec OMNet++

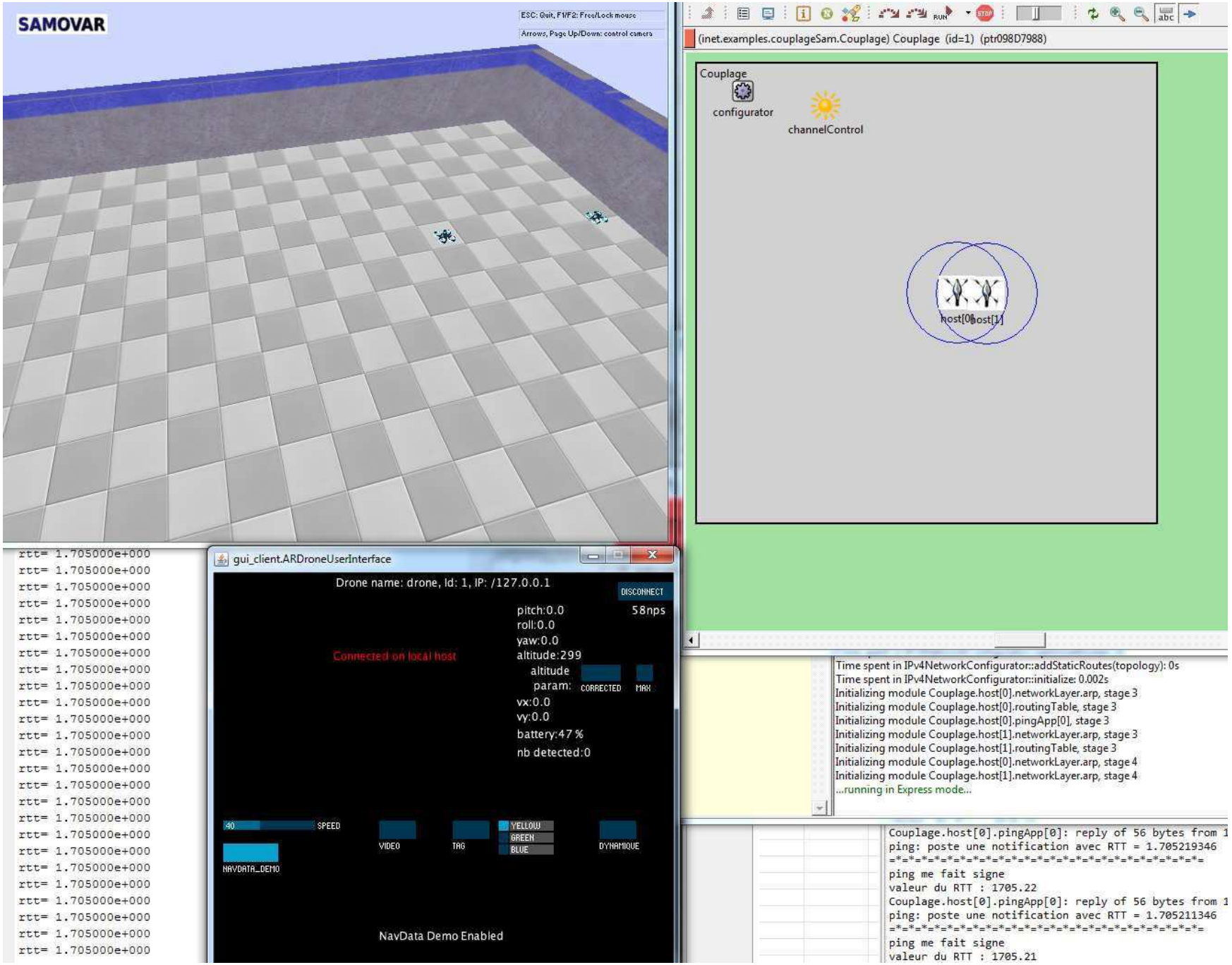


FIGURE 6.22 – L'exécution d'une simulation couplant SAMOVAR et OMNet++

```
if (rtt == 0)

        etat = 1; % le drone decolle
        theta = 0.2; % le drone se deplace

else
        theta = 0; % le drone se stabilise

end % fin if
```

Au contraire, le scénario répulsion place initialement les drones hors de portée de réception, et lorsque le drone *leader* se déplace et rentre dans la portée du drone *indépendant*, celui-ci s'éloigne jusqu'à ne plus être *détectable* :

```
if (rtt >= 0)

        etat = 1; % le drone decolle
        theta = 0.2; % le drone se deplace

else
        theta = 0; % le drone se stabilise

end % fin if
```

6.2.4 Limitations

La valeur du RTT fournie par OMNet++ varie peu en fonction de la position des drones, dès qu'ils sont dans la portée de réception l'un de l'autre. Cette information ne permet pas de déterminer la distance séparant les drones. Des scénarios complexes mettant en œuvre une *escadrille* de drones ne sont pas envisageables. Une solution serait de modifier le module *ChannelControl* afin que celui-ci calcule et fournisse la puissance reçue par chaque nœud.

Alors qu'OMNet++ autorise un changement dynamique de la configuration de la simulation, SAMOVAR ne permet pas de changer de manière simple le nombre de drones impliqués dans la simulation, et un nouveau modèle doit être développé pour chaque cas d'étude.

Le proxy serveur développé sert juste à établir la communication entre les deux simulateurs et il est transparent au niveau des informations échangées. Un *middleware* plus évolué servant de réservoir pour des événements ou des notifications permettrait un couplage plus lâche et une modularité accrue.

Chapitre 7

Utilisation et évaluation du démonstrateur

7.1 Plates-formes de test

Les moyens suivants ont été utilisés pour mener les tests des différents composants du démonstrateur :

- AR.Drones première version équipés des *firmwares* 1.4.7, 1.7.4 et 1.7.6 ;
- ordinateur de type PC équipé d'un processeur Intel Core 2 Quad Q9400 à 2,6 GHz (quadcore sans hyperthreading), 4 Gb de RAM DDR2 à 800 MHz et d'un chipset graphique intégré ;
- ordinateur de type PC équipé d'un processeur Intel Core i7-2600k à 3,4 GHz (quadcore avec hyperthreading), 8 Gb de RAM DDR3 1600 MHz et d'une carte graphique Radeon HD 6970 ;
- ordinateur portable de type PC équipé d'un processeur Intel Core i5-2430M (dualcore avec hyperthreading), 8 Gb de RAM DDR3 à 1333 MHz et d'une carte graphique Nvidia GeForce GT540M ;
- système d'exploitation Ubuntu 11.10 32 bits ;
- système d'exploitation Windows 7 Professionnel 64 bits ;
- Java SE version 7 et OpenJDK 7 ;
- réseaux de type IP *ethernet* filaire et/ou Wi-Fi.

7.2 Tests réalisés

Afin de valider tous les composants du démonstrateur, différents tests ont été menés tout au long de la conception de celui-ci.

Interface de contrôle seule Les tests suivants ont été effectués avec JaSMinDrone sur un ou plusieurs AR.Drone, avec les différentes versions de *firmware* :

- décollage et atterrissage. Résultat correct ;
- vol stationnaire. Résultat correct ;
- déplacements du drone sur tous les axes d'Euler et en hauteur (montée, descente). Résultat correct ;
- variations de la vitesse de déplacement en fonction de la position du *slider* (position min, max, intermédiaires). Résultat correct ;

- essai de toutes les manœuvres précédentes en intérieur et en extérieur. Une légère dérive est constatée en extérieur due au vent, un essai avec l'application Parrot sur un smartphone Android a montré les mêmes résultats. Résultat correct ;
- essai avec la carène d'intérieur et d'extérieur. Le drone est légèrement plus maniable avec la carène d'extérieur. Résultat correct ;
- réception de la vidéo dans tous les modes (caméra frontale, verticale, *PiP*). Résultat correct ;
- réception des NavData, en mode complet ou en mode NavDataDemo, affichage des paramètres de navigation et des différents capteurs du drone, affichage de l'état de la batterie et du mode *low battery*. Résultat correct ;
- réception et affichage du masque d'état du drone. Résultat correct (il demeure un flou sur la signification de quatre variables, voir annexe 10) ;
- envoi de commandes évoluées (activation du mode *emergency*, configuration de l'altitude maximale, changement de la couleur du *tag*, etc.). Résultat correct ;
- réalisation du scénario « Suivi de tag », en intérieur et en extérieur, avec les trois couleurs disponibles. La détection du *tag* est fortement dépendante des conditions de luminosité, un même *tag* est ou n'est pas détecté suivant l'éclairage utilisé, l'heure de la journée, la position par rapport à la source lumineuse, etc. Le traitement vidéo de détection du *tag* est effectué par le drone lui-même, aucune solution afin d'améliorer ce problème n'a pu être apportée. Résultat satisfaisant ;
- réalisation du scénario « Suivi dynamique », en intérieur et en extérieur, avec le drone *leader* contrôlé à distance ou tenu en main à la manière d'une télécommande. Résultat correct ;
- réalisation de vol programmé grâce à la classe Autopilot. Résultat correct ;
- essai sur les trois PC décrits à la section 7.1. Un *bug* intermittent de l'affichage survient de manière plus fréquente sur le système d'exploitation Ubuntu.

L'ensemble des tests effectués permet de conclure à un fonctionnement correct de la solution JaSMinDrone.

Couplage de JaSMinDrone et de SAMOVAR Les tests suivants ont été menés sur la solution de couplage JaSMinDrone/SAMOVAR , avec un ou plusieurs drones simulés :

- réalisation des mêmes mouvements effectués avec JaSMinDrone seule. Résultat correct ;
- affichage dans JaSMinDrone de l'état du drone simulé, en vol stationnaire ou en mouvement. Résultat correct ;
- réalisation du scénario « Suivi dynamique », avec comme *leader* le drone réel et comme suiveur le drone simulé, puis en inversant les rôles. Résultat correct ;
- vol en formation de drones simulés, en contrôlant le *leader* par les touches du clavier, puis par la réalisation d'un scénario « Suivi dynamique », avec inversion des rôles comme précédemment. Résultat correct ;
- essai sur les trois PC décrits à la section 7.1. Avec Ubuntu, la version du couplage utilisant les S-Function souffre d'une latence entre l'envoi d'une commande et la réalisation de celle-ci. Cette latence n'existe pas pour la même configuration matérielle sous Windows. Résultat correct sous Windows, à perfectionner sous Ubuntu ;

– essai en exécution locale ou répartie, à travers un réseau filaire ou Wi-Fi, en réalisant des mouvements simples ou les scénarios précédents. Résultat correct.

