M2M - Drone

Version 2010-2011

Thomas Calmant
François-Karim Laben

26 April 2011

Table des matières

CHAPITRE 1

Préparation du terrain

1.1 Description du projet

1.1.1 Objectif

Le projet consiste à réaliser un drone rondier en milieu industriel. Ce rondier devra être en mesure de se déplacer en autonomie, de mesurer des grandeurs physiques, les renvoyer à un serveur et ne rentrer à la base que pour être rechargé.

Dans le cadre du M2M, nous devons intégrer un ensemble de technologies déjà existantes et inclure une partie de contrôle d'une machine par une autre. Ainsi, le rondier devra être autonome pour effectuer l'ensemble des missions qui lui seront attribuées.

1.1.2 Le matériel

Le matériel utilisé pour le projet est le suivant :

- Un A.R. Drone fabriqué par Parrot. L'A.R. Drone est un drone quadricoptère pilotable à distance en Wifi Ad-hoc depuis un terminal (iPhone, iPad, PC Windows ou Linux).
  - Tourne sur un processeur ARM 9
  - Basé sur un noyau Linux 2.6.27
  - Shell simplifié (Busybox)
  - Carte wifi Atheros
  - Ports *ftp* (21) et *telnet* (22) ouverts. Accès **root** sur telnet.
  - Au moins 64 Mo de mémoire flash sont disponibles et inscriptibles via *ftp*. Ils sont disponibles dans le dossier */data/video*.
  - Dispose d'un port USB On-The-Go (OTG), lui permettant d'être hôte ou esclave.
  - Dispose de 2 caméras (sol et frontal)

  Le drone tient environ 10 minutes sur batterie en temps de vol, 90 minutes en liaison wifi seule.
- Un compteur Geiger connectable en USB, de chez SparkFun
- Un téléphone Nokia N95. L'un des rares à gérer correctement le Wifi Ad-Hoc utilisé par le drone. Il fournit également la connectique 3G.
- Un serveur HTTP depuis lequel on récupèrera les ordres et à qui on enverra les grandeurs physiques mesurées.

Évolution

L'architecture et le matériel utilisé pour mener à bien ce projet ont évolué en fonction des problèmes rencontrés.

On notera particulièrement l'obligation d'utiliser un périphérique effectuant la liaison entre le réseau Wifi local du drone et le réseau 3G permettant l'accès à Internet. Dans notre cas, il s'agit d'un téléphone N95, choisi pour ses

FIG. 1.1 – L'A.R. Drone de Parrot

capacités GPS et sa gestion du Wifi Ad-hoc. On notera que le drone n'est capable de porter le N95, pesant environ 120g, que s'il est utilisé avec sa coque pour le vol en *extérieur*.

Ce choix a été forcé par l'incapacité du pilote USB hôte de Synopsys à gérer les hubs USB.

1.1.3 Intérêt et portée du document

Ce document présente les travaux effectués afin d'atteindre l'objectif du projet.

Pour ce faire, les moyens et méthodes seront décrites et des explications seront fournies sur les difficultés que nous avons rencontrées. Les stratégies employées pour les contourner seront indiquées.

1.2 Outils de cross-compilation ARM

1.2.1 Outils testés

Ce projet a été notre premier cas de cross-compilation, nous avons donc dû découvrir les différents outils pouvant nous assister dans cette tâche, parmi lesquels :

Bitbake et OpenEmbedded

OpenEmbedded est une collection de recettes pour BitBake afin d'automatiser la compilation d'outils pour une plate-forme cible (architecture, système, bibliothèques, fonctionnalités...).

Le principal problème de cette distribution est que toutes les dépendances nécessaires à l'outil sont téléchargées et compilées, y compris la chaîne GCC complète.

Dans le cas de Python, la compilation prend plus de 3 heures sur un processeur quadri-cœurs à 3.2 GHz, et occupe plus de 8 Go d'espace disque. La compilation de Python correspond à environ 370 recettes, on peut imaginer le temps et l'espace nécessaire à la compilation de JamVM correspondant à plus de 1000 recettes.

Site web : http ://www.openembedded.org/

Dépôts binaires Debian ARM

Une autre technique est de récupérer les fichiers voulus dans les paquets .deb correspondant à l'architecture ARM.

