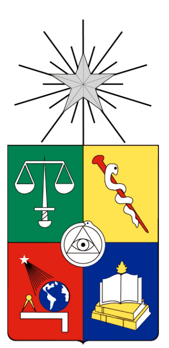


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

PLATAFORMA DE COMUNICACIÓN ENTRE LIVE ROBOT PROGRAMMING Y EL ROBOT AR.DRONE 2.0

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

CAROLINA MASSIEL HERNÁNDEZ PHILLIPS

PROFESOR GUÍA:
JOHAN FABRY

MIEMBROS DE LA COMISIÓN:
AIDAN HOGAN
JOCELYN SIMMONDS WAGEMANN

SANTIAGO DE CHILE

2016

RESUMEN DE LA MEMORIA PARA OPTAR AL
TÍTULO DE: Ingeniera Civil en Computación
POR: Carolina Massiel Hernández Phillips
FECHA: 25/10/2016
PROFESOR GUÍA: Johan Fabry

PLATAFORMA DE COMUNICACIÓN ENTRE LIVE ROBOT PROGRAMMING Y EL ROBOT AR.DRONE 2.0

La robótica es un área de estudio joven que presenta gran potencial de progreso. Esto incita a expertos de diversas áreas a contribuir en su desarrollo mediante proyectos de investigación y de ingeniería. En este escenario el estudiante Miguel Campusano y el profesor Johan Fabry, ambos integrantes del DCC de la Universidad de Chile, proponen un lenguaje de programación para robots denominado LRP. Este lenguaje es implementado en Pharo Smalltalk, un lenguaje de programación caracterizado por su fuerte orientación a objetos y su poderoso ambiente de desarrollo.

Utilizar LRP con un robot determinado requiere una aplicación estilo puente escrita en Pharo, que sirva como interfaz de comunicación entre el lenguaje y el robot. Previo a la realización de este trabajo, sólo dos puentes habían sido implementados. Estos permitieron hacer de pruebas con algunos robots. No obstante, para fines investigativos es muy deseable utilizar el lenguaje LRP con tantos robots como sea posible. El AR.Drone 2.0, de la empresa Parrot, es un cuadricóptero eléctrico no tripulado, controlado remotamente y que cuenta con diversos sensores. Este robot es un excelente caso de prueba para LRP, dada su gran diferencia con los robots utilizados anteriormente.

La solución presentada en este trabajo de memoria contempla la implementación de la infraestructura necesaria para hacer pruebas utilizando LRP con el AR.Drone y su validación mediante distintos programas de prueba implementados en LRP. Se ha creado una API capaz de comunicarse con el drone y la aplicación tipo puente requerida. Ambas aplicaciones fueron escritas en Pharo.

La API permite establecer una conexión con el drone, recibir los datos que publica y enviar comandos de control y configuración. El puente adapta una serie de métodos de la API y permite utilizarlos desde LRP. Provee además mecanismos intuitivos para que el desarrollador interactúe directamente con el drone durante una sesión de programación. La mayoría de las dificultades enfrentadas durante el desarrollo de la API se debieron a la indocumentación de características o requerimientos impuestos por el firmware del drone. Por otro lado, el desarrollo de esta herramienta permitió generar conocimiento técnico específico sobre el AR.Drone, relevante para trabajos posteriores que utilicen este robot.

Los programas de ejemplo creados en LRP evidenciaron que no todas las cualidades de la modalidad de programación en vivo pueden ser aprovechadas trabajando con un robot aéreo. Esto porque es difícil restringir límites para el movimiento del robot, por su limitado tiempo de autonomía y porque la interrupción de una sesión, producto de un choque por ejemplo, implica que el desarrollador deba suspender su trabajo para reposicionar y despegar el robot. Sin embargo, live programming permitió ajustar las variables asociadas a programas ya creados, lo que adquiere gran valor al momento de establecer distancias esperadas para el recorrido del robot. Finalmente, la sintaxis de LRP y sus diagramas de estado animados en vivo facilitaron enormemente el proceso de desarrollo.

A la memoria de Carmen Aguilera.

Agradecimientos

Son varias las personas a quienes debo la más sincera gratitud por el rol que jugaron en el proceso que me llevó a concebir este trabajo.

Agradezco en primer lugar a Johan Fabry, mi profesor guía, por su constante apoyo y dedicación. Por haber orientado mi trabajo y esfuerzo con sabiduría.

A mis amigos Miguel Campusano y Pablo Estefó, que tuvieron siempre la mejor disposición para responder a mis dudas y me ofrecieron palabras de aliento cada vez que acudí a ellos.

A Morph, que mejoró la calidad de mi redacción con mucha paciencia y gran talento, y siempre me ofreció ayuda desinteresadamente.

Agradezco especialmente a mi pololo Max Willembrinck por su apoyo incondicional, por darme la oportunidad de trabajar sólo en esta memoria durante meses y por haberme aconsejado cada vez que se lo pedí. Por haber compartido conmigo toda su experiencia e indudable capacidad técnica, guardando para mí siempre la mejor intención.

Para concluir quiero agradecer a mi familia. A mi Madre, quien desde que nací ha creído en mí ciegamente, de la forma más obstinada y también más honesta; motivándome a superar cada obstáculo que se ha presentado y a no desistir nunca en el camino del estudio. A mi hermana Lissete, a quien quiero tanto y con quien mejor nos entendemos; por su comprensión, compañía e incondicionalidad; por haber sido mi amiga desde siempre. Sin ella mi mundo sería más oscuro.

Finalmente a mi Padre, por haberme enseñado matemáticas en la primera infancia y haber plantado así la semilla que me trajo hasta aquí.

Tabla de contenido

I. Presentación del tema 1

1. Introducción 2
1.1. Objetivos . . . 3
1.2. Estructura del texto . . . 3

2. Contexto para cuadricópteros en la actualidad 6
2.1. Aplicaciones no militares para cuadricópteros . . . 6
2.2. Cuadricópteros en proyectos de investigación universitaria . . . 6
2.2.1. Trabajos de investigación relativos a estabilización del vuelo y realización de acrobacias . . . 7
2.2.2. Trabajos de investigación relativos a la navegación autónoma, al mapeo autónomo de terrenos y de estructuras 3D . . . 8
2.2.3. Trabajos de investigación relativos al trabajo cooperativo autónomo con otros robots . . . 9

3. Descripción técnica del AR.Drone 2.0 10
3.1. Estructura . . . 11
3.2. Movimientos . . . 12
3.3. Hardware . . . 12
3.4. Embedded Software (Firmware) . . . 13
3.5. Experimentos de uso . . . 15

4. Comunicación con el AR.Drone 2.0 desde un dispositivo cliente 16
4.1. Descripción de la red creada por el AR.Drone 2.0 . . . 16
4.2. Comunicación con el AR.Drone utilizando la Librería en C de Parrot . . . 17
4.2.1. Contenido y descripción de la Librería en C de Parrot . . . 18
4.2.2. Ciclo de vida de una aplicación creada utilizando la Librería en C de Parrot 21
4.2.3. Creación de una aplicación de ejemplo utilizando la Librería en C de Parrot 22
4.2.4. Resumen de funciones importantes de la Librería en C de Parrot para controlar el drone . . . 25
4.3. Comunicación con el AR.Drone sin utilizar la Librería en C de Parrot . . . 28
4.3.1. Consideraciones para envío de AT commands . . . 28
4.3.2. Consideraciones para la recepción de navdata . . . 29
4.3.3. Consideraciones para la recepción del stream de video . . . 30