Ces différents essais montrent un fonctionnement correct du couplage en lui-même entre JaSMinDrone et SAMOVAR, bien qu'il existe une latence à la réalisation des commandes, lorsque SAMOVAR est exécuté sous Ubuntu (en local ou de manière répartie).

Couplage SAMOVAR/OMNeT++ Les essais suivants ont été effectués sur la solution de couplage entre SAMOVAR et OMNeT++ :

- réalisation des différents mouvements du drone et du nœud associé depuis l'interface de contrôle. Résultat correct.
- essai avec un drone simulé et un nœud, puis avec deux drones et deux nœuds. Résultat correct.
- vérification de la transmission de la valeur du RTT à SAMOVAR, en approchant et en éloignant les nœuds. Résultat correct.
- réalisation du scénario simple "répulsion". Résultat correct.
- essai en exécution locale ou répartie des composants, à travers un réseau filaire ou Wi-Fi, en réalisant des mouvements simples ou le scénario précédent. Résultat correct.

Malgré la latence relevée précédemment lors de l'exécution de SAMOVAR sous Ubuntu, ces tests ont permis de vérifier le bon fonctionnement de la solution de couplage finale mettant en œuvre JaSMinDrone, SAMOVAR etOMNeT++.

7.3 Facilité d'utilisation

Tous les tests menés ont montré une fluidité de la solution, quel que soit le PC de la plate-forme de test utilisé, en exécution locale des composants ou bien en répartie.

Il est à noter que l'exécution du démonstrateur est rythmée par les échanges entre SAMOVAR et OMNeT++, la fluidité de la solution dépend donc du choix effectué quant à l'exécution de la simulation dans OMNeT++ : en mode avec animation, en mode rapide ou express. Le mode animation permet de se rendre compte visuellement des échanges effectués entre les nœuds, en contrepartie la simulation est bloquée pendant l'affichage de l'animation. Il est recommandé l'usage du mode express afin de se rapprocher d'un déroulement « temps réel ».

L'utilisation de ce démonstrateur demande cependant une bonne connaissance des composants, notamment de OMNeT++ et de Matlab. En effet, le lancement des composants s'effectue un par un et des réglages avant l'utilisation sont parfois nécessaires suivant la machine utilisée ou le type d'exécution (locale ou répartie). Il est néanmoins envisageable de développer un script permettant d'automatiser le lancement des composants.

L'interface de contrôle JaSMinDrone est utilisée actuellement au LORIA dans le cadre du développement de la plate-forme AETOURNOS. Les utilisateurs n'ont remonté aucun problème de fonctionnement ni de difficultés d'utilisation.

7.4 Tests futurs

Il est prévu d'effectuer au cours du mois de septembre les tests suivants :

- essai avec la deuxième version de l'AR.Drone de l'interface de contrôle d'abord puis de l'ensemble des composants ;
- essai avec un plus grand nombre de drones simulés et de nœuds correspondants ;
- essai de la facilité de changement du type de nœud et des protocoles de communication (bluetooth, ZigBee, etc.) ;
- calibration des simulateurs (déplacements du drone dans SAMOVAR et portée des nœuds dans OMNeT++).

Conclusion et perspectives

Bilan du travail réalisé

Rappel des objectifs du projet

Le travail réalisé entre dans le cadre d'un projet visant à étudier et relever les défis de la multisimulation. Un cas d'étude était de développer une plate-forme de simulation permettant d'étudier les influences mutuelles entre les choix réalisés au niveau d'un réseau de communication et le comportement des éléments communicants au travers de ce réseau. La finalité était d'obtenir une plate-forme permettant de tester différents algorithmes de vol en formation pour des drones, en se basant sur des informations de bas niveau (couches 1 à 3 du modèle OSI) et en tenant compte aussi bien des caractéristiques mécaniques des drones que des paramètres des protocoles de communication employés. Le travail réalisé se divise en deux volets principaux : le développement d'une interface de contrôle d'AR.Drones d'une part, et l'association de cette interface avec deux simulateurs s'exécutant en simultané d'autre part.

L'interface de contrôle d'AR.Drone

L'interface JaSMinDrone développée permet de piloter un quadricoptère grand public, l'AR.Drone de Parrot, à partir d'un PC équipé de Wi-Fi et de visualiser en temps réel sur le PC l'état du drone ainsi que les images captées par ses caméras. Le pilotage peut être interactif, les ordres de déplacement étant fournis au clavier par l'utilisateur, ou bien programmé, les ordres de déplacements étant fournis par une application Java écrite par l'utilisateur. Plusieurs drones peuvent être contrôlés à partir du même PC, ce qui permet entre autres de réaliser des scénarios de vol où un drone est asservi à l'état d'un autre, ou bien où un drone en suit un autre équipé d'un *tag* coloré particulier, ces 2 scénarios étant inclus dans l'interface.

Cette interface Java constitue une API de plus haut niveau que celle offerte par le SDK de Parrot et, contrairement au code ARDRoneMines qui a servi de base à ces développements, sa documentation permet une prise en main rapide. Le code va être rendu public et il est d'ores et déjà prévu qu'il soit utilisé dans des projets de SUPELEC et de l'École des Mines de Nancy.

Les principales difficultés rencontrées lors du développement et de la mise au point de cette interface ont été :

- le manque de documentation concernant l'interface de contrôle initiale ;
- l'impossibilité de tester cette interface de contrôle initiale avec la version du *firmware* installée sur les drones de SUPELEC ;
- l'absence de connexion logicielle directe au drone, rendant obligatoire l'utilisation d'un logiciel de capture du trafic réseau afin d'analyser les dysfonctionnements et de combler les vides de la documentation ;
- l'absence initiale d'indication de passage en modes *emergency* ou *low battery* dans lesquels le drone ne répond plus aux commandes et me faisant douter des modifications apportées au code ;
- les problèmes intermittents d'affichage de l'interface graphique.

Association de l'interface à deux simulateurs

Concernant la plate-forme de multisimulation, le premier travail a consisté à identifier les logiciels permettant déjà de simuler certains aspects d'un scénario de vol en formation : il fallait un simulateur de réseaux, ainsi qu'un simulateur de drones. En parallèle j'ai effectué un inventaire des solutions existantes permettant de faire collaborer des simulations. La conclusion de cette phase du travail est que l'utilisation du standard HLA demande une grande expertise pour être mise en place et maintenue. De plus, si HLA est bien adapté aux multisimulations consistant à faire figurer sur un terrain commun des objets issus de simulateurs distincts, il se prête moins à nos besoins consistant à simuler différents aspects d'un même élément — un drone par exemple — par des simulateurs différents. Une alternative à HLA, et qui est déjà utilisée avec succès dans l'industrie automobile, est FMI, un standard qui malheureusement évolue encore rapidement. Afin d'obtenir une plate-forme fonctionnelle dans les délais impartis et avec les ressources disponibles, HLA et FMI ont été écartés. C'est donc un couplage ad hoc qui a été réalisé entre OMNeT++, SAMOVAR et JaSMinDrone. Néanmoins la compatibilité avec HLA est restée dans nos critères de choix, et la recherche bibliographique effectuée montre que les simulateurs sélectionnés, OMNeT++ et SAMOVAR basé sur Matlab, ont déjà été couplés avec succès avec une telle architecture, Matlab/Simulink étant de plus compatible FMI.

Le modèle d'AR.Drone de SAMOVAR a été modifié afin qu'il reçoive par le réseau les ordres de déplacement émanant de JaSMinDrone, exactement comme ces ordres sont envoyés à un drone réel. De la même manière, les informations sur l'état du drone sont envoyées à JaSMinDrone qui les affiche absolument comme si elles provenaient d'un drone réel. Il est alors possible de combiner les fonctionnalités et d'asservir les déplacements d'un drone simulé aux mouvements effectués par un drone réel, ou réciproquement.

Pour réaliser un vol en formation, les drones doivent avoir « conscience » de la position de leurs voisins. Pour le scénario nous intéressant, il ne s'agit pas d'avoir une information visuelle sur leurs voisins, mais plutôt qu'ils estiment la distance les séparant les uns des autres en se basant sur la puissance des signaux radio reçus. OMNeT++ ne permettant pas de récupérer aisément cette information, en première approximation les drones s'échangent des pings et c'est le RTT, le délai entre l'envoi de la requête et la réception de la réponse, qui sert d'information de proximité. Cette information doit ensuite être transmise à SAMOVAR où l'algorithme de déplacement des drones simulés est implémenté. De la même manière, SAMOVAR va prendre des décisions de déplacement basées sur les informations de proximité qu'il a reçues puis il devra informer OMNeT++ de la nouvelle position des drones, de manière à ce qu'OMNeT++ puisse réévaluer le RTT, et ainsi de suite. Pour éviter un couplage trop fort des deux simulateurs, un élément intermédiaire, un proxy serveur, a été développé. SAMOVAR a été modifié pour envoyer à ce proxy la position de deux drones simulés et attendre en retour la valeur du RTT calculé par OMNeT++. De même, un nouveau modèle de mobilité a été introduit dans OMNeT++ de manière à asservir les déplacements et à envoyer la valeur du RTT entre les nœuds représentant les drones.

Le résultat de ce couplage est illustré par un scénario de type « répulsion » où l'utilisateur pilote interactivement un drone simulé. Lorsque ce drone entre à portée

d'un autre, ce dernier décolle et part de manière à ne plus être à portée du drone piloté. À l'inverse, dans un scénario nommé « attraction », le deuxième drone tente de rester à portée du drone piloté. Il suffira maintenant d'ajouter des drones dans la simulation pour étudier des algorithmes de vol cohérent plus évolués. L'influence de différents paramètres réseaux sur le maintien de la formation pourra aussi être étudiée facilement en changeant les protocoles dans OMNeT++.