Nous utilisons notamment cette technique pour récupérer des bibliothèques "basiques" manquantes. Nous ne pouvons cependant pas nous permettre de l'appliquer sur des programmes "complexes", tels que Python, JamVM ou autre, car cela entrainerai trop de dépendances, la plupart inutiles dans notre cas.

CodeSourcery G++ Lite

CodeSourcery fourni une version pré-compilée de la chaîne de compilation GCC permettant de créer des exécutables pour processeurs ARM.

Cet outil est basé sur GCC 4.5.2, ce qui n'a pas posé de problème pour compiler des modules pour un noyau lui-même compilé avec GCC 4.3.3.

Site web : http ://www.codesourcery.com/sgpp/lite/arm/

Cross-compilation avec Code Sourcery

Il y a deux techniques pour cross-compiler un outil avec CodeSourcery, dépendant de la manière dont fonctionne le script `configure` utilisé :

– Méthode des variables d'environnement : on force la variable d'environnement **GCC** lors de la configuration :

```
$ GCC=arm-linux-gcc ./configure
$ make
$ make install
```

– Méthode *–host* : on utilise les paramètres *–host* du script de configuration :

```
$ ./configure --host=arm-linux
$ make
$ make install
```

Dans ce cas, le contenu de *–host* doit être le préfixe utilisé par la chaîne de cross-compilation : ici, on utilisera donc `arm-linux-gcc`.

Dans certains cas, il est nécessaire d'indiquer le paramètre *–target* avec la même valeur que *–host*.

1.3 Ajout de modules noyau au drone

1.3.1 Compilation des modules

Configuration du noyau

Pour qu'un module puisse être ajouté à un noyau en place, il doit être compilé en adéquation avec la configuration de celui-ci.

Dans notre cas, celle-ci est disponible sur le site des développeurs A.R. Drone, à l'adresse : https ://projects.ardrone.org/projects/ardrone-api/documents.

Compilation des modules

Nous avons commencé par sélectionner les modules à compiler à l'aide de la commande :

```
$ make ARCH=arm menuconfig
```

La spécification de l'architecture est nécessaire pour assurer le bon fonctionnement des scripts de configuration.

La compilation des modules s'est effectuée simplement à l'aide de la commande :

```
$ make
   # 5 compilations parallèles (nombre de CPU + 1)
   -j 5 \
   # Architecture
   ARCH=arm \
   # Outils de cross-compilation
   CROSS_COMPILE=/opt/CodeSourcery/Sourcery_G++_Lite/bin/arm-linux- \
   # Règle make
   modules
```

Ici, la variable **CROSS_COMPILE** doit contenir le préfixe complet des outils de compilation à utiliser, chemin d'accès compris.

1.3.2 Modules ajoutés

DWC OTG

Ce module permet au drone de gérer la norme USB On-The-Go, lui permettant d'être soit client, soit hôte USB.

Dans la version livrée avec les sources du noyau du drone, seul le mode client est activé, nous l'avons donc modifié comme indiqué sur le blog E/S and I.

Pour compiler ce module, nous l'avons sélectionné à l'aide de `make menuconfig`, en suivant le chemin :

Chemin	Description
System Types	Configuration spécifique à la cible
System Types > Parrot Drivers	Pilotes spécifiques au matériel Parrot
System Types > Parrot Drivers > PARROT6 USB Driver (Synopsys) (m)	Pilote DWC OTG : support de l'USB On-The-Go (hôte et esclave)

Pour utiliser le port USB avec le module sur le drone, il ne faut pas oublier d'y activer les 5 Volts nécessaires à son fonctionnement :

```
# Activation des 5V
$ gpio 127 -d i
# Installation du module
$ insmod dwc_otg.ko
```

Ce module a été testé et est considéré fonctionnel sur le drone.

Problèmes rencontrés

Ce module est parfaitement capable de gérer une connexion directe avec un périphérique USB, mais ne peut pas être utilisé avec un hub.