5. Live Robot Programming 33
5.1. Diseño de LRP . . . 33
5.2. Sintaxis de LRP . . . 35
5.3. Puente entre LRP y la API del robot . . . 38
5.4. Pharo Smalltalk . . . 38

II. Descripción de la Solución 40

6. API en Pharo para comunicación con el AR.Drone 43
6.1. Diseño y arquitectura 44
6.2. Observaciones y conceptos relevantes 46
6.2.1. Observaciones sobre la configuración de los datos de navegación 46
6.2.2. Observaciones sobre el envío de comandos 47

7. Descomposición de la API en Pharo según componentes 49
7.1. Configuration variables 49
7.2. Adapters 49
7.3. Process Managers 50
7.3.1. Connection Manager 51
7.3.2. Navdata Manager 53
7.3.3. Command Manager 54
7.3.4. Configuration Manager 57
7.4. Drone Internal State 60
7.5. State Manager 63
7.6. Drone Motion State 63
7.7. Interfaces de usuario 64
7.7.1. Interfaz ARDUIMotionState 64
7.7.2. Interfaz ARDUIKeyboard 64

8. Descripción Funcional de la API en Pharo 68
8.1. Resumen de mensajes públicos 68
8.2. Excepciones asociadas a la API en Pharo (Exceptions de Smalltalk) 68
8.3. Anuncios asociados a la API en Pharo (Announcements de Smalltalk) 69
8.4. Indicaciones de uso y ejemplo demostrativo 69
8.5. Disponibilidad y requisitos de la API en Pharo 70

9. Puente entre LRP y la API en Pharo para el AR.Drone 72
9.1. Puentes implementados anteriormente 72
9.1.1. Puente entre LRP y PhaROS 72
9.1.2. Puente entre LRP y JetStorm 75
9.2. Diseño e implementación del puente entre LRP y la API en Pharo 77
9.2.1. Especificaciones sobre la implementación del puente 77
9.2.2. Métodos de la API en Pharo accesibles desde LRP 78
9.2.3. Diseño de la interfaz de usuario 79
9.2.4. Visualización de video 83
9.3. Programas de prueba 84
9.3.1. Square path 84
9.3.2. Tag detector 86
9.4. Disponibilidad 87

10. Validación y Lecciones Adquiridas **88**
10.1. Validación 88
10.1.1. Programa Shell tag follower 91
10.1.2. Programa Oriented roundel navigator 96
10.2. Lecciones Adquiridas 100

III. Conclusión y Trabajo Futuro **102**

11. Conclusión **103**

12. Trabajo Futuro **105**

Bibliografía **106**

Anexos **111**

A. Resumen de mensajes públicos de la API en Pharo **111**
A.1. Instancia de la clase ARDrone y conexión con el drone 111
A.2. Control de movimiento 112
A.3. Animación de LEDs 112
A.4. Consulta de datos de navegación 113
A.5. Consulta y modificación de la configuración interna del drone 114
A.6. Detección de tags 115
A.7. Configuración de la API y captura de excepciones 116

Índice de tablas

1. Resumen de los AT commands definidos por el firmware del AR.Drone 2.0 29
2. Estructura de un paquete de navdata 30
3. Extracto del archivo de configuración enviado por el drone 58
4. Controles del teclado para movimiento 66
5. Controles del teclado para Piruetas 66
6. Controles del teclado para animaciones LED 66

Índice de figuras

1. AR.Drone 2.0 carcasa para vuelos en interiores 10
2. AR.Drone 2.0 carcasa para vuelos en el exterior 11
3. Exhibición de partes del AR.Drone 2.0 11
4. Ejes de movimiento 12
5. Tags detectados por el AR.Drone 2.0. 14
6. Arquitectura de un sistema que utilice el SDK de Parrot. 18
7. Esquema de la Librería en C de Parrot 20
8. Ciclo de vida de una aplicación cliente que utiliza la librería en C de Parrot . . . 22
9. Editor de LRP 35
10. Diagrama de la máquina follower generado por LRP 36
11. Diagrama de componentes de la solución 42
12. Diagrama de componentes de la API 45
13. Interfaz de usuario ARDUIMotionState reflejando inclinación hacia abajo de la nariz del drone 64
14. Distribución de comandos del teclado 65
15. Captura de video publicado en YouTube que muestra un vuelo de prueba controlado con teclado. 67
16. Interfaz de usuario que muestra Publicadores y Subscriptores 74
17. Interfaz de usuario para crear nueva Subscripción 74
18. Imagen referencial del robot robuLAB 75
19. Imagen referencial del robot Ev3 Lego Mindstorms 77
20. Interfaz de usuario del puente entre LRP y la API del AR.Drone 79
21. Interfaces para el puente con LRP: Navdata 82
22. Interfaces para el puente con LRP: Configuration 83
23. Interfaces para el puente con LRP: Vision Detections 83
24. Diagrama de estados de la máquina anidada: flightpath 86
25. Diagrama de estados de la máquina: tagdetector 86
26. Demostración «Tag follower» versión simple, realizada en Colegio Saint George . 89
27. Demostración «Oriented Roundel visitor» realizada en Colegio Saint George . . . 90
28. Diagrama de estados de las máquinas tagFollower y followAlgo, versión simple . . 93
29. Diagrama de estados de las máquinas tagFollower y followAlgo, versión compleja 95
30. Diagrama de estados de las máquinas tagNavigator y lookAlgo 98

Capítulo I.

Presentación del tema

1. Introducción

El surgimiento de la robótica como área de estudio y como nicho de progreso tecnológico ofrece múltiples oportunidades a académicos e ingenieros para desarrollar trabajo de investigación que contribuya en su avance. No obstante, la ejecución de este tipo de proyectos conlleva dificultades prácticas importantes. Una de las más significativas surge de la brecha que separa los eventos de creación del código y la realización de pruebas en un simulador o en un robot real. Entre ambos eventos el programador debe compilar el programa y cargarlo al hardware del robot o al simulador; y en caso de detectar errores, necesitará rastrear el fallo mentalmente de vuelta a su código. Este proceso interrumpe el hilo de razonamiento referente al diseño y corrección del código, lo que mella la continuidad de la labor, eficiencia y probabilidad de éxito del programador.

En este contexto el estudiante de doctorado Miguel Campusano y el profesor Johan Fabry, ambos integrantes del Departamento de Ciencias de la Computación (DCC) de la Universidad de Chile, proponen un lenguaje de programación para robots caracterizado por permitir utilizar en ellos la modalidad Live Programming. En ésta el programador interactúa con un sistema que está siempre activo, de modo que la retroalimentación obtenida durante la experiencia de programación es máxima. Así, el nuevo lenguaje, denominado LRP (Live Robot Programming) [1], permite programar un robot que se encuentra ejecutando el código mientras está siendo escrito. LRP se encuentra implementado en el lenguaje de programación Pharo Smalltalk [2], caracterizado por su fuerte orientación a objetos y su poderoso ambiente de programación, enfocado en brindar retroalimentación al programador.