La première difficulté rencontrée dans la réalisation de ce couplage a été la connaissance minimale nécessaire des simulateurs impliqués ; or ils sont très riches et plutôt destinés à des experts de chacun des domaines. De plus, savoir les utiliser de manière individuelle n'était pas suffisant, il a ensuite fallu identifier les possibles points d'échange d'information et réaliser les « accroches » nécessaires.

Problèmes non résolus

Malgré les recherches menées et les différents essais effectués, il ne m'a pas été possible de résoudre le problème du *bug* intermittent d'affichage de JaSMinDrone. Je pense que le problème provient de l'utilisation des bibliothèques Napplet et/ou ControlP5. Ces bibliothèques, développées par des particuliers, sont en version bêta et non finalisées.

Il n'a pas été possible d'utiliser dans OMNeT++, comme cela était initialement prévu, la puissance du signal reçu afin de pouvoir déterminer la distance et la position relative entre plusieurs nœuds. Une modification du module ChannelControl est nécessaire. Cependant les calculs et les changements à effectuer dans le code sont complexes et demandent beaucoup de temps.

Le problème de la latence à l'exécution d'une commande sous Ubuntu lors de l'utilisation des S-Function n'a pu être résolu. Le modèle de couplage avant l'utilisation de celles-ci ne présentait pas ce problème et je n'ai pas réussi à déterminer l'origine du problème.

Perspectives

De nombreuses pistes de poursuite de ces travaux sont envisagées. On peut y distinguer deux objectifs différents :

- accroitre l'expertise et « l'outillage » dans le domaine de la multisimulation et l'appliquer à une autre problématique industrielle comme la simulation des systèmes complexes que sont les *SmartGrids* par exemple ;
- utiliser la plate-forme pour mettre au point des algorithmes de vol en formation adaptés à certaines configurations réseaux ou faire évoluer des protocoles réseau existants de manière à faciliter le vol en formation.

Malgré l'utilisation d'un proxy pour éviter les communications directes de simulateur à simulateur, le couplage reste relativement fort. La plate-forme gagnerait en flexibilité si ce couplage était plus lâche. Un simulateur aurait alors la possibilité de déclarer les informations qu'il est capable d'exporter, puis de poster ces valeurs si un autre simulateur se montre intéressé. Cela faciliterait le changement, peut-être même pendant le déroulement de la simulation, d'un simulateur par un autre exposant les mêmes informations.

OMNeT++ et SAMOVAR ont dû être modifiés. En multipliant les utilisations de la plate-forme et les scénarios, on pourrait identifier des modifications récurrentes et proposer une architecture plus générique limitant ce genre de développements.

Comme cela a été souligné dans la section 7.3, l'expérience d'utilisation de la plate-forme pourrait être améliorée si un script se chargeait de lancer les différents logiciels nécessaires sur les différentes machines, de réaliser les configurations et de démarrer chaque exécution. Disposer d'une interface graphique pour assembler et configurer les différents éléments de la simulation et conduisant à l'écriture automatique de ce script serait remarquable.

Pour toutes ces évolutions, il faut étudier l'opportunité d'utiliser FMI.

La plate-forme actuelle est déjà utilisée dans le projet AETOURNOS pour étudier des comportements de déplacement en formation. Cependant de nombreux codes implémentant des algorithmes de *flocking* sont disponibles pour des plates-formes suivant une approche *agents*, et devraient être entièrement portés pour être utilisés dans le démonstrateur actuel. Il faut donc intégrer une de ces plates-formes, comme NetLogo, au démonstrateur. Cela constitue un des objectifs à moyen terme du projet AETOURNOS.

La multisimulation peut être utilisée pour étudier des solutions d'aménagement et d'organisations urbaines. Avoir une interface de sortie conviviale peut réellement contribuer à présenter les alternatives d'aménagement aux élus, aux citoyens, etc. Améliorer le rendu 3D du démonstrateur est dans ce cas nécessaire, une solution étant peut-être le couplage avec FlightGear. Un projet est en cours à l'École Nationale Supérieure des Mines de Nancy.

Quatrième partie

Annexes

Chapitre 8

Les informations ajoutées à l'interface utilisateur

FIGURE 8.1 – Les interfaces utilisateur d'ARDroneMines (à gauche) et de JaSMin-Drone (à droite)

FIGURE 8.2 – Information de mode *emergency*

FIGURE 8.3 – Information de batterie faible

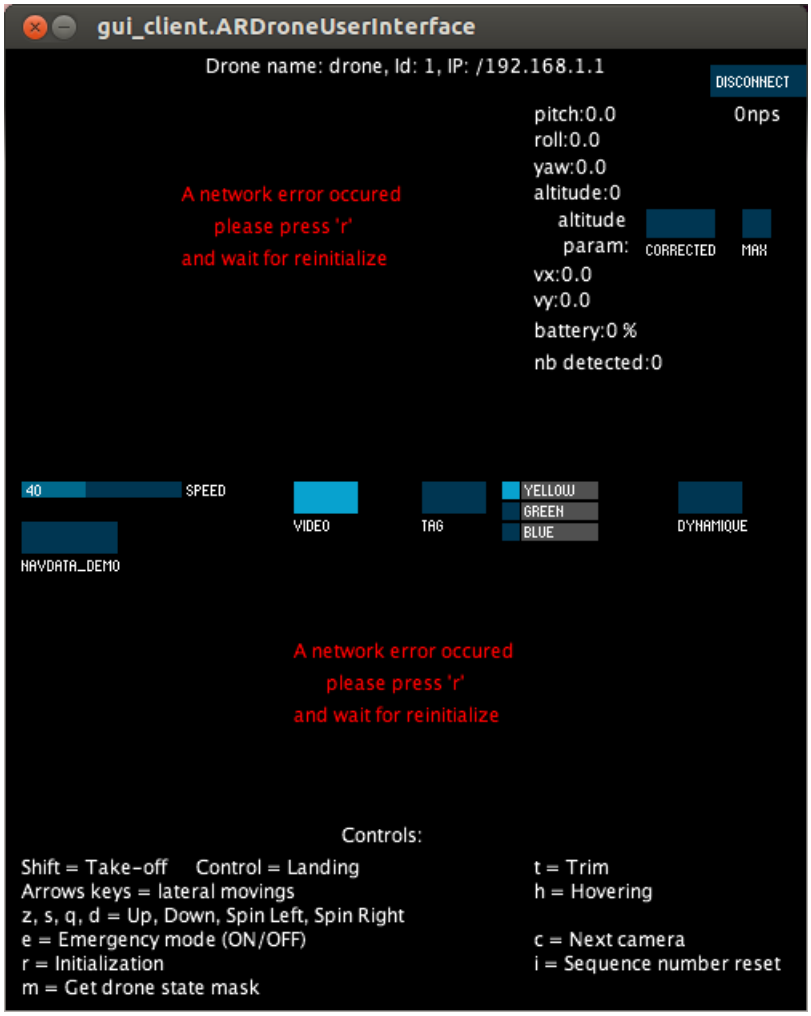


FIGURE 8.4 – L'affichage lors d'un problème réseau

Chapitre 9

La classe Autopilot

```
package non_interactive_client;

import java.io.IOException;
import lib.ardrone.ARDroneEntity;
import lib.ardrone.control.Control;

/**
 *
 * @author Yannick Presse
 */
public class AutoPilot {

    /**
     *
     * @param args
     */
    public static void main(String[] args) {

        ARDroneEntity myDrone = new ARDroneEntity();
        boolean stopScenario;
        boolean lowBattery;
        myDrone.setSpeed(40);
        myDrone.setName("drone");
        myDrone.setInetAdress("192.168.1.1");
        myDrone.setId(1);
        myDrone.connect();
        Control myController = myDrone.getController();


        try {
            Thread.sleep(1000);
        } catch (Exception e) {
        }
```

```
        stopScenario = myDrone.testEmergency();
         if (!stopScenario) {

            lowBattery = myDrone.testBattery();

            if (!lowBattery) {

                  /*
                   * Scenario begins here
                   *
                   * Use the method with time arguments in millisec
                   * An optionnal second argument is the movement speed in % (
                       default value is 40%)
                   *
                   * The combined move needs 4 arguments: time in millisec, speed
                       in % [forward(-) or backward(+)],
                   * speed in % [down(-) or up (+)], speed in % [spin left(-) or
                       right(+)]
                   * In the following example the drone will go forward at 40%, go
                       up at 30% and spin right at 50% during 2 seconds
                   * myDrone.combinedMoves(2000, -40, 30, 50);
                   *
                   */

                  myDrone.trim();
                  myDrone.takeoff();
                  myDrone.hover(1500);
/* myDrone.goForward(800, 30);
                  myDrone.goBackward(800, 30);
                  myDrone.hover(1000);
                  myDrone.goForward(1500, 10);
                  myDrone.goBackward(2000, 10);
                  myDrone.hover(1000);
                  myDrone.goLeft(600);
                  myDrone.goRight(800);
                  myDrone.up(800);
                  myDrone.down(600);
                  myDrone.spinLeft(1000);
                  myDrone.spinRight(1200);
                  myDrone.combinedMoves(2000, -40, 30, 50);
                  myDrone.hover(1500);
*/ myDrone.landing();


                  /*
                   * The scenario ends here
                   */

             } else {
```

```
                for (int i = 0; i < 5; i++) {
                    System.out.println("\n LOW BATTERY");
                }
            }
        }
        myDrone.Disconnect();

    }
}
```