C'est pour cette raison que notre première architecture a dû être abandonnée, du fait de l'impossibilité de connecter un capteur et un module 3G en même temps. On notera également que la clé 3G USB que nous comptions utiliser est vue comme un hub USB, lui permettant d'être reconnue à la fois comme modem et comme périphérique de stockage.

Une solution pour palier à ce problème serait de construire l'équivalent d'un *routeur USB*, c'est-à-dire un périphérique aggrégeant les données reçues depuis ses propres ports hôtes, et envoyant des trames au drone à intervalle régulier.

USB Serial

Ce module permet de dialoguer avec un port USB comme s'il s'agissait d'un port série en utilisant les nœuds */dev/ttyUSBx*.

Une fois fonctionnel, il peut être utilisé pour la lecture des données des capteurs branchés sur le drone, ainsi que pour l'utilisation du GPS par **gpsd**, un outil se chargeant de gérer un périphérique USB et de récupérer les coordonnées en C ou en Python.

Le support d'USB serial a été activé en suivant le chemin :

Chemin	Description
Device Drivers	Pilotes supportés
Device Drivers > USB Support (y)	Support USB niveau noyau, déjà présent
Device Drivers > USB Support > USB Serial Converter Support (m)	Support USB Serial (module à compiler)
Device Drivers > USB Support > USB Serial Converter Support (m) > USB Generic Serial Driver (y)	Pilote USB Serial générique, à compiler dans le module précédent. Facultatif.
Device Drivers > USB Support > USB Serial Converter Support (m) > Prolific 2303 Single port... (m)	Support du convertisseur USB - Série pl2303, utilisé par le GPS Inforad (module à compiler)

Ce module doit être installé après l'activation du port USB en mode hôte. Il se charge de créer le fichier */dev/ttyUSB0* lors de la connexion d'un périphérique reconnu.

1.4 JamVM

1.4.1 Description rapide

JamVM est une machine virtuelle Java, conforme aux spécifications JVM v2. Son objectif est d'être très légère (environ 100Ko) afin de pouvoir cibler le matériel embarqué. Elle supporte JNI et l'API Reflection.

JamVM a besoin d'une bibliothèque externe pour faire fonctionner les applications Java standard : GNU Classpath.

GNU Classpath est une implémentation libre de la Java Class Library, la bibliothèque standard du langage Java, faisant partie du projet GNU. Sa compilation nécessite un compilateur Java autre que celui d'Oracle : nous avons choisi celui d'Eclipse (*ecj*).

1.4.2 Compilation

La compilation de JamVM nécessite la bibliothèque Zlib (nous avons utilisé la version 1.2.3) afin de pouvoie lire les fichiers JAR. Pour faire fonctionner un programme Java, nous avons compilé GNU Classpath 0.98.

Les configurations utilisées pour compilées sont :

```
# Variables d'environnement pour configure et make
$ export AR=arm-linux-ar
$ export CC=arm-linux-gcc
$ export LD=arm-linux-ld
$ export JAVAC=ecj

# /content correspond à un lien symbolique vers /data/video sur le drone :
# c'est un dossier inscriptible par FTP et non touché par le flashage du drone

# Compilation de Zlib
$ ./configure --prefix=/content/jvm/zlib --shared
$ make
$ make DESTDIR=`pwd`/_install install

# Compilation de JamVM
$ CFLAGS=-I/content/jvm/zlib/include \
```

```
  CPPFLAGS=$CFLAGS \
  LDFLAGS=-L/content/jvm/zlib/lib \
  ./configure \
    --target=arm-linux --host=arm-linux \
    --prefix=/content/jvm/jamvm \
    # Répertoire où JamVM cherchera le GNU Classpath à l'exécution
    --with-classpath-install-dir=/content/jvm/classpath \
    # Évite un segfault sur la version 1.5.4
    --enable-int-threading=no --enable-int-direct=no \
    --enable-int-inlining=no
$ make
$ make DESTDIR=`pwd`/_install install

# Compilation de GNU Classpath
$ ./configure \
   --target=arm-linux --host=arm-linux \
   --prefix=/content/jvm/classpath \
   --without-x --disable-gtk-peer --disable-gconf-peer \
   --disable-gjdoc --disable-plugin --disable-examples \
   --disable-Werror
$ make
$ make DESTDIR=`pwd`/_install install
```

Les configurations de Zlib et JamVM sont classiques. Pour ce qui est de GNU Classpath, nous avons utilisé ces options afin de supprimer toute dépendance inutile à des outils graphiques.