Para utilizar LRP con un robot determinado es necesario contar con una aplicación estilo puente (bridge) escrita en Pharo Smalltalk, que sirva como interfaz de comunicación entre el lenguaje y el robot. En la actualidad sólo dos aplicaciones puente han sido implementadas. Una de ellas posibilita la comunicación con el middleware ROS [3]. Gracias a ella pudieron realizarse pruebas usando los robots Turtlebot [4] y PR2 [5]. La segunda aplicación puente comunica con Ev3 Lego Mindstorms [6], y permitió utilizar el lenguaje con robots Lego. Para la investigación sobre LRP es muy deseable poder probar el lenguaje con tantos robots como sea posible, comprobando de esta manera su aplicabilidad dentro de un gran universo de prueba en el que varíen diversos factores, tales como el tipo de hardware involucrado, el contexto de desempeño, el tipo de tareas a ser efectuadas, etc.

El AR.Drone 2.0, de la empresa Parrot [7], es un cuadricóptero eléctrico no tripulado manejado por control remoto. Consiste en un cuerpo plástico soportado por una estructura de fibra de carbono, cuatro motores (cada uno de los cuales controla una hélice), varios sensores (de posición, inercia y orientación) y dos cámaras, una frontal y una inferior. El AR.Drone alberga en su interior un computador encargado de asistir en maniobras de vuelo complicadas, como despegar o aterrizar, y que además administra la recepción y envío de paquetes TCP/IP y UDP. Estos protocolos constituyen el medio de comunicación del drone con dispositivos externos [8].

Originalmente este cuadricóptero fue concebido como un sistema para juegos de realidad aumentada, sin embargo sus prestaciones actuales más frecuentes se relacionan con la captura de imágenes y con el vuelo radiocontrolado recreativo. Tales usos pueden llevarse a cabo utilizando aplicaciones provistas por Parrot. Pese a lo anterior, las capacidades del drone superan las de

un simple vehículo radiocontrolado, pues su equipamiento le permite ser programado para modificar su comportamiento de acuerdo a su entorno y estado interno, pudiendo llegar a hacerse completamente autónomo. De acuerdo a lo anterior, el AR.Drone cumple con la definición de robot.

El AR.Drone es un excelente caso de prueba para LRP, dada la gran diferencia que existe entre éste y los robots con los cuales el lenguaje ya ha sido probado. Estos han sido siempre terrestres y han contado con sensores totalmente distintos los del drone. Previo al inicio de este trabajo, no había sido posible efectuar pruebas con el cuadricóptero, debido a que no existía la infraestructura necesaria para que LRP pudiera comunicarse con él.

Para usuarios avanzados que busquen crear aplicaciones para el AR.Drone, Parrot ofrece un kit de desarrollo que incluye una API (Application Programming Interface) de código abierto, escrita en el lenguaje de programación C [9]. Ésta permite establecer una conexión inalámbrica con el drone, enviarle instrucciones de vuelo y configuración, y recibir sus señales de estado y el stream de video.

Múltiples trabajos de investigación se han realizado utilizando cuadricópteros, en particular drones. Entre éstos se encuentran tanto trabajos que utilizan la API provista por Parrot para comunicarse con el drone [10], como otros en que los investigadores han decidido programar su propia API [11], argumentando que requerían una herramienta que se adaptara mejor a sus necesidades.

1.1. Objetivos

El objetivo general del presente trabajo de memoria es contribuir al desarrollo de LRP, posibilitando la realización de pruebas de uso sobre el robot AR.Drone 2.0 y luego efectuando un set de pruebas básicas en las que se use LRP para controlar este robot.

Para cumplir con lo anterior será necesario llevar a cabo una serie de etapas que se resumen en la siguiente lista de objetivos:

- Diseñar e implementar en Pharo una API de capaz de comunicarse con el drone, ya que aquella provista por Parrot no puede ser usada desde Pharo, y en consecuencia tampoco desde LRP.
- Diseñar e implementar la aplicación tipo puente que comunique la API en Pharo con LRP.
- Crear la documentación completa de la solución para facilitar su uso por parte de terceros.
- Validar la solución, implementando varios programas en LRP que demuestren la factibilidad de utilizar este lenguaje en un robot con las características del drone.
- Publicar el trabajo en un repositorio de código abierto.

1.2. Estructura del texto

El presente documento se ha dividido en 3 partes.

La primera parte: «Presentación del tema» busca introducir a la materia, ofreciendo al lector todos los conceptos, herramientas y tecnologías relevantes para la comprensión de las partes posteriores. Además contextualiza el problema según la situación actual.

La segunda parte: «Descripción de la solución» está dedicada a explicar detalladamente el trabajo desarollado. Presenta su diseño y proceso de implementación, justificando las decisiones de diseño más importantes. Adicionalmente describe el proceso de validación efectuado y resume las principales lecciones aprendidas.

La tercera parte: «Conclusión y trabajo futuro» sintetiza el trabajo realizado y presenta una serie de ideas para su continuación.

A lo largo de todo el texto se han dispuesto recuadros celestes con encabezado «Lección aprendida». Éstos contienen ideas que fueron relevantes para el desarrollo de la solución y que no surgieron de la documentación oficial de Parrot para el AR.Drone 2.0. En cambio fueron en su mayoría obtenidas de la experiencia de desarrollo o investigación adicional.

Para facilitar el modo de explicar los programas de ejemplo, se ha optado por presentar su código dividido en secciones acompañadas inmediatamente por una explicación. Sin embargo todos los ejemplos para el AR.Drone 2.0 en LRP están disponibles en la siguiente URL:

`http://bit.ly/ARDroneLRPex`

Cada una de las partes de este documento se ha dividido en secciones. A continuación se ofrece una descripción general para cada sección:

Capítulo I: Presentación del tema

- Sección 1: Introducción

  Expone a grandes rasgos el problema que motiva el desarrollo del presente trabajo de memoria. Además presenta al lector la estructura del texto de modo que resulte más fácil de abordar.

- Sección 2: Contexto para cuadricópteros en la actualidad

  Presenta una visión general acerca de los usos civiles que tienen los cuadricópteros en la actualidad, haciendo hincapié en su participación en proyectos de investigación universitaria.

- Sección 3: Descripción técnica del AR.Drone 2.0

  Describe los componentes y capacidades del AR.Drone 2.0, tanto respecto a su hardware como a su software.

- Sección 4: Comunicación con el AR.Drone 2.0 desde un dispositivo cliente

  Expone en detalle el protocolo de comunicación del AR.Drone 2.0. Distingue entre dos alternativas: comunicarse utilizando la librería de Parrot o hacerlo sin ella.

- Sección 5: Live Robot Programming

  Ofrece una descripción del lenguaje de programación para robots LRP, de manera que el lector comprenda su rol en la programación de robots y, más adelante en el texto, sea capaz de entender los programas de ejemplo escritos en este lenguaje. Por otro lado explica los desafíos que surgen al intentar utilizar LRP con un nuevo robot.

Capítulo II: Descripción de la Solución

- Sección 6: API en Pharo para comunicación con el AR.Drone

  Inicia la descripción de la API desarrollada. Explica su diseño general e introduce consideraciones relevantes para abordar el resto de su descripción.

- Sección 7: Descomposición de la API en Pharo según componentes

  Describe detalladamente la API en Pharo, dividiéndola en componentes funcionales, consistentes en subsistemas responsables de una labor determinada. Justifica este diseño y explica su funcionamiento en base a la interrelación de sus partes.