Chapitre 10

Le masque d'état du drone

L'extrait suivant du fichier header config.h du SDK présente la signification des bits du masque d'état du drone. Cependant des contradictions existent entre le commentaire et l'énumération quand à la signification des bits 7 (Trim command ACK ou Firmwre file is good), 8 (Trim running ou Firmware update is newer), 9 (Trim result ou Firmware update is ongoing) et 16 (Vbat high ou User emergency landing).

```
/// \enum def_ardrone_state_mask_t is a bit field representing ARDrone' state


// Define masks for ARDrone state
// 31                                                              0
//  x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x -> state
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | FLY MASK : (0) ardrone is landed, (1) ardrone is flying
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | VIDEO MASK : (0) video disable, (1) video enable
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | VISION MASK : (0) vision disable, (1) vision enable
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | CONTROL ALGO : (0) euler angles control, (1) angular speed control
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | ALTITUDE CONTROL ALGO : (0) alt control inactive (1) alt control active
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | USER feedback : Start button state
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Control command ACK : (0) None, (1) one received
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Trim command ACK : (0) None, (1) one received
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Trim running : (0) none, (1) running
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Trim result : (0) failed, (1) succeeded
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Navdata demo : (0) All navdata, (1) only navdata demo
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Navdata bootstrap : (0) options sent in all or demo mode, (1) no navdata options sent
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Motors status : (0) Ok, (1) Motors Com is down
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Bit means that there's an hardware problem with gyrometers
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | VBat low : (1) too low, (0) Ok
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | VBat high (US mad) : (1) too high, (0) Ok
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Timer elapsed : (1) elapsed, (0) not elapsed
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Power : (0) Ok, (1) not enough to fly
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Angles : (0) Ok, (1) out of range
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Wind : (0) Ok, (1) too much to fly
//  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Ultrasonic sensor : (0) Ok, (1) deaf
```

```
//  | | | | | | | | | Cutout system detection : (0) Not detected, (1) detected
//  | | | | | | | | | PIC Version number OK : (0) a bad version number, (1) version number is OK
//  | | | | | | | | ATCodec thread ON : (0) thread OFF (1) thread ON
//  | | | | | | | Navdata thread ON : (0) thread OFF (1) thread ON
//  | | | | | | Video thread ON : (0) thread OFF (1) thread ON
//  | | | | | Acquisition thread ON : (0) thread OFF (1) thread ON
//  | | | | CTRL watchdog : (1) delay in control execution (> 5ms), (0) control is well scheduled // Check frequency of control loop
//  | | | ADC Watchdog : (1) delay in uart2 dsr (> 5ms), (0) uart2 is good // Check frequency of uart2 dsr (com with adc)
//  | | Communication Watchdog : (1) com problem, (0) Com is ok // Check if we have an active connection with a client
//  | Emergency landing : (0) no emergency, (1) emergency

typedef enum {
  ARDRONE_FLY_MASK            = 1 << 0,  /*!< FLY MASK : (0) ardrone is landed, (1) ardrone is flying */
  ARDRONE_VIDEO_MASK          = 1 << 1,  /*!< VIDEO MASK : (0) video disable, (1) video enable */
  ARDRONE_VISION_MASK         = 1 << 2,  /*!< VISION MASK : (0) vision disable, (1) vision enable */
  ARDRONE_CONTROL_MASK        = 1 << 3,  /*!< CONTROL ALGO : (0) euler angles control, (1) angular speed control */
  ARDRONE_ALTITUDE_MASK       = 1 << 4,  /*!< ALTITUDE CONTROL ALGO : (0) altitude control inactive (1) altitude control active */
  ARDRONE_USER_FEEDBACK_START = 1 << 5,  /*!< USER feedback : Start button state */
  ARDRONE_COMMAND_MASK        = 1 << 6,  /*!< Control command ACK : (0) None, (1) one received */
  ARDRONE_FW_FILE_MASK        = 1 << 7,  /* Firmware file is good (1) */
  ARDRONE_FW_VER_MASK         = 1 << 8,  /* Firmware update is newer (1) */
//  ARDRONE_FW_UPD_MASK         = 1 << 9,  /* Firmware update is ongoing (1) */
  ARDRONE_NAVDATA_DEMO_MASK   = 1 << 10, /*!< Navdata demo : (0) All navdata, (1) only navdata demo */
  ARDRONE_NAVDATA_BOOTSTRAP   = 1 << 11, /*!< Navdata bootstrap : (0) options sent in all or demo mode, (1) no navdata options sent */
  ARDRONE_MOTORS_MASK         = 1 << 12, /*!< Motors status : (0) Ok, (1) Motors problem */
  ARDRONE_COM_LOST_MASK       = 1 << 13, /*!< Communication Lost : (1) com problem, (0) Com is ok */
  ARDRONE_VBAT_LOW            = 1 << 15, /*!< VBat low : (1) too low, (0) Ok */
  ARDRONE_USER_EL             = 1 << 16, /*!< User Emergency Landing : (1) User EL is ON, (0) User EL is OFF*/
  ARDRONE_TIMER_ELAPSED       = 1 << 17, /*!< Timer elapsed : (1) elapsed, (0) not elapsed */
  ARDRONE_ANGLES_OUT_OF_RANGE = 1 << 19, /*!< Angles : (0) Ok, (1) out of range */
  ARDRONE_ULTRASOUND_MASK     = 1 << 21, /*!< Ultrasonic sensor : (0) Ok, (1) deaf */
  ARDRONE_CUTOUT_MASK         = 1 << 22, /*!< Cutout system detection : (0) Not detected, (1) detected */
  ARDRONE_PIC_VERSION_MASK    = 1 << 23, /*!< PIC Version number OK : (0) a bad version number, (1) version number is OK */
```

```
  ARDRONE_ATCODEC_THREAD_ON   = 1 << 24, /*!< ATCodec thread ON : (0) thread OFF (1) thread ON */
  ARDRONE_NAVDATA_THREAD_ON   = 1 << 25, /*!< Navdata thread ON : (0) thread OFF (1) thread ON */
  ARDRONE_VIDEO_THREAD_ON     = 1 << 26, /*!< Video thread ON : (0) thread OFF (1) thread ON */
  ARDRONE_ACQ_THREAD_ON       = 1 << 27, /*!< Acquisition thread ON : (0) thread OFF (1) thread ON */
  ARDRONE_CTRL_WATCHDOG_MASK  = 1 << 28, /*!< CTRL watchdog : (1) delay in control execution (> 5ms), (0) control is well scheduled */
  ARDRONE_ADC_WATCHDOG_MASK   = 1 << 29, /*!< ADC Watchdog : (1) delay in uart2 dsr (> 5ms), (0) uart2 is good */
  ARDRONE_COM_WATCHDOG_MASK   = 1 << 30, /*!< Communication Watchdog : (1) com problem, (0) Com is ok */
  ARDRONE_EMERGENCY_MASK      = 1 << 31  /*!< Emergency landing : (0) no emergency, (1) emergency */
} def_ardrone_state_mask_t;
```

Chapitre 11

Documentation utilisateur

11.1 About

The software was originally developed as part of a student project by Paulin Andurand, Jeremie Hoelter, Mathieu Plachot and Baptiste Tissot under the direction of Laurent Ciarletta at Ecole Nationale Supérieure des Mines de Nancy in 2011. A couple of videos are available on youtube :
http ://www.youtube.com/user/ARDroneMines

This version was developed by Yannick Presse as part of EiCNAM Engineer Diploma Thesis at Supelec and was supervised by Virginie Galtier.

This software is released under a GPL version license.

11.2 Technical requirements

Obviously you need a Parrot AR.Drone. The current software version only supports AR.Drone version 1. A description of the meanings of the LED colors (emergency mode, landing, flying, etc.) and how to connect to the AR.Drone wifi network is provided in the user manual and is available on the website of Parrot :
http ://ardrone.parrot.com/parrot-ar-drone/usa/support

This software was developed in Java, a Java Virtual Machine(JVM) is mandatory (at least JVM 1.6).

The core functionalities of the software are located in the file ardronelib.jar.

The graphic user interface (GUI, see 11.3.2) is provided by the ARDroneDemo.jar package. As the GUI relies on Processing, the following packages (located in the lib directory) are required :

- controlP5.jar
- core.jar
- napplet.jar

Depending on your OS, you can either launch the GUI by double-clicking on the ARDroneDemo.jar or type the following command in a terminal from the directory where it is located : java -jar ARDroneDemo.jar

This software works with AR.Drone firmware above version 1.3.3 (disconnection occured after several seconds in this firmware, this issue may have been fixed but I haven't any drone left with this version to test it) and it has been tested with firmware version until 1.7.11.

A wifi interface on your computer is mandatory.

11.3 Interactive interface

11.3.1 Connection

The first window (fig. 11.1) contains a single button "NEWDRONE". By clicking this button, a new entity "drone" will be created. As it is possible to operate several drones in the same time, each drone is considered as a separate entity.

FIGURE 11.1 – Initial window

Then, the new drone entity needs to be parametrized (fig. 11.2). The parameters are a name (characters string), an ID (integer number) and an IP address. Some features such as geolocation, accelerometer or magnetic compass may be provided by an android phone attached to the drone. So the toggle button PHONE_ATTACHED will activate this features.
A dedicated application must be installed on the phone. This application is available on SourceForge, but it hasn't been tested up to now.

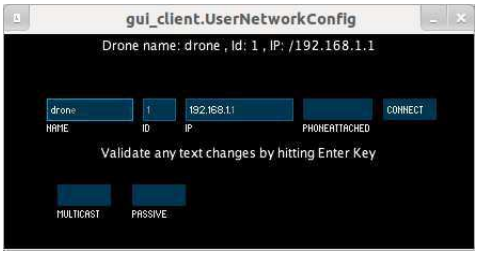


FIGURE 11.2 – drone Parameters

The ID is useful when you intend to realize a multi-drones scenario (see 11.3.4). In this case, each ID has to be different to the others. The default IP address of the AR.Drone is 192.168.1.1, there is no need to change it for a normal use of a single drone.

The MULTICAST and PASSIVE buttons are useful for multicasting purpose (see 11.3.3).

The connection is initiated by pressing CONNECT (the wifi connection to the AR.Drone network is supposed to be already done).

11.3.2 Graphic User interface

The user interface window (fig. 11.3) displays video flow and navigation data (navdata) information. This window also allows the user to control the drone. The drone can send navdata information either in full mode or in demo mode (fig. 11.4). When in navdata demo mode, the drone will not send information about motor,

gyroscopes, etc. The bandwidth will be saved in this mode. The NAVDATA_DEMO button allows to select the desired mode.

FIGURE 11.3 – User interface

The video flow can be turned on or off by the toggle button VIDEO. The drone carries two cameras, a horizontal one (fig. 11.4) and a lower resolution vertical one (fig. 11.5). The user can switch the video flow by pressing the "c" key. The following modes are proposed : horizontal or vertical cam only, picture-in-picture mode (fig. 11.6) either vertical in horizontal or horizontal in vertical.