Nous avons également utilisé *–disable-Werror* afin de supprimer une erreur de compilation dans la partie réseau de GNU Classpath.

1.4.3 Problèmes rencontrés

Nous avons testé 3 versions de JamVM :
– 1.5.0 : Fonctionnant à peu près, mais certaines commandes entraînent un crash de la libc
– 1.5.3 : Même chose qu'avec la 1.5.0
– 1.5.4 : Cette version est incapable de comparer des entiers *int* et des entiers *long*, ce qui empêche la lecture correcte des fichiers ZIP et donc le fonctionnement de GNU Classpath.

Dans le cas des versions 1.5.0 et 1.5.3, le lancement d'une distribution de base de Felix 3.0.7 (avec le Gogo Shell) dure **65 secondes** !

Dans la cas de la version 1.5.4, cette incapacité à comparer deux types différents d'entiers semble venir de la modification du code de la JamVM, changeant l'utilisation de définition de types internes spécifiques par l'utilisation de types *génériques* (*int8_t*, ...). C'est la plus grosse différence entre les version 1.5.3 et 1.5.4.

1.5 Compilation de Python

1.5.1 Version du drone (cross-compilée)

Python n'est absolument pas prévu pour être cross-compilé : une fois compilé, l'interpréteur est exécuté pour pré-compiler la librairie standard et pour se tester. Évidemment, exécuter un programme ARM sur un x86 pose problème.

Pour ce faire, nous devons compiler Python pour la machine de compiler, qui sera utilisée pour l'exécution des tests.

Nous avons donc utilisé le patch disponible sur le blog Randomsplat.com, inspiré de ceux d'OpenEmbedded.

Nous avons appliqué le patch avant d'exécuter *configure*, et modifié le fichier *pyconfig.h* afin d'ajouter la ligne :

```
#define  PY_FORMAT_LONG_LONG "lld"
```

Sans celle-ci, la compilation de l'interpréteur échouera, cette macro étant utilisée afin de définir le format d'affichage des entiers en mode intératif.

Nous avons ensuite exécuté les commandes suivantes afin de compiler Python :

```
# Compilation minimale pour l'hôte
$ ./configure
$ make python Parser/pgen

# Conservation des outils générés
$ mv python hostpython
$ mv Parser/pgen Parser/hostpgen

# Nettoyage complet
$ make distclean

# Export des variables d'environnement pour Make
$ export CC=arm-linux-gcc CXX=arm-linux-g++ \
$ export AR=arm-linux-ar RANLIB=arm-linux-ranlib \

# Application du patch de cross-compilation
$ patch -p1 < Python-2.6.6-xcompile.patch

# Cross-compilation
$ CFLAGS="-I/opt/zlib" LDFLAGS="-L/opt/zlib" \
  ./configure \
   --host=arm-linux --build=x86_64 \
   --prefix=/content/python \
   --disable-ipv6 --without-pydebug

$ make HOSTPYTHON=./hostpython HOSTPGEN=./Parser/hostpgen \
   BLDSHARED="arm-linux-gcc -shared" CROSS_COMPILE=arm-linux- \
   CROSS_COMPILE_TARGET=yes

$ make install HOSTPYTHON=./hostpython \
   BLDSHARED="arm-linux-gcc -shared" CROSS_COMPILE=arm-linux- \
   CROSS_COMPILE_TARGET=yes \
   prefix=/content/python/_install \
   DESTDIR="/opt/python-arm"
```

La compilation peut s'arrêter sur une erreur lors des tests de la bibliothèque durant la dernière commande : elle peut être ignorée.

Python a été testé et validé sur le drone.

1.5.2 Version N95 (Python for S60)

Pour coder sur le téléphone, nous avons également choisi Python. En effet, Java n'a pas suffisamment de droits pour avoir accès à certaines données, ou pour avoir la main sur les accès réseau.

Nokia propose une version de Python pour Symbian S60, permettant d'utiliser une grande partie de la bibliothèque standard du langage, ainsi que des modules spécifique au système Symbian.