- Sección 8: Descripción Funcional de la API en Pharo

  Contiene información suficiente para que un usuario de la API en Pharo pueda utilizarla sin necesidad de comprender en detalle los protocolos exigidos por el drone, ni el funcionamiento interno de la herramienta.

- Sección 9: Puente entre LRP y la API en Pharo para el AR.Drone

  Explica la función del puente en LRP, describiendo la estructura y funcionalidades de dos puentes que ya han sido creados para comunicar LRP con otras plataformas. A continuación ofrece una descripción específica del puente implementado para la API del drone, explicando su diseño y justificando cada una de sus partes.

- Sección 10: Validación y Lecciones Adquiridas

  Narrativa del proceso de validación más complejo al que se sometió la solución, que consistió en la realización de diversas demostraciones en el Encuentro de Robótica Educativa del Colegio Saint George. Incluye y explica el código LRP de las dos demostraciones más interesantes desde el punto de vista del uso de las herramientas desarrolladas. Luego se presenta un resumen de las lecciones más importantes que fueron obtenidas producto del desarrollo del trabajo de memoria.

Capítulo III: Conclusión y trabajo futuro

- Sección 11: Conclusión

  Resume el trabajo realizado y destaca los aprendizajes más importantes obtenidos.

- Sección 12: Trabajo futuro

  Propone ideas para trabajos que complementen o extiendan las herramientas desarrolladas durante este trabajo.

2. Contexto para cuadricópteros en la actualidad

Durante la década del 2000 la tecnología hizo posible manufacturar sensores de vuelo baratos y livianos: cámaras, acelerómetros, giroscopios y sonares. Esto permitió la creación de cuadricópteros pequeños, seguros y de bajo costo, cuyo alcance civil, comercial y militar se amplió a un ritmo exponencial. Del mismo modo lo hicieron las aplicaciones diseñadas para adaptar este tipo de robots a diversos usos. Se espera que esta tendencia de crecimiento no decline. En un anuncio de prensa publicado en agosto de 2015, Teal Group (Corporación estadounidense de investigación y publicación de información sobre la industria aeroespacial y de defensa) predijo que el gasto anual en producción de cuadricópteros en el mundo entero aumentará desde los 4 mil millones de dólares actuales a 14 mil millones de dólares en 2016, y sumará durante los próximos 10 años un total de 93 mil millones de dólares [12].

2.1. Aplicaciones no militares para cuadricópteros

Actualmente los cuadricópteros están siendo utilizados de diversas formas que van más allá de la defensa y el comercio. Su naturaleza aérea y su capacidad de transmitir datos inalámbricamente abren un mundo de oportunidades a desarrolladores interesados en crear aplicaciones para solucionar problemas de distinta índole. Buscando manifestar la variedad de aplicaciones existentes que tienen estos vehículos, a continuación se presentan algunos ejemplos, que abordan problemas de áreas heterogéneas del quehacer humano:

- Búsqueda y rescate de personas [13]
- Mapeo de terrenos de cultivo para optimizar la distribución de agua y fertilizantes [14]
- Captura de escenas aéreas para la industria cinematográfica, que son difíciles de conseguir mediante técnicas tradicionales [15]
- Fotografía deportiva [16]
- Estudios de preservación del medio ambiente y de la vida silvestre, tanto terrestre como marina [17][18][19]
- Recolección de datos científicos [20]
- Actividades relativas al periodismo [21]
- Transporte de insumos, por ejemplo medicinas, a lugares de difícil acceso [22]
- Comunicación humana post desastres [23]

Todas las aplicaciones anteriores han podido concretarse gracias a años de investigación realizada en universidades de distintas partes del mundo.

2.2. Cuadricópteros en proyectos de investigación universitaria

En los últimos años, universidades de variadas localidades han utilizado cuadricópteros, incluyendo el AR.Drone, como herramienta para investigar distintas áreas del conocimiento científico.

Tal preferencia se debe a que estos robots son económicos, pueden conseguirse en distintos tamaños y poseen una mecánica relativamente sencilla, haciéndolos fáciles de mantener. Además, muchos de ellos proveen mecanismos de comunicación directa con el sistema operativo de la máquina, e incluso ofrecen librerías que sirven de puente para efectuar dicha comunicación, hecho que facilita enormemente el desarrollo de aplicaciones por terceros. Un caso particular es el AR.Drone, cuyas caracterísitcas y software asociado serán presentados en los siguientes capítulos.

Trabajo de varias décadas dio como resultado el cuadricóptero típico actual, que tiene capacidad para estabilizar su vuelo y automatizar maniobras complejas como despegue y aterrizaje. Esto se logra gracias a algoritmos que utilizan información proveniente de los sensores, para controlar el vuelo en tiempo real. El cuadricóptero moderno es una máquina que, si bien se vale de la automatización como apoyo para la realización de maniobras complejas, aún depende del control de un usuario para efectuar tareas de navegación en el espacio y trabajos de mayor complejidad.

Los trabajos de investigación posteriores a 2010 relativos a cuadricópteros, centran sus esfuerzos en lograr la autonomía de la máquina, permitiéndole realizar tareas prescindiendo de un operador humano. Estas labores de investigación pueden ser clasificadas en su mayoría dentro de las siguientes tres áreas, a cada una de las cuales se ha dedicado un apartado inmediatamente a continuación:

- Algoritmos para estabilización de vuelo en condiciones adversas y realización de acrobacias. Sección 2.2.1
- Navegación autónoma, mapeo autónomo de terrenos y de estructuras 3D. Sección 2.2.2
- Trabajo cooperativo autónomo con otros robots (tanto aéreos como terrestres). Sección 2.2.3

2.2.1. Trabajos de investigación relativos a estabilización del vuelo y realización de acrobacias

Esta es una de las áreas que más llama la atención del público general. Prueba de esto es la creciente popularidad de videos publicados en YouTube por varios laboratorios de robótica, entre los que destaca GRASP de la Universidad de Pensilvania. En éstos se muestran cuadricópteros realizando autónomamente intrépidas acrobacias casi imposibles de lograr mediante control humano [24][25].

Por otro lado, si bien la manera convencional de controlar un cuadricóptero en la actualidad es mediante controles asistidos por software, los algoritmos tradicionales que se encargan de la estabilización del vehículo no responden adecuadamente bajo condiciones de fuertes ventiscas. Investigaciones de Sydney et al. [26] y otras de Alexis et al. [27] abordan el problema proponiendo algoritmos capaces de estabilizar al cuadricóptero en condiciones donde el viento es un factor determinante.

Ya que los cuadricópteros pueden ser utilizados como vehículos de carga, estabilizar su vuelo considerando que desde el vehículo cuelga una carga como peso libre, es un problema que requiere un nuevo modelo para el sistema y por ende investigación adicional. Sreenath y Kumar [28] han propuesto respuestas que consideran tanto sistemas de una sola máquina de la que pende una

carga, como otros compuestos por un equipo de máquinas que trabajan cooperativamente para elevar una única carga.

Buscando controlar el vuelo de los cuadricópteros bajo cualquier circunstancia, Hicham Khebbache et al. [29] abordan el problema de la estabilización en caso de graves fallas de hardware y proponen una solución (por ejemplo para el caso de pérdida o deformación de una hélice, falla de un motor, etc.).