In order to control the drone, you must first click anywhere inside the user interface window to enable the keyboard listener. Then, the following keys are used to control the drone :

- shift : take off
- control : landing
- arrows key : forward, backward, go left, go right
- z, s, q, d : up, down, spin left, spin right
- h : hovering
- t : trim (resetting the value of the inertial unit to zero, the drone must be settled on a flat surface)
- c : next camera
- e : emergency mode (on/off)
- r : perform a connection initialization sequence (for development purpose only)
- i : reset the sequence number of the ATcommand (for development purpose only)
- m : get the drone state mask (for development purpose only)

The slider SPEED adjusts the speed of the drone between 0% and 100% of its capacity. The default value is 40%.

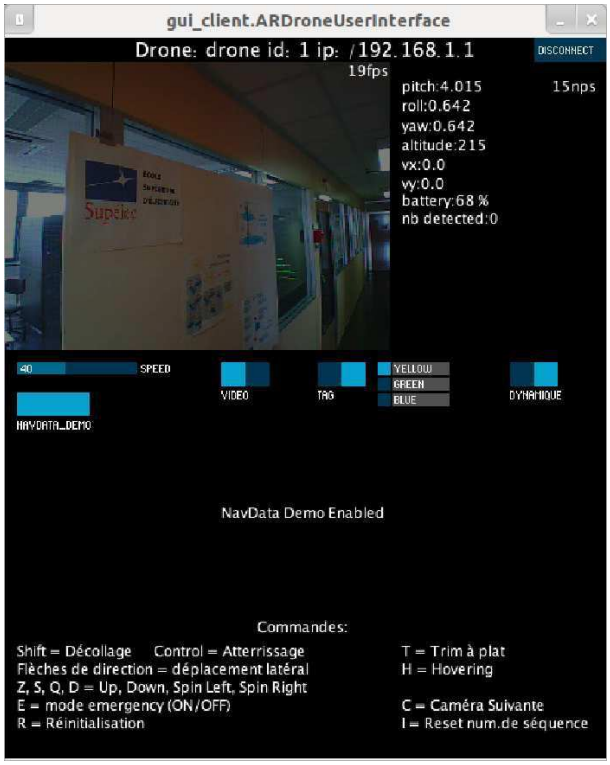


FIGURE 11.4 – Navdata demo enabled

FIGURE 11.5 – Vertical cam

You can disconnect the drone by clicking the DISCONNECT button, the drone

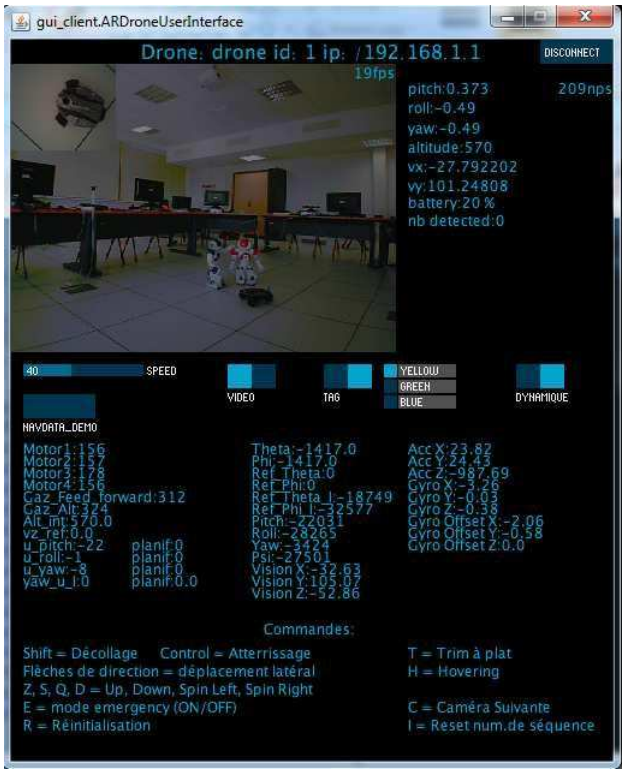


FIGURE 11.6 – Picture-in-Picture display

will automatically land.

The drone may end up in the state of emergency when carried and turned to its back, or even after a crash. Then, the emergency mode warning will be displayed on the video flow (fig. 11.7). In this mode, the drone will not respond to any control but the emergency command.

When battery turns low (information from the drone state mask), a warning message will be displayed on the video flow (fig. 11.8).

There is also two buttons concerning altitude parameters.

MAX will allow to set the maximal altitude of the drone between two options : 3 meters (button off) or 10 meters (button on).

CORRECTED will apply a correction (a cosine function depending on pitch and roll of the drone) on the measured altitude.

11.3.3 Multicast

Navdata and video can be multicasted. To do so, an active client have first to connect to the drone in multicast mode.

When clicking on the MULTICAST button on the network configuration window, a text field will appear in which the IP address of the wifi interface on your computer have to be filled (fig. 11.9).

There is no difference for the active client in multicast mode, but another computer(s) connected to the drone wifi network can execute this application in passive mode (MULTICAST_INTERFACE_IP text field needs to be filled too). The passive client(s) will be able to display the video flow and navigation data information in the GUI (fig. 11.10), but only the active client will be able to send controls to the

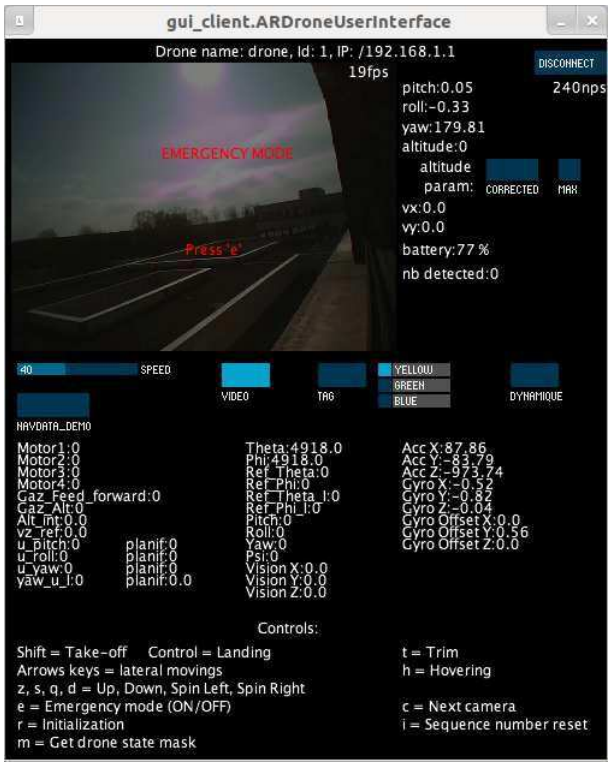


FIGURE 11.7 – Emergency mode

FIGURE 11.8 – Low battery

drone.

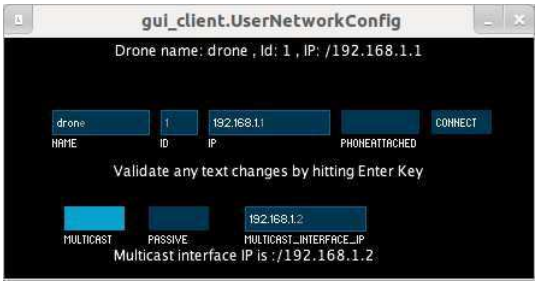


FIGURE 11.9 – Multicast connection

FIGURE 11.10 – Multicast passive interface

This may be useful for example for video processing, recording or navigation monitoring.

In case of connection with several drones (see 11.3.4), only the leader can be allow to broadcast in multicast way.

11.3.4 Flight scenarios

The user interface allows to launch two different scenarii : tag tracking and dynamic following.

Tag tracking

The drone has a tag detection function. The tags to be detected are 2D tags (fig. 11.11) or the outdoor hulls of the drone. The color of the tag may be chosen by the radio buttons yellow, green or blue.

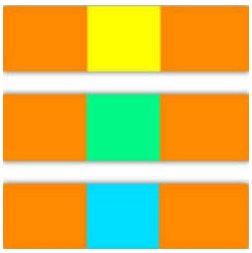


FIGURE 11.11 – Yellow 2d Tag

When a tag is detected, a circle will be drawn around it on the video flow. Distance and coordinates in x and y axis are indicated on the user interface (fig. 11.12). When the tag tracking mode is activated, the drone will take-off and follow the detected tag.

FIGURE 11.12 – Tag detection

The tag tracking uses the built-in feature provided by the drone. The results vary depending on the lightning conditions.

Dynamic follow

The dynamic follow involves several drones. One will be the leader, other(s) will reproduce(s) its movements.

To do this, all the drones have to be configured to belong to the same wifi ESSID. This is a bit tough because of Parrot limitations on drone wifi configuration. The configuration is non permanent and has to be done on every drone boot.

The principle is to configure the wifi connection on one drone to be managed,

with the same ESSID and frequency that the other drone, and set its IP address to 192.168.1.2, the other drone keeps its default value (192.168.1.1).