Cet interpréteur est disponible sur Maemo Garage. Il est recommandé de le signer numériquement pour avoir un accès plus complet aux fonctionnalités du téléphone sur le site Symbian Signed.

Problèmes rencontrés

Problèmes d'import

Le code que nous avons développé est fourni dans un seul fichier. À la base, nous avions utilisé la logique des modules de Python, mais certains bugs au niveau des noms de fichiers nous a fait changer de méthode. En effet, lors de la copie de fichiers vers le téléphone en utilisant le câble USB, nous n'avons pas la garantie que la casse de leur nom soit conservée : les noms peuvent alors être en minuscules ou en majuscules, ce qui empêche leur chargement par Python, sensible à la casse.

Problèmes de sockets

Le premier point important sur les sockets sous Symbian est la notion de point d'accès : c'est avec ceux-ci que le téléphone choisi sur quel périphérique une socket est assignée (Wifi ou 3G). Dans le cas du Wifi, le script peut choisir son point d'accès sans contrainte ; dans le cas de la 3G, il peut indiquer le point d'accès par défaut, mais c'est à l'utilisateur de confirmer ce choix.

Un deuxième point à noter est qu'il est impossible sous Symbian de partage un objet Socket entre threads : chacun peut avoir une socket, mais pas la même. On notera aussi qu'une socket serveur ne peut pas être non bloquante, sous peine de voir apparaître des erreurs de types "périphérique occupé".

Enfin, il n'est pas possible de se connecter à un périphérique en indiquant son adresse IP, c'est pour cela que c'est le téléphone qui joue le rôle du serveur.

CHAPITRE 2

Production

2.1 Architecture du projet

2.1.1 Vue globale

Le projet est réparti sur 3 machines :
– le drone, sur lequel tourne un script Python se chargeant de récupérer les ordres connus par le téléphone et de les exécuter,
– le téléphone, sur lequel un script Python se chargea de télécharger depuis le serveur HTTP les ordres à donner aux drone,
– un serveur HTTP, en Python, installé sur une Dedibox (Iliad, Paris).
Le téléphone met en cache les ordres puis les donne au premier drone se connectant : il n'est pas possible de gérer correctement une connexion 3G au serveur et une connexion Wifi au drone en même temps.

2.1.2 Protocoles

Téléphone <-> Serveur

Nous utilisons ici le protocole HTTP standard.
– Côté client, le téléphone utilise la bibliothèque **urllib** pour ne pas avoir à implémenter le protocole HTTP *à la main*.
– Côté serveur, nous utilisons la classe utilitaire **BaseHTTPServer** de Python, permettant d'avoir un serveur HTTP en quelques lignes.
Le serveur HTTP se limite à accepter les requêtes **GET** sur les fichiers */log* et */orders*, ainsi que les requêtes **POST** sur le fichier */log*. Ces dernières servent à ajouter des lignes d'information au fichier journal.

Drone <-> Téléphone

La communication entre le drone et le téléphone se fait de manière spécifique. Le téléphone joue le rôle de serveur, celui-ci n'étant pas capable de se connecter à une machine désignée par une IP fixe.

Dans notre architecture, le téléphone, récupère les ordres en 3G sur le serveur web, se déconnecte puis crée un serveur Wifi UDP.

Le drone peut alors se connecter au serveur sur le téléphone pour récupérer les ordres mis en cache. Il les exécute et expédie toutes les grandeurs physiques qu'il mesure à intervalle régulier au téléphone.

La connexion du drone au téléphone se fait en expédiant une phrase contenant le mot-clé *witches*. Le téléphone doit lui répondre par une phrase contenant le mot-clé *wood*, auquel cas le drone sera considéré connecté. Il va alors

demander au téléphone l'heure en envoyant la requête *GET time*, afin de connaître l'heure réelle d'exécution : l'horloge système du drone est remise au 1er Janvier 1970 00 :00 GMT à chaque redémarrage.