2.2.2. Trabajos de investigación relativos a la navegación autónoma, al mapeo autónomo de terrenos y de estructuras 3D

Uno de los mayores desafíos para concretar la navegación autónoma es detectar la posición en el espacio del robot junto con las características de su entorno, para permitirle así calcular su trayectoria.

Krajńık et al. [30] proponen utilizar la información visual proveniente de las cámaras del robot para orientarlo en cierta dirección, y dejan la estimación de la distancia recorrida y por recorrer a la odometría. Este método de navegación se basa en una técnica de tipo «grabar y reproducir». La etapa de mapeo (grabado) consiste en condicir manualmente al drone por una ruta compuesta sólo por líneas rectas, la cual será navegada autónomamente en la etapa de reproducción. La ventaja de este método es que permite navegación tanto en ambientes cerrados como al aire libre.

Bills et al. [31] proponen una solución distinta que implementa navegación autónoma por ambientes cerrados utilizando los puntos de fuga (perspective cues) de la imagen capturada por la cámara frontal del drone. El método propuesto no requiere la construcción ni utilización de un modelo 3D del terreno, en cambio se basa exclusivamente en el cálculo de los puntos de fuga para direccionar correctamente el movimiento necesario para recorrer pasillos largos o escaleras. Este método es muy efectivo, pero se ve limitado a entornos cerrados que contengan largas líneas rectas y paralelas.

Por otro lado, Nguyen et al. [32] proponen utilizar la detección de esquinas (corner feature) en vez de los puntos de fuga para navegar autónomamente sin contar con un modelo del terreno previo. Este método ha resultado ser muy útil dado que presenta una menor cantidad de limitaciones respecto al tipo de entorno en que funciona.

Tanto la solución de Bills et al. como la de Nguyen et al. utilizan la técnica «grabar y reproducir».

Para las labores de mapeo en 3D, Fossel et al. [33] han propuesto una solución que utiliza diversos sensores que pueden asociarse a un cuadricóptero. En este ejemplo en particular, se utiliza un sonar para medir altura, en conjunto con un láser de barrido para modelar un entorno 3D desconocido. Los resultados son altamente satisfactorios tanto para entornos de prueba simulados como para entornos reales.

2.2.3. Trabajos de investigación relativos al trabajo cooperativo autónomo con otros robots

Se ha visto que los usos para un cuadricóptero trabajando en solitario son muy variados. No obstante, el número de posibles usos se incrementa notoriamente al considerarlo parte de un equipo de robots. Cuando el drone trabaja en equipo con otros robots, su capacidad de reemplazar al humano en tareas peligrosas y/o repetitivas se hace mucho más evidente.

Un factor fundamental para efectuar trabajo en equipo es la coordinación de las partes. Mellinger et al. [34] demuestran sus avances en esta materia utilizando un conjunto de mini-drones capaces de auto-organizarse, de modo de movilizarse como un escuadrón, superando obstáculos ordenadamente y cumpliendo con formaciones dinámicas definidas.

Una aplicación más práctica para el trabajo coordinado de robots, es presentado por Augugliaro et al. [35][36][37] y consiste en un equipo autónomo de cuadricópteros capaces de tejer un puente colgante de cuerda.

Dependiendo del tipo de trabajo requerido, puede que sea necesario integrar el funcionamiento de vehículos aéreos y terrestres en tareas coordinadas. Saska et al. [38][39][40] logran coordinar un robot terrestre y un drone. En su trabajo, el robot terrestre sirve de plataforma al aéreo, y se desplazan autónomamente en búsqueda de un objetivo visual. Cuando lo ha alcanzado, envía una señal al cuadricóptero posado en su superficie para que despegue y obtenga imágenes aéreas del objetivo encontrado.

3. Descripción técnica del AR.Drone 2.0

Para claridad del lector se ofrece a continuación un breve resumen sobre las especificaciones técnicas conocidas del AR.Drone 2.0. Es importante mencionar que muchos detalles del diseño de esta máquina permanecen ocultos porque éste es propiedad privada de la empresa Parrot. Un componente de especial interés cuyo diseño se desconoce es el software incrustado (embedded software) que el AR.Drone 2.0 corre dentro de un ambiente linux en su placa madre (mainboard). Este software es el encargado de leer los sensores, controlar motores, asistir maniobras de vuelo y comunicarse con dispositivos externos.

La información expuesta se ha dividido en secciones según la funcionalidad de los componentes del drone:

- Sección 3.1: "Estructura" describe partes físicas y cinéticas.
- Sección 3.2: "Movimientos" ilustra las maniobras básicas que realiza el drone durante el vuelo.
- Sección 3.3: "Hardware" muestra componentes electrónicos, incluyendo los sensores.
- Sección 3.4: "Embedded Software" describe los programas que corren en el mainboard del drone.

Figura 1: AR.Drone 2.0 carcasa para vuelos en interiores

Figura 2: AR.Drone 2.0 carcasa para vuelos en el exterior

Figura 3: Exhibición de partes del AR.Drone 2.0

3.1. Estructura

Como se muestra en la figura 3, el AR.Drone posee un esqueleto en forma de "X" fabricado de tubos de fibra de carbono, que sostiene cuerpo de espuma y piezas de plástico nylon. Su estructura ha sido diseñada para resguardo de las partes internas de la vibración y para resistir impactos. Incorpora además dos cubiertas: una sencilla para vuelo en exteriores (figura 2), y otra con protección para las hélices, para vuelo en interiores (figura 1). El peso del drone con batería y cubierta para exteriores es de 436 gramos.

El AR.Drone cuenta con cuatro motores con rotor interno, cada uno de los cuales propele una

hélice plástica. En estado de suspensión en el aire, estos motores funcionan en conjunto a 14,5W y 28.500rpm. Cada uno incorpora un microcontrolador reprogramable.

3.2. Movimientos

La figura 4 ilustra los 3 tipos de movimientos básicos que puede realizar el drone. La ejecución de estas maniobras es asistida por el software incrustado en el mainboard del drone, al igual que lo son el despegue (take-off) y el aterrizaje (landing).

Las maniobras básicas durante el vuelo son:

- **Yaw**: rotación en el eje **vertical**
- **Pitch**: rotación en el eje **lateral**
- **Roll**: rotación en el eje **longitudinal**

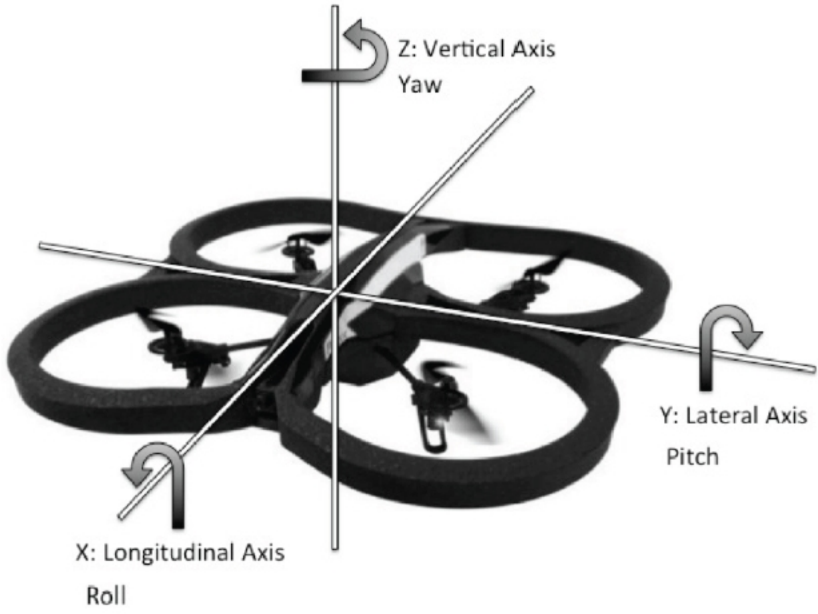


Figura 4: Ejes de movimiento

3.3. Hardware

Los siguientes dispositivos se integran a la placa madre:

- Procesador (CPU) de 1GHz y procesador digital de señales (DSP) de 800MHz.
- Chip de Memoria RAM de 1Gbit DDR2.
- Memoria persistente NAND flash de 32MB.
- Chip de red inalámbrico (WiFi) marca Atheros de 802.11b/g/n.
- Un puerto USB compatible con dispositivos USB de almacenamiento formateados en FAT32, cuya función es almacenar videos grabados durante el vuelo.

Todo el sistema es alimentado por batería de Polímero de Lítio (LiPo). Los sensores del drone son los siguientes:

- Acelerómetro de 3 ejes con precisión de ±50 mg
- Giroscopio de 3 ejes con precisión de 2000 g/segundo
- Sensor de presión con precisión de ±10 Pascal
- Sensor ultrasónico de 40Hz, rango de 0 a 6 metros.
- Cámara de video VGA CMOS, QVGA (320 x 240) a 60 fps. Se ubica en la sección inferior del cuerpo del drone, apuntando al piso
- Cámara de video HD CMOS, 720p (1280 x 720) a 30 fps. Lente de amplitud de 92°. Se ubica en el extremo frontal del cuerpo del drone, apuntando hacia adelante.

3.4. Embedded Software (Firmware)

Se refiere al software incrustado en el AR.Drone 2.0 que viene cargado de fábrica en su placa marde, responsable de la lectura de los sensores, automatización de maniobras de vuelo complejas e interacción con dispositivos externos mediante una conexión WiFi. En su configuración por defecto, el firmware del drone actúa como un access point que crea su propia red WiFi, a la cual se conectarán los dispositivos externos que deseen interactuar con el drone. Las características de esta red se describen en detalle en la sección 4.1.

Como se mencionó anteriormente, el conocimiento sobre el firmware del drone es muy limitado dada su característica de privado, no obstante algunas de sus características han sido inferidas por especialistas [41][42].

Cabe destacar no todos los cuadricópteros utilizan firmwares privados. Diversas comunidades y compañías ofrecen software de código abierto para controlar y comunicarse con cuadricópetos y UAVs, proveyendo completo acceso al hardware del vehículo, incluyendo motores, sensores y cámaras [43][44][45][46].

Capacidad de Detección de Tags

El firmware del AR.Drone incorpora un mecanismo de detección de tags en tiempo real, que utiliza el video proveniente de ambas cámaras y que, según el código de la API de Parrot para el AR.Drone, es capaz de reconocer 8 tipos de tags distintos: *Shell-Tag, Roundel, Black Roundel, Stripe, Cap, Shell-Tag V2, Tower Side* y *Oriented-Roundel*. Sin embargo en la documentación oficial de Parrot sólo se incluyen los tipos *Cap, Shell-Tag* y *Oriented-Roundel* (ver figura 5). Este trabajo se ha limitado a utilizar los tipos *Shell-Tag* y *Oriented-Roundel*.

El drone es capaz de detectar 4 tags a la vez, y para cada uno de ellos reconoce su tipo, posición que ocupa en la imagen de la cámara en que fue detectado, distancia a la que se encuentra de la cámara y, sólo para el caso de *Oriented-Roundel*, informa su orientación. La información acerca del resultado de detección de tags es enviada al cliente a través del stream de datos de navegación (Ver sección 4).

El tipo *Shell-Tag* consiste en un rectángulo compuesto por dos cuadrados naranjos en sus extremos y un cuadrado central que podrá ser amarillo, verde o azul. En la figura 5 se aprecian 3 ejemplares de Shell-Tag. Este tipo de tags fue diseñado para adherirse a la carcasa para vuelos exteriores del drone, logrando así que los drones se reconozcan unos a otros en sesiones de juegos de realidad aumentada. Las distintas opciones de colores existen para poder agrupar a los drones en equipos, distinguiendo entre aliados y enemigos. El drone podrá reconocer tags de este tipo solamente de un color determinado a la vez.

El tipo *Oriented-Roundel* consta de una línea negra acompañada de un círculo negro que contiene en su interior un rectángulo blanco. Cuando el drone reconoce este tag en su cámara inferior indicará que el ángulo de orientación es cero cuando su nariz se encuentre paralela a la línea negra y se haya orientado de modo que la línea negra quede a su derecha y el círculo a su izquierda. El valor del ángulo aumentará en sentido horario.

Tags de izquierda a derecha: Cap, Shell-Tag (3 ejemplares), Oriented Roundel.

Figura 5: Tags detectados por el AR.Drone 2.0.

En el drone, la capacidad de detección de tags automática y eficiente tiene gran potencial, ya que se permite programar comportamientos autónomos asociados al reconocimiento de tags estratégicamente posicionados, o bien que se encuentren en movimiento. En las secciones 10.1.2 y 10.1.1 se presentan programas de ejemplo en los que este potencial es aprovechado.

LECCIÓN APRENDIDA

Aunque la documentación oficial menciona que múltiples tags pueden ser detectados a la vez, lo cierto es que esto no aplica cuando múltiples Shell Tags son capturados una misma cámara, ya que sólo uno de ellos logra ser detectado.

3.5. Experimentos de uso

Luego de realizar numerosos vuelos de prueba, utilizando las aplicaciones oficiales de Parrot para controlar el drone, tanto desde un dispositivo móvil (android) como desde un PC (linux), se ha concluido que las maniobras de vuelo asistidas que provee el software incrustado son limitantes. Si bien estabilizan el vuelo, impiden la ejecución de nuevas piruetas arbitrarias, definidas por el usuario, y el vuelo por trayectorias que rodeen de cerca un obstáculo.

LECCIÓN APRENDIDA

El firmware del drone está programado para evitar que el drone se estrelle contra el piso, por ello el drone se eleva automáticamente cuando se acerca excesivamente a un obstáculo que esté debajo suyo.

4. Comunicación con el AR.Drone 2.0 desde un dispositivo cliente

Como se ha visto en la sección 3.4, el AR.Drone 2.0 tiene la capacidad de crear una red WiFi que sirve de canal para la comunicación bilateral entre el éste y un dispositivo externo cliente (host). Los protocolos de este proceso de comunicación son establecidos por el firmware del drone, tal como lo son la estructura de los mensajes intercambiados y su efecto sobre la máquina. Según este esquema existen cuatro tipos de mensajes, cada uno con un protocolo específico:

1. Envío de instrucciones de navegación y de modificación de la configuración
2. Recepción de la configuración actual del drone
3. Stream de video
4. Datos de navegación (navdata)

El primer tipo se origina en el host y tiene como destino el drone, los tres últimos tipos viajan en sentido contrario. Las instrucciones que se envían al drone corresponden a strings con una estructura definida y que incorporan el prefijo «AT*». Estos mensajes se denominan «AT commands» y son descritos en detalle en la sección 4.3.1.