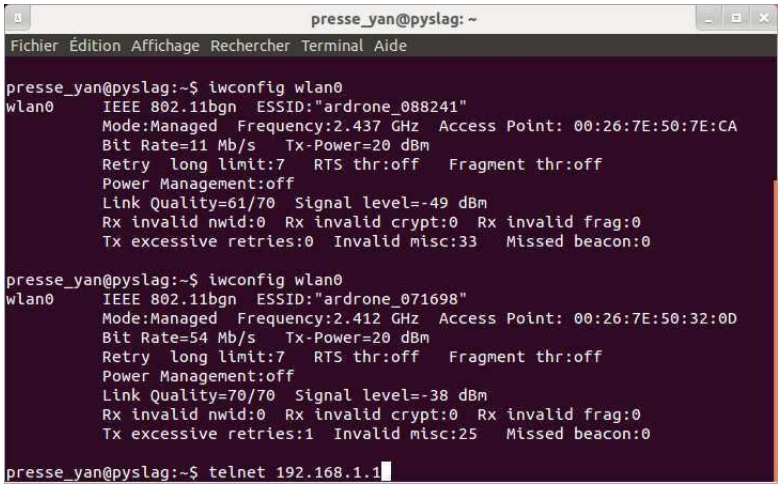


FIGURE 11.13 – Drones wifi characteristics

Here is an example for two drones :

- first, manually configure your IP address on your computer to 192.168.1.3 (or above)
- turn-on the drones, connect the computer (wifi connection) with one of them ("drone a") and note the ESSID and the frequency (fig. 11.13). In this example, ESSID is ardrone_088241 and frequency is 2.437
- adapt the following script with ESSID and frequency previously noted

```
echo ifconfig ath0 down > /data/mode_multi.s
echo iwconfig ath0 mode managed >> /data/mode_multi.s
echo iwconfig ath0 essid ardrone_088241 >> /data/mode_multi.s
echo iwconfig ath0 freq 2.437 >> /data/mode_multi.s
echo ifconfig ath0 up 192.168.1.2 netmask 255.255.255.0 >> /data/mode_multi.s
chmod +x /data/mode_multi.s
/data/mode_multi.s
```

- connect the wifi to the second drone ("drone b"), telnet it (telnet 192.168.1.1), paste the modified script, and valid by the enter key (Windows Seven users may need to first activate the telnet client by typing "pkgmgr /iu :TelnetClient" and the telnet server by "pkgmgr /iu :TelnetServer"). From now on "drone b" will not respond since his network configuration changed
- connect to the wifi network on the first drone, "drone a"
- verify the connection with pings (fig. 11.14) on each drones (192.168.1.1 and 192.168.1.2)
- if the drone b doesn't respond, you may need to do again the third previous steps, after having reboot the "drone b"
- launch ARDroneDemo
- create two drones and configure corrects name, ID and IP (fig. 11.15)
- finally, this is done (fig. 11.16)

Please note that in this network configuration, the "drone a" is set as access point, that means that all communications will pass through it, even communications from "drone b" to the computer like video flow and navdata flow. If you're going to make a try with a lot of drones, "drone a" may be overloaded.

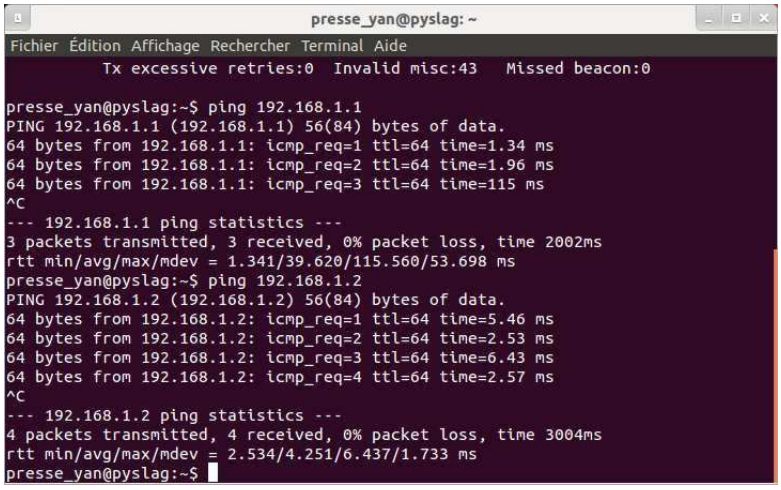


FIGURE 11.14 – Ping on each drone

A trial in ad-hoc mode (that means configuring drones and computer in ad-hoc mode) has not been conclusive on our side with two drones, it seems like the wifi interface on the computer was too weak (wifi USB key).

Another way will to use a wifi router as access point and configure drones in managed mode.

FIGURE 11.15 – Drones configuration in the software

Take-off individually each drones. The leader will be the drone with the ID set to 1. The follower will be the second one. By moving the leader, the follower will repeat the same movements.

The dynamic follow use the navdata of the leader (pitch, roll, yaw) to create controls for the follower. So, the leader doesn't need to take-off. You can manually move it, the follower will move in the same way.

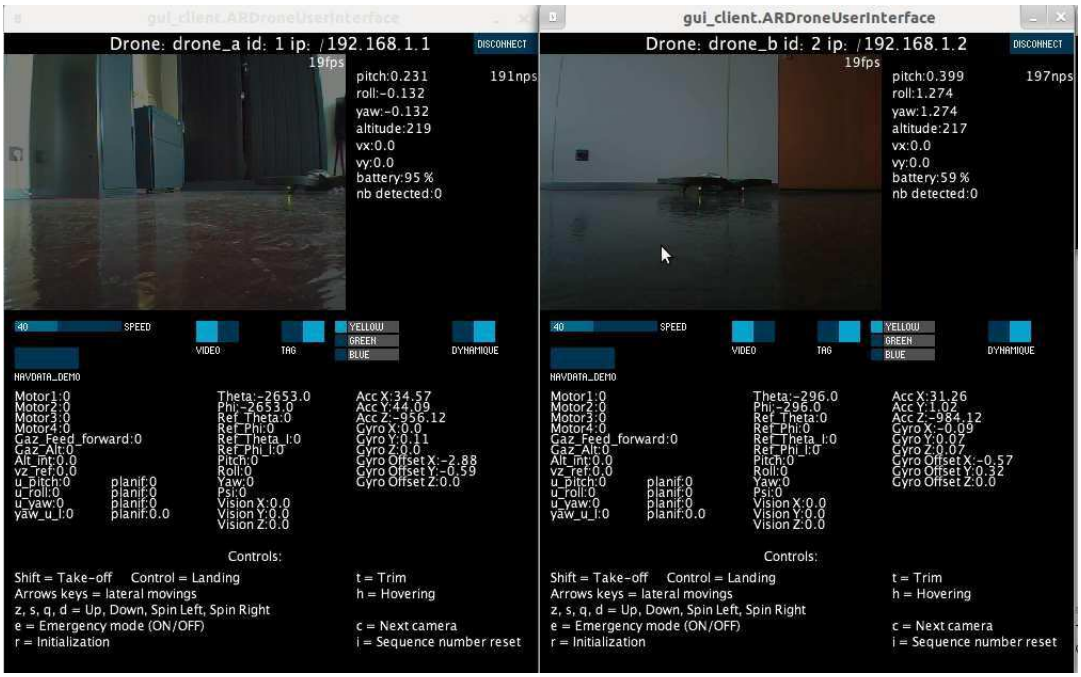


FIGURE 11.16 – Two connected drones

11.4 Non-interactive way

You may also control the drone in a non-interactive way (it means with a pre-programmed scenario) from you own java program using the ARDroneEntity class from the ardronelib package. Below is an example you can find in the Autopilot class. See also ARDroneEntity class documentation located in the doc repertory.

```
myDrone.trim();

myDrone.takeoff();

myDrone.hover(1000);

myDrone.goForward(600);

myDrone.goBackward(800);

myDrone.goForward(1500, 10);

myDrone.goBackward(2000, 10);

myDrone.hover(1000);

myDrone.landing();
```

The drone will take-off, hover during 1 second, go forward during 0,6s, go backward during 0,8s, go forward during 1,5s at 10% of speed, go backward during 2s at 10% of speed, hover during 1 second then land.

The first argument is the time of movement in millisecond, and the optional second argument is the speed of movement in % (the default value is 40%).

The combinedMoves method permits multiple movements at once. The arguments are :

- time in millisecond
- speed in % (negative value to go forward, positive value to go backward)
- speed in % (negative value to go down, positive value to go up)
- speed in % (negative value to spin left, positive value to spin right)

Here is an example :

```
myDrone.combinedMoves(2000, -40, 30, 50);
```

In the example above, the drone will move during 2 seconds and will go forward at 40% of speed, go up at 30% of speed and spin right at 50% of speed, all in the same move.

This is the full list of methods available :

- myDrone.trim() is trim
- myDrone.takeoff() is take-off
- myDrone.hover(int time) is hovering
- myDrone.goForward(int time, opt int speed) is go forward
- myDrone.goBackward(int time, opt int speed) is go backward
- myDrone.goLeft(int time, opt int speed) is go left
- myDrone.goRight(int time, opt int speed) is go right
- myDrone.up(int time, opt int speed) is go up
- myDrone.down(int time, opt int speed) is go down
- myDrone.spinLeft(int time, opt int speed) is spin left
- myDrone.spinRight(int time, opt int speed) is spin right
- myDrone.combinedMoves(int time, int speedFB, int speedDU, int speedYawLR) is a combined move
- myDrone.landing() is landing

A warning will be displayed in case of low battery and the scenario won't be executed. A warning will also be displayed if the drone is in emergency mode, and the user will be asked to choose to continue (emergency mode will be turn off) or to end the scenario.

11.5 Known issues

Sometimes the program freezes after pressing the CONNECT button on the configuration window. Other times the user interface window is not drawn properly. Close the program, and try again.

Chapitre 12

Documentation développeur

12.1 About

The software was originally developed as part of a student project by Paulin Andurand, Jeremie Hoelter, Mathieu Plachot and Baptiste Tissot under the direction of Laurent Ciarletta at Ecole Nationale Supérieure des Mines de Nancy on 2011. A couple of videos are available on youtube :
http ://www.youtube.com/user/ARDroneMines

This version was developed by Yannick Presse as part of EiCNAM Engineer Diploma Thesis at Supelec and was supervised by Virginie Galtier.

This software is released under a GPL version license.

12.2 ARDroneDemo package

12.2.1 gui_client

This package includes the gui for the interactive client. It consists of three classes : main one is located in ARDroneDemo.java

They use Napplet (Papplet for the main one) and controlP5 in order to draw the windows and manage buttons, sliders and text field (named controllers). The characteristics of these components are located in setup() methods.

A controller (button, slider, etc.) is attached to a method by his name, i.e the button NewDrone will call the method NewDrone().

ARDroneDemo.java

The NewDrone() method creates two Napplets and calls the config() method included in UserNetworkConfig.java

UserNetworkConfig.java

This software can manage several drones, each is considered as an entire entity. The config() method creates a new ARDroneEntity (named drone). When network configuration is done, the button "connect" calls the connect() method of this new drone entity, then set the ARDroneUserInterface Napplet as visible, and the windows closes.

ARDroneUserInterface.java

This is the graphical interface. Video and navigation datas (NavData) are shown in the window, a keyboard listener and several controllers are used to control and set up the drone. This interface uses the ardronelib.jar library :

- control.java is used to send displacement or configuration commands to the drone
- VideoHandler.java, ImageListener.java and package parrot.video are used to manage the video flow
- NavDataHandler.java, NavDataParser.java and packages lib.ardrone.listeners and lib. ardrone.navigationdata.structures are used to collect, parse and dispatch navigation data information

12.2.2 non_interactive_client

It consists of the Autopilot.java class allowing the course of a pre-programmed scenario located in this class. The main method creates a new ARDrone entity, set up the network configuration and connect to the drone.