Ensuite, le drone créera 4 threads :
– Le premier thread sert à tester la connexion : le drone envoie toutes les 50ms un *ping* auquel le téléphone doit répondre par un *pong*. La perte de connexion entraîne l'atterrissage et l'arrêt du drone.
– Le second thread sert à récupérer toutes les 2 secondes la position auprès du téléphone en envoyant la requête *GET gps*. S'il y a une réponse du téléphone contenant le mot-clé *GPS* alors, la réponse sera évaluée par l'interpréteur Python pour récupérer les dernières latitude et longitude connues par le téléphone.
– Le troisième thread récupère les informations du capteur chaque seconde, leur ajoute un horodatage puis les ajoute à un cache.
– Le quatrième thread récupère les ordres dans la liste des ordres puis les exécute un à un en laissant une marge temporelle pour que la commande soit bien exécutée et terminée.

2.1.3 Sur le serveur HTTP

À faire

doc précise du serveur (doc code)

Format du fichier d'ordres

Les ordres sont définis dans des dictionnaires Python : un dictionnaire par commande à exécuter et par ligne dans le fichier.

Commande	Paramètre(s)	Description
takeOff		Décoller (environ 4 secondes d'exécution)
goTo	latitude, longitude	Ordonner d'aller au point GPS *(latitude, longitude)*
dontMove	durée	Rester sur place pendant *durée* secondes
land		Atterrir

Quelques exemples :

Commande	Syntaxe
takeOff	{'command' : 'takeoff'}
goTo	{'command' : 'goto', 'latitude' : *latitude*, 'longitude' : *longitude*}
dontMove	{'command' : 'dontmove', 'time' : *durée* }
land	{'command' : 'land'}

Le fichier d'ordres peut comprendre des commentaires : ceux-ci sont formés d'une ligne commençant par un #. Les lignes vides ou mal formatées sont ignorées.

2.1.4 Sur le téléphone

Le script sur le téléphone se charge de récupérer le fichier d'ordres depuis le serveur HTTP. Le contenu de ce fichier est alors filtré et analysés : les commentaires et les lignes vides sont supprimés, tandis que les dictionnaires sont évalués par l'interpréteur Python, devenant de vrais objets.

Le téléphone contient alors une liste de dictionnaire, prête à être envoyée au drone.

2.1.5 Sur le drone

Le drone va d'abord attendre l'établissement complet d'une connexion avec le téléphone. Il va alors lui demander l'heure et la position GPS actuelle, puis la liste des ordres à exécuter.

Dans le cas des commandes *takeOff* et *land*, il ne s'agit que de l'envoi de commandes AT ; la commande *dontMove* correspond à un simple *time.sleep()*.

Dans le cas de la commande *goTo*, le drone va calculer le cap pouvant le mener *à vol d'oiseau* de sa position actuelle à la position cible. L'orientation du drone selon ce cap se fait de manière bloquante à l'aide d'une suite de commandes AT. Lors du déplacement, le cap à suivre est recalculé toutes les 4 secondes, afin de compenser d'éventuelles erreurs de positionnement ou les déviations dues au vent.

Le drone se déplace jusqu'à atteindre le but à environ 10 mètres des coordonnées GPS cibles.

Orientation du drone

L'orientation du drone peut être connue à l'aide des *Navdata* : les informations sur l'état du drone fournies par le logiciel de contrôle. Le cap qui y est indiqué en millièmes de degrés, relativement à l'orientation du drone à son démarrage. Les *Navdata* sont reçues un peu plus de 10 fois par secondes, ce qui permet d'avoir un contrôle assez fin de la rotation du drone lors de son changement de cap. Elles sont en binaire, leur structure étant décrite dans les fichiers d'en-tête C du SDK officiel (*navdata_demo_t*, ...).

Warning : Les *Navdata* sont transmises en binaire en mode **Little Endian**, contrairement aux autres données disponibles.

Pour récupérer ces informations, nous avons dû faire face à un problème de port réseau : les *Navdata* sont émises en continu après avoir effectué une requête sur le port 5554 du drone. Or les paquets *Navdata* sont envoyés à destination du port 5554 de l'IP ayant effectué la requête d'émission.