Parrot provee a los desarrolladores que deseen programar aplicaciones para el AR.Drone 2.0, un Software Development Kit (SDK) [47] que incluye una librería de código abierto escrita en el lenguaje de programación C, la cual suministra APIs de alto nivel para comunicarse con el drone. Su utilización no es obligatoria.

La librería provista por Parrot es open source y en el SDK se incluye una especificación de los protocolos de comunicación exigidos. Esto permite programar desde cero una librería propia en cualquier lenguaje, que permita comunicarse con el AR.Drone mediante intercambio de paquetes vía WiFi.

4.1. Descripción de la red creada por el AR.Drone 2.0

Como se dijo anteriormente, en su configuración por defecto el AR.Drone 2.0 crea su propia red para comunicarse con dispositivos externos. En este modo, el drone ejecuta un servidor DHCP con las siguientes características:

- IP Drone: 192.168.1.1
- IP Host: 192.168.1.2 (sólo el dispositivo con esta IP puede controlar al drone)
    - Rango DHCP: 192.168.1.2-5
    - Gateway: 192.168.1.1
    - WiFi
        - Encriptación: Ninguna

- ESSID: adrone2_xxx (por defecto)

- Protocolos:
  - TCP/IP
  - UDP/IP
  - IPv4

Para usuarios avanzados, es posible configurar el drone vía telnet, asignándole una IP estática y creando una conexión con un punto de acceso determinado; haciendo esto, una aplicación en el cliente podría conectarse con múltiples drones a la vez. No obstante lo anterior, tal opción no es relevante para efectos de este trabajo, por lo cual a partir de este punto se considerará que la IP del AR.Drone será siempre fija, de valor 192.168.1.1.

4.2. Comunicación con el AR.Drone utilizando la Librería en C de Parrot

Parrot provee a sus usuarios un Software Development Kit (SDK) para facilitar la creación de aplicaciones que controlen el AR.Drone 2.0. Este SDK incluye:

- Un documento llamado «AR.Drone Developer Guide» que explica cómo utilizar el SDK y describe los protocolos de comunicación con el drone.
- La denominada «AR.Drone 2.0 Library», que es un conjunto de APIs escritas en C, necesarias para comunicarse con el drone y configurarlo.
- La librería «AR.Drone 2.0 Tool», que forma parte de la «AR.Drone 2.0 Library», y que provee un cliente totalmente funcional con el cual los programadores simplemente necesitarán insertar directamente el código de su propia aplicación.
- La librería «AR.Drone 2.0 Control Engine», que es una interfaz para controlar el drone fácilmente desde dispositivos iOS. No se utilizará en este trabajo.

Para efectos de este trabajo de memoria se denominará «librería en C de Parrot» a la «AR.Drone 2.0 Library». Esta librería es de código abierto y, como ya se dijo, contiene APIs de alto nivel que posibilitan la comunicación con el AR.Drone 2.0[47].

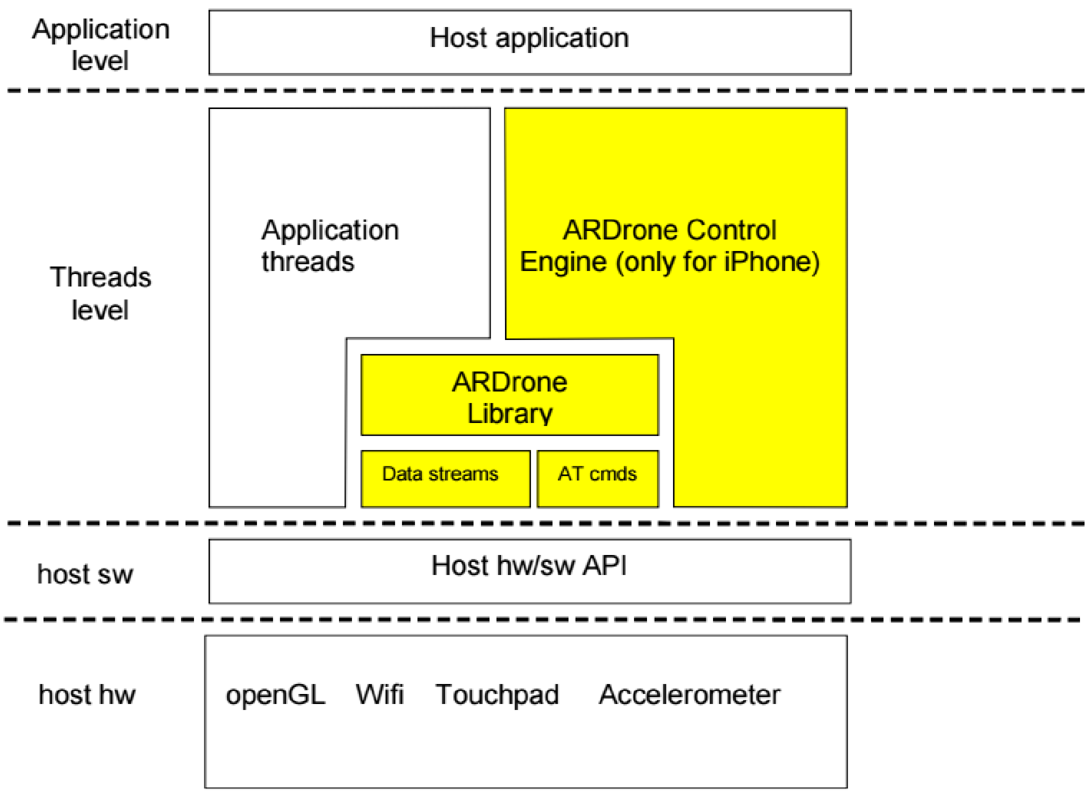
La librería en C de Parrot puede ser utilizada desde los siguientes dispositivos:

- Cualquier PC que corra Linux y tenga tarjeta de red WiFi
- Dispositivos Android
- Dispositivos Apple iOS

Esta librería no permite interacción directa con el hardware del AR.Drone (cámaras, sensores, motores, etc.), ya que sólo se comunica a través de su firmware.

En la figura 6 se muestra la arquitectura en capas que debiese tener una aplicación que utilice las librerías del SDK de Parrot para comunicarse con el drone. Se destacan en amarillo las componentes del SDK. Entre ellas se encuentran los AT commands que se envían al drone

(AT cmds), pues son generados por la librería; y los streams de datos provenientes desde el drone (Data streams), que son recepcionados y decodificados por la librería también. Es importante ver que la aplicación cliente («Host application») interactúa con la AR.Drone 2.0 Library sólo a través de threads, lo cuales son creados utilizando la librería «AR.Drone 2.0 Tool».



Fuente: AR.Drone Developer Guide

Figura 6: Arquitectura de un sistema que utilice el SDK de Parrot.

4.2.1. Contenido y descripción de la Librería en C de Parrot

La librería en C de Parrot se encuentra dividida en módulos según funcionalidad. A continuación se los enlista junto con su descripción.