The scenario uses methods located in the ARDroneEntity.java class (package lib.ardrone) : first it execute tests for emergency mode and low battery states, then it performs its displacement.

12.2.3 Library

The following packages are mandatory :

- ControlP5.jar is used by the gui for add buttons, sliders, controllers
- core.jar is for processing
- napplet.jar is for processing sketches embedded in other sketches
- ardronelib.jar is the software library used by the gui client and the non interactive client

12.2.4 data

It consists of one font used by the gui interface.

12.3 ardronelib package

12.3.1 lib.ardrone

These are the classes including the drone entity, the registry for multi-drones configuration (dynamic follow for example) and the global variables such as the UDP ports used.

ARDroneEntity.java

This can be considered as the core of the lib. Here is located several variables and methods used by a lot of other classes.

Here is performed the connection initialization : this software is multi-threaded, so the connect() method will create and start threads for manage controls (Control.java), video (VideoHandler.java), NavData (NavDataHandler.java) and also phone (PhoneServer) and flight scenarii (FlightScenario.java) when needed.

Each thread (control, video and navdata) is attached to a socket manager as illustrated in figure 12.1.

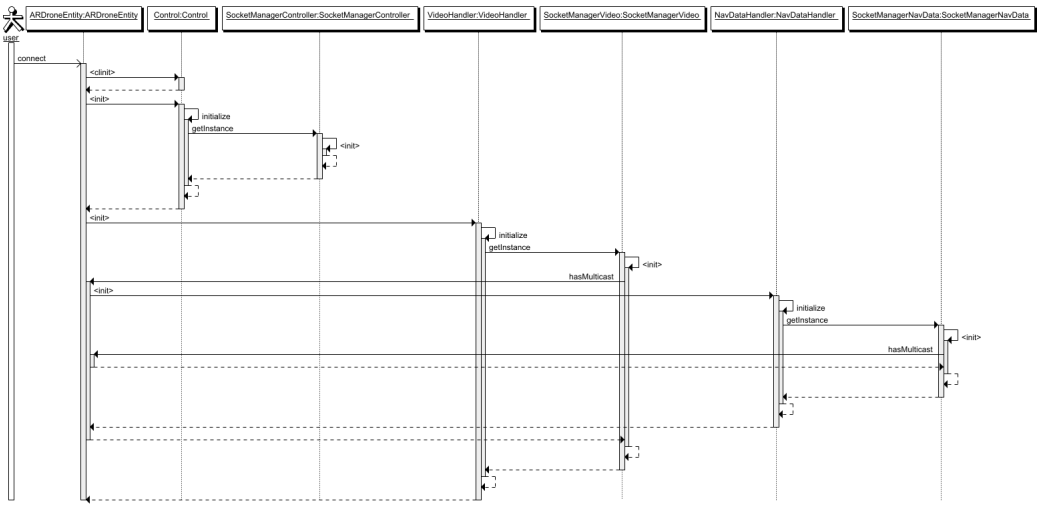


FIGURE 12.1 – Sequence diagram (threads view)

Each socket manager will manage an UDP port (5554 for navdata, 5555 for video and 5556 for control).

This class also manages drone's disconnection.

ARDroneRegistry.java

This class is for manage multi-drones scenario.

An array list of ARDroneEntity is created, then the scenario can add or remove a drone and get the leader.

Global.java

Here is a list of public static parameters such as UDP ports, multicast group to join, etc.

12.3.2 lib.ardrone.control

It consists of the class control.java. This is the thread used for send the ATcommands to the drone.

As connection uses an UDP port (non connected protocol), the correct order of received packets is not guaranteed. Thus, a sequence number is send with each command and drone will not respond to older commands.

This thread will first send some configuration commands (initialize() method). In the run() method, a loop is executed in which commands are sent :

- either a user displacement command (ATconstruct is true)
- or a scenario displacement command (atCommand isn't null), this command may be repeated until continuance is true
- or a landing command, even if the drone is still landed ("AT*REF=" + (seq++) + ",290717696")
- or a hovering command if drone has taken off ("AT*REF=" + (seq++) + ",290718208")

The others methods of this class are used by the GUI or the different scenarii in order to construct and/or send specific commands and they are self-explanatory.

12.3.3 lib.ardrone.flightscenarii

This package includes all scenarii available such as the tag tracking, the dynamic follow and others experimentals ones.

DynamicFollowing and TagFollowing extends the FlightScenario.java thread.

DynamicFollowing.java

The principle is to detect if the drone is the leader or not(via the registry), to get the attitude parameters (yaw, pitch, roll and altitude) of the leader and to send them to the drone via the move() method of the control thread : this.drone.getController().move()

TagFollowing

The principle here is to get the parameters of the detected tag (located in the NavDataVisionDetect.java structure, see the explanation done below) which consist in three axes (x, y and z), to compute the norm of the vector drone-tag, and finally adjust the displacement of the drone via the advancedMove() method of the control thread : this.drone.getController().advancedMove(vectX, vectZ, vectY, yaw)

12.3.4 lib.ardrone.listeners

These are all the listeners used for update the video and the navdata informations shown on the gui (an explanation is done below).

12.3.5 lib.ardrone.navigationdata.structures

These are all the structures of the navdata contained in the navdata informations flow (an explanation is done below).

12.3.6 lib.ardrone.navigationdata

This package consists of the navdata thread (NavDataHandler.java) and parser needed for the navdata.

NavDataHandler will first perform an initialization (creation of all of the structures). Then, in the run() :

- The drone have to receive specific buffer on its NavData UDP port before it begins to send navigation data information. The tickleNavDataPort method send this buffer depending on the wanted connection mode (unicast or multicast)
- Instantiation of all of the listeners
- Then, in loop, reception of a packet and call to the parser

The parser will dispatch the buffer and call the appropriate listener. The listener will set the parameters in the corresponding structure. Then a client interested in this information (ARDroneUserInterface.java for the GUI_client) may get data via NavDataHandler. For example to get pitch information, the following command is needed : drone.getNavDataHandler().getNavDataDemo().getPitch()

The figure 12.2 shows an example of how is parsed a navdata structure.

This example is made of only one structure (NavDataDemo) and one listener (AttitudeListener), but in fact all of the structures and listeners participate in the navdata work.

In order to parse the NavData stream (fig. 12.3) the NavDataParser will :

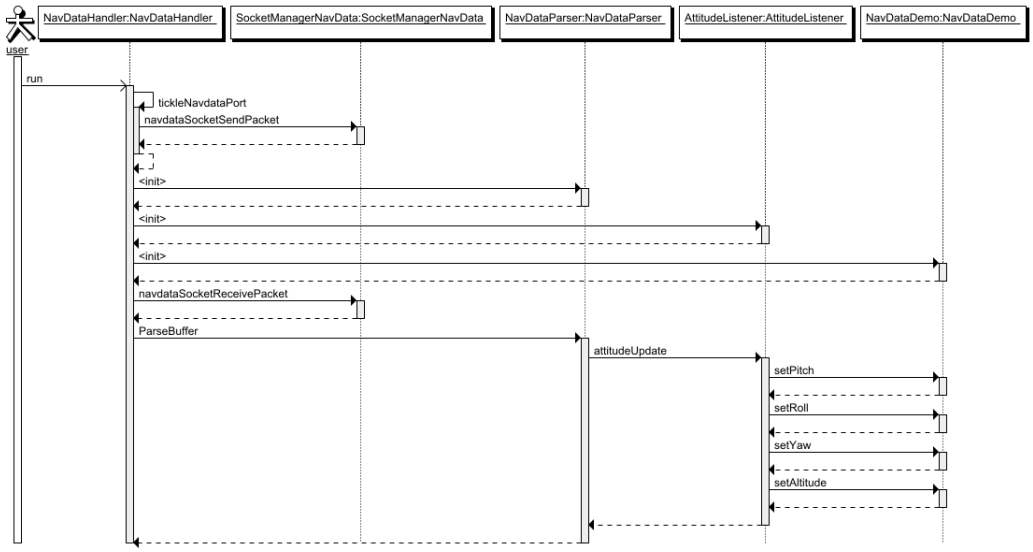


FIGURE 12.2 – Sequence diagram (navdata view)

Header	Drone State	Numéro de séquence	Flag vision	Block 1			Block ...	Checksum
				id	Taille	Données		
32 bits	32 bits	32 bits	32 bits	16	16	...	...	64 bits

FIGURE 12.3 – NavData Stream

- Grab the header (called magic in the NavDataParser code) and verify it (0x55667788)
- Grab the drone state and update the stateMaskListener
- Grab an integer called vision (unused up to now, reserved for future use by Parrot, it is not related to tag detection algorithm)
- Grab and verify the sequence number (it's the same purpose that the control sequence number)
- Finally, grab, dispatch (depending on the id tag) and process the data blocks

12.3.7 lib.ardrone.network

These are the three classes managing the sockets for the three different flows : Atcommands (performing controls), navdata and video.

A socket manager creates a datagram socket (or a multicast socket for navdata and video when needed). Then it allows handlers (control, video or navdata) to send or receive a packet.

12.3.8 lib.ardrone.phone

The PhoneServer.java is a class used to handle the communication with the phone attached to the drone (when needed). This thread works in the the same way that the NavDataHandler thread with a PhoneDataListener but parsing is done into it by the recuperationParams() method.

12.3.9 lib.ardrone.video

This the thread managing the video flow.

It works in the same way than the NavDataHandler thread with an imageListener and it used the Parrot library in order to decode the video flow.

12.3.10 lib.utils

It consists of one class performing a couple of fonction :

- getDateTime is used by the navdata parser to compute nps (navdata per second)
- intOfFloat is used by the control thread to transform the displacement value (in percent) in a single-precision IEEE-754 floating-point value.

12.3.11 parrot.video

This the library provided by Parrot in order to decode the video flow.

Bibliographie

[1] AAL JP. Ambient Assisted Living Joint Programme of European Commission (FP7), 2007. http ://www.aal-europe.eu.

[2] ERDF. Dossier de presse : *Le compteur communicant Linky d'ERDF : une expérimentation réussie*, juillet 2011.