Dans notre cas, le client et le serveur sont sur la même machine, le serveur s'envoie donc les informations, empêchant le client de connaître le cap. Nous avons donc créé une bibliothèque (*libsocket_fake.so*), que nous spécifions dans la variable d'environnement **LD_PRELOAD** lorsque nous relançons le programme de contrôle (*program.elf*).

LD_PRELOAD est utilisée par *ld* afin de désigner la bibliothèque à utiliser en priorité pour résoudre les symboles. Ainsi, nous avons redéfini la fonction *sendto()* afin de rediriger tous les envois à destination du port 5554 vers le port 6005, nous permettant d'avoir un client sur le drone, et donc de récupérer en local les *Navdata*.

2.2 Conception détaillée

À faire

à compléter / extraire de projet-architecture

2.2.1 Sur le serveur HTTP

Documentation

Simple HTTP server that stores log lines and retrieves order files

class drone_server.**ReqHandler**(*request*, *client_address*, *server*)
HTTP requests handler

do_GET()
Handles GET requests : reads the log file or the orders file

do_POST()
Handles POST requests : logs the received data

send_file(*content*, *code=200*)
Sends a response with the given code and content. Automatically sets the content length header

drone_server.**append_to_file**(*filename*, *line*)
Appends file content and a line feed

drone_server.**read_file**(*filename*)
Reads a file content. Returns a tuple (code, content). Sets the content to FILE_NOT_FOUND_PAGE if the file is not found

drone_server.**run_server**(*port=8080*)
Run the HTTP server "forever"

2.2.2 Sur le téléphone

La classe *start_drone* récupère les ordres, les distribue, rend disponible des coordonnées GPS et rapatrie les valeurs mesurées par le drone.

Documentation

2.2.3 Sur le drone

La classe *drone* gère le contrôle primitif du drone : il s'agit d'incliner le drone autour d'un des axes selon l'attitude souhaitée (cf. la documentation Parrot).

La classe *pilot* se connecte au serveur et effectue tous les traitements de haut niveau pour le drone. Elle comprend en particulier la logique de navigation.

Documentation

Contrôle du drone

Main A.R. Drone controller.

class drone.**Drone**(*drone_com=<drone.DroneCommunication object at 0x3611c90>*)
Controls the Parrot A.R. Drone via AT commands sent to program.elf

batterylevel()
Retourne the battery level percentage

emergency()
Drone emergency stop : stops all motors

gaz_down()
Decrease gaz

gaz_up()
Increase gaz

land()
Safe drone landing

left()
Turns left

right()
Turns right

set_heading(*heading*)
Sets the drone heading. Blocking while not around the wanted value.
Paramètres **heading** – The new heading, in radians

set_max_altitude(*altitude*)
Sets the maximum altitude of the drone

set_pitch(*pitch*)
Sets the drone pitch. Makes the drone going ahead or back.

set_roll(*roll*)
Sets the drone roll. Makes the drone going left or right.

set_yaw(*yaw*)
Set the drone angular speed

Paramètres **yaw** – The yaw angular speed

stop()
Stops all threads and coms

take_off()
Drone take off

test()
Some heading tests

update_drone()
Updates the drone state. To be used for moving the drone.

update_drone_state(*battery_level*, *yaw*)
Updates the drone state

Paramètres
- **battery_level** – The updated battery level percentage
- **yaw** – The current UAV heading

class drone.**DroneCommunication**(*ip_drone='192.168.2.1'*, *port_drone=5556*)
Handles communications with program.elf

close()
Closes the connexion

recv_from()
Waits for some data from the drone, with a 1 Kb buffer

send_at_cmd(*cmd*, *args_format_str=None*, **args_values*)
Sends an AT command to the drone. Automatically sets the AT command sequence number.

Paramètres
- **cmd** – command (e.g. "CONFIG")
- **args** – command arguments

Retourne The formatted AT command

drone.**float_to_bits**(*val*)
Converts the given float to its binary IEEE representation

Paramètres **val** – The float to be converted

Retourne The integer representation of the float

Pilotage complet

Main module to execute on the drone

Executes 4 threads in order to perfectly pilot the drone

class pilot.**Pilot**(*ip_phone='192.168.2.2'*, *port_phone=1234*)
Pilots the drone with N95 orders

Members : * base_time : Time difference between the drone (epoc) and the phone (actual) * drone : The uav controller * __currentlocation : Last known uav GPS location (private)

execute_order(*order*)
Executes the given order. May block during some time.

Paramètres **order** – Order to be executed

exit()
Avoid next thread loops

get_orders()
Retrieves orders from the phone

Retourne The orders list

loop_follow_orders()
Follows the orders caught from the N95

loop_liveness()
Ping the N95 sometimes

loop_update_location()
Asks the N95 for its last known position

loop_update_sensor()
Reads sensor values every second and stores them in a local cache. Tries to send the cache content to the N95 every 20 seconds.

request_get(*what*)
Requests information to the server

Paramètres **what** – What do we want ?

Retourne The server answer

request_put(*content*)
Push information to the server

Paramètres **content** – Data to be sent

Retourne The server answer

start()
Waits for an answer from the phone, then starts the communications

sync_time()
Asks the phone for its current time and store the difference with the current time on the drone.

pilot.**admissible_disparity**(*position_0*, *position_1*)
Tests if the drone is near its goal.

Retourne True if the drone is sufficiently near its goal to execute the next order

pilot.**compute_distance**(*latitude0*, *longitude0*, *latitude1*, *longitude1*)
Calculates the distance between 2 GPS points

Retourne The calculated distance, in meters

pilot.**compute_heading**(*latitude0*, *longitude0*, *latitude1*, *longitude1*)
Compute heading from a location(latitude0, longitude0) to another (latitude1, longitude1)

pilot.**modulate_value**(*coord*, *min_value*, *max_value*, *increment=3.141592653589793*)
Inc/Decrement by increment the coord value to be beetween min and max values

Retourne The new value

pilot.**normalize_coordinates**(*coords*)
Normalizes the given dictionnary.

Paramètres **coords** – A dictionnary with at least 'latitude' and 'longitude' entries

pilot.**normalize_coordinates_2**(*latitude*, *longitude*)
Normalizes the given coordinates and put'em in a dictionnary.

pilot.**sensor_read_geiger**(*length=8000*)
Reads geiger sensor values
TODO : test it...

2.3 Hommage à Youri Gagarine

2.3.1 Présentation du 12 avril 2011

Il s'agit de la présentation "officielle" des résultats du projet pour la matière M2M. La communication entre le drone et le téléphone n'était pas possible : nous ne savions pas encore qu'il n'était pas possible d'avoir une socket serveur non bloquante sous Symbian, ou au moins sur PyS60.

Les différentes tentatives de vol effectuées se sont alors basées sur les scripts de tests que nous avions précédemment appliqué pour comprendre le fonctionnement du drone.

À titre d'information, nous avions un vent d'Est en rafales lors de cette présentation.

Mode Columbia

La première tentative a été un échec cuisant : nous avions tenté de montrer que le drone pouvait voler avec le Nokia N95 sur son dos. Pour rappel, ce téléphone pèse environ 125g.

Le drone a pu décoller de quelques centimètres, mais n'a pas pu dépasser la hauteur des pissenlits qui se sont alors pris dans les pales et ont mis les moteurs en sécurité.

Mode Gilbert Montagné

Ensuite, nous avons supprimé le téléphone et lancé le script qui nous a servi à valider le changement de cap du drone.

Le drone a alors parfaitement décollé et a réussi à se stabiliser malgré le vent. Il a alors effectué correctement les ordres qui lui étaient indiqués par le script, mais sa rotation nécessite toujours un espace d'environ 4 mètres carrés.

Des vidéos de cet essai ont été tournées, mais ne sont toujours pas disponibles.

2.3.2 Essais du 16 avril 2011

Cet essai a été effectué après avoir réussi à faire communiquer le drone et le téléphone.

Nous avons ainsi pu détecter certains problèmes au niveau de la gestion de la commande *goto* et de la logique de navigation.

Nous avons refait les tentatives de rotation du drone, mais cette fois elles ont échoué : le drone se déplaçait en même temps qu'il tournait.

Des vidéos de cet essai ont été tournées, mais ne sont toujours pas disponibles.

À titre d'information, nous n'avions pas de vent lors de cette démonstration.

Index

S

T

U