[3] INRIA Activity Report 2011, Project-Team MADYNES. *Management of dynamic networks and services.*

[4] EvAAL. The EvAAL contest : Evaluating AAL systems through Competitive Benchmarking. http ://evaal.aaloa.org/.

[5] Bernard P. Ziegler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation, second edition.* Academic Press, 2000.

[6] Youssef Monsef. *Modélisation et simulation des systèmes complexes.* Lavoisier Technique et Documentation, 1996.

[7] Grégory Zacharewicz. *Un environnement G-DEVS/HLA : application à la modélisation et simulation distribuée de workflow.* PhD thesis, Université Paul Cézanne, Aix-Marseille III, 2006.

[8] Agostino Bruzzone. Federating multidisciplinary simulators, 2011. DIPTEM University of Genoa.

[9] Lui Kam, Xavier Lecinq, Pascal Cantot, and Dominique Luzeaux. ITCS : l'infrastructure technique commune dédiée à la simulation pour l'acquisition. In *Les partenariats NATO-PfP/Industrie/Nations dans le domaine de la modélisation, Conférence NMSG RTO*, Octobre 2002.

[10] Pascal Cantot. Introduction à la simulation. Janvier 2010. http ://simu-centre.free.fr/cours/.

[11] Agostino Bruzzone. Fundamentals of interoperability in modeling simulation, 2011. DIPTEM University of Genoa.

[12] Edward T. Powell and J. Russel Noseworthy. The test and training enabling architecture (tena), 2012.

[13] VanWinkle. Tena technical introduction course, 2011. https ://www.tena-sda.org/download/.

[14] Farid Mamaghani. An introduction to sedris, 2008. http ://www.sedris.org/stc/2001/tut1trpl.htm.

[15] Harry Chan, Filip Perich, Tim Finin, and Anupam Joshi. Soupa : Standard ontology for ubiquitous and pervasive applications. In *International Conference on Mobile and Ubiquitous Systems*, 2004.

[16] Julien Siebert. *Approche multi-agent pour la multi-modélisation et le couplage de simulations.* PhD thesis, Université Henri Poincaré Nancy 1, 2011.

[17] Philippe Le Marrec. *Cosimulation multiniveaux dans un flot de conception multilangage*. PhD thesis, INPL Grenoble, 2000.

[18] SISO Product Development group. *Generic Methodology for Verification and Validation (GM-VV)to support acceptance of models, simulations and data*, 2010.

[19] RTO NATO M&S Task Group 008. Nato hla certification. Technical report, OTAN, 2002.

[20] Michael Imbrogno, Wayne Robbins, and Gerard Pieris. Selecting a hla run-time infrastructure. Technical report, Defence Research and Development Canada, 2004.

[21] DGA/CAD. Cours Escadre-HLA : Intégration de HLA à ESCADRE. 2010, http ://simucentre.free.fr/cours/.

[22] Judith S. Dahmann, Richard M. Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Winter Simulation Conference*, 1997.

[23] Modelisar ITEA2 Project. Functional Mock-up Interface for Co-Simulation, 2010. Standard Specification.

[24] OMG. Data distribution service (DDS) Brief, 2011. White paper.

[25] How to Develop True Distributed Real Time Simulations ? Mixing IEEE HLA and OMG DDS standards. In *2011 Spring Simulation Interoperability Workshop*, 2011.

[26] Akram Hakiri. *Mise en réseau de simulateurs de conduite*. PhD thesis, Université Paul Sabatier, Toulouse, 2011.

[27] Akram Hakiri, Pascal Berthou, and Thierry Gayraud. Design of low cost pc-based simulators for education and training purpose using dds. In *International Conference on Computer as a Tool (EUROCON) 2011*.

[28] Akram Hakiri, Pascal Berthou, and Gayraud Thierry. Addressing the challenge of distributed interactive simulation with data distribution service. In *Euro Simulation Interoperability Workshop 2010*.

[29] Valery Raulet. *Prototypage interactif et collaboratif*. PhD thesis, Université de Bretagne occidentale, LI2 ENIB, 2003.

[30] Aron Rubin, Carl Hein, and Guru Prasad. [multisimulation interface for complex simulation.

[31] Aron Rubin and Carl Hein. MSI primer : MSI description. 2006, http ://msi.sf.net.

[32] Luiza Gheorghe. *Continuous discrete co-simulation interfaces from formalization to implementation*. PhD thesis, École Polytechnique de Montréal, août 2009.

[33] Cécile Hardebolle, Frédéric Boulanger, and Christophe Jacquet. ModHel'X : Heterogeneous systems and multi-paradigm modeling, 2012.

[34] iTETRIS Project Consortium, 2010. http ://www.ict-itetris.eu.

[35] Dmitry Kachan. Integration of ns-3 with matlab/simulink. Master's thesis, Luleå University of Technology, 2010.

[36] Emanuele Galli. Projet HLA-OMNet, 2006. http ://www.ce.uniroma2.it/ egalli/projects.html.

[37] Christoph sommer, Reinhard German, and Falko Dressler. Bidirectionally coupled network and road traffic simulation for improved ivc analysis. 2011.

[38] Zhi Zhang, Zhonghai Lu, Qiang Chen, Xiaolang Yan, and Li-Rong Zheng. COSMO : CO-Simulation with MATLAB and OMNeT++ for Indoor Wireless Networks. In *IEEE GLOBAL COMMUNICATIONS CONFERENCE (IEEE GLOBECOM 2010)*, 2010.

[39] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Jist : Embedding simulation time into a virtual machine. In *EuroSim congress on medelling and simulation*, 2004.

[40] OMNeT++ Community. Site web du projet OMNeT++ : Network Simulation Framework, 2009. http ://www.omnetpp.org.

[41] NS-3. Site web du projet NS-3 : Network Simulation, 2012. http ://www.nsnam.org.

[42] Tommaso Pecorella and Mudit Raj Gupta. HLA interfaces for NS-3, 2012. http ://www.nsnam.org/wiki/index.php/GSOC2012HLA.

[43] OPNET. Site web de OPNET : Modeler, accelerating network R&D, 2012. http ://www.opnet.com/solutions/networkrd/modeler.html.

[44] Industrie et Technologies. Une plate-forme de simulation pour les drones, 2011. http ://www.industrie.com/it/une-plate-forme-de-simulation-pour-les-drones.11819.

[45] The Player Project. Free software tools for robot and sensor applications, 2010. http ://playerstage.sourceforge.net.

[46] FlightGear. Site web : Sophisticated, professional, open-source simulation, 2012. http ://www.flightgear.org.

[47] Adrien Guenard. *SAMOVAR 2.2 : manuel d'utilisation*, 2011. http ://samovar.loria.fr/.

[48] C. Stenzel and S. Pawletta. Matlab HLA Toolbox, 2008. http ://www.mb.hs-wismar.de/ stenzel/software/MatlabHLA.html.

[49] *ARDrone SDK 1.7 Developer Guide*, 2011. https ://projects.ardrone.org/.

[50] Paulin Andurand, Jeremie Hoelter, Mathieu Plachot, and Baptiste Tissot. Création d'une plateforme de développement robotique en java pour l'AR.Drone. Technical report, ENSMN, 2011.

[51] Guillaume Hiel and Etienne Lansin. Pilotage d'un ARDrone virtuel, 2012. Rapport d'un projet de conception de deuxième année.

[52] Florian Besnard, Lilian Chiassai, and Lucas Renaud. Création d'un simulateur de vol pour l'AR DRone de Parrot, 2012. Rapport d'un projet de conception de deuxième année de l'ENSMN.

[53] Kevin Barlett. A simple UDP communications application for Matlab, 2010. http ://www.mathworks.com/matlabcentral/fileexchange/24525-a-simple-udp-communications.

Étude et conception d'une plate-forme de multisimulation et de contrôle de drones. Mémoire d'ingénieur CNAM, Metz 2012.

Résumé

Les solutions d'informatique ambiante sont difficiles à développer et évaluer à cause de l'hétérogénéité des éléments en jeu et de leurs nombreuses interactions. Le recours à la multisimulation — faisant collaborer différents simulateurs exécutant chacun une partie de la simulation complète — permet d'obtenir une simulation globale et riche du système étudié.

Une fois établies les correspondances aux frontières des modèles des différents simulateurs, notre difficulté est alors de faire collaborer les logiciels associés, prévus pour fonctionner de manière autonome. Le travail réalisé a consisté à concevoir une architecture logicielle couplant un simulateur réseaux (OMNet++), un simulateur de drones (SAMOVAR) et une interface de contrôle de quadricoptères, et à adapter ces simulateurs afin de les faire collaborer grâce à des échanges réseaux. Le multisimulateur obtenu permet le vol en formation de drones réels et virtuels en utilisant des informations issues d'un réseau simulé et réalise ainsi l'émulation d'une escadrille de drones. Il peut être utilisé pour étudier l'influence d'un changement de protocole réseaux sur le maintien en formation des drones.

De plus, l'interface du multisimulateur permet de contrôler à distance de manière interactive ou programmée (en Java) un ou plusieurs drones, réels ou virtuels.

Mots clés : simulation distribuée, multisimulation, AR.Drone, simulateur OMNet++, simulateur SAMOVAR, émulation.

Abstract

Pervasive computing solutions are difficult to develop and assess because they involve many heterogeneous elements with multiple interactions. Multisimulation consists in making different simulators cooperate, running each part of the whole simulation; it provides a rich and comprehensive representation of the system which is studied. Once the correspondences have been established at the edges of underlying models of the different simulators, the difficulty is to coordinate the associated software as they are designed to operate autonomously. A platform combining through network exchanges a network simulator (OMNeT++), a UAV (Unmanned Aerial Vehicle) simulator (SAMOVAR) and a quadricopter control interface was designed and developped. The resulting multisimulator supports flocking scenarii where virtual and actual UAVs base their decisions on information from a simulated network. It achieves an emulation of a UAVs squadron. It can be used to study the influence of a change in network protocol on keeping coherent movements. In addition, the developped UAV control interface can be used to remotely control one or more UAVs, either real or virtual, interactively or thanks to a (Java) programme.

Keywords: distributed simulation, multisimulation, AR.Drone, OMNet++ simulator, SAMOVAR simulator, emulation.