- **SOFT**: código específico para comunicarse con el drone, incluye:
  - **COMMON**: contiene headers (.h) con las estructuras (structs) utilizadas en la librería.
  - **ardrone_tool**: conjunto de herramientas para controlar el drone fácilmente. Contiene por ejemplo un thread para enviar AT commands, un thread que recepciona navdata, un pipeline de video listo para utilizar y una función `main` lista para utilizar.
  - **utils**: conjunto de utilidades para escribir aplicaciones para el AR.Drone.
- **VIDEO PROCESSING**: esta es una categoría creada dentro del contexto de esta memoria con fines explicativos. Agrupa todos los módulos encargados de procesamiento de video e imágenes.

    - **VLIB**: librería de procesamiento de video del AR.Drone 1.0. Contiene funciones para recibir y decodificar el stream de video.
    - **FFMPEG**: compendia lo esencial de una librería FFMPEG (librería open source para grabar, convertir y hacer streaming de video y audio [48]), junto con scripts para las aplicaciones del AR.Drone 2.0.
    - **ITTIAM**: librería precompilada, altamente optimizada para aplicaciones iOS y Android (CPUs ARMv7 NEON).
- **VPSDK**: módulo compuesto por librerías de propósito general, incluye:
    - **VPSTAGES**: secciones de procesamiento de video que pueden ser ensambladas para construir un pipeline de video.
    - **VPOS**: utilidades que encapsulan tareas a nivel de sistema multiplataforma (Windows, Linux, Parrot), tales como asignación de memoria, administración de threads, etc.
    - **VPCOM**: utilidades que encapsulan funciones de comunicación multiplataforma (WiFi, Bluetooth, etc.).
    - **VPAPI**: utilidades para controlar pipelines de video y threads.

De todos los módulos nombrados, aquel de mayor relevancia para el programador es **ardrone_tool**. En seguida se detalla su contenido:

- **ardrone_tool.c**: contiene la función `ardrone_tool_main` lista para ser utilizada. Ésta inicializa la red WiFi y todas las comunicaciones con el drone.
- **UI**: contiene código listo para utilizarse, para el uso de joystick.
- **AT**: contiene todas las funciones que se utilizan para controlar el drone. La mayoría de ellas se corresponde directamente con un AT command, el cual es construido automáticamente con la sintaxis correcta y con un número de secuencia adecuado, y luego es enviado al thread de administración AT.
- **NAVDATA**: contiene un sistema listo para usar que recibe y decodifica los datos de navegación provenientes del drone (navdata).
- **ACADEMY**: contiene un sistema listo para usar, de carga y descarga de contenido multimedia desde y hacia el sistema Parrot Drone Academy [49].
- **VIDEO**: contiene funciones de alto nivel relacionadas con recepción, decodificación y grabación de video, tanto para AR.Drone 1.0 como para su versión 2.0. Utiliza las librerías **VLIB** y **FFMPEG**.
- **CONTROL**: contiene una herramienta de configuración para el AR.Drone lista para usar.

La siguiente figura esquematiza el contenido de la librería.

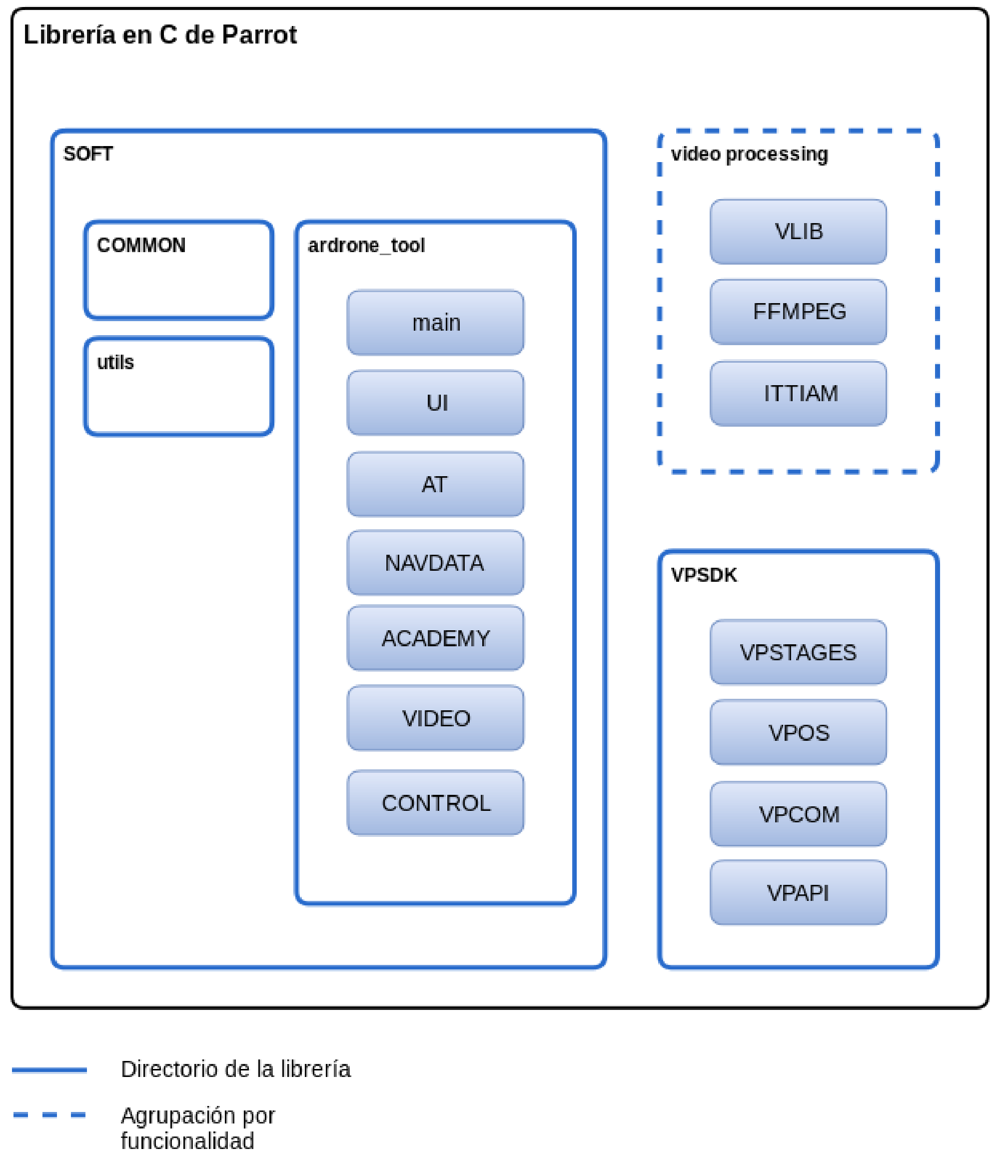


Figura 7: Esquema de la Librería en C de Parrot

El módulo **ardrone_tool** implementa eficientemente los siguientes hilos de ejecución (threads):

- **Thread administrador de AT commands**, que recopila los comandos enviados por todos los otros threads, les agrega un número de secuencia adecuado y los envía de manera ordenada.
- **Thread administrador de navdata**: automáticamente recibe el stream de datos navdata, lo decodifica y, a través de una función call-back, provee a la aplicación host con datos de navegación listos para ser usados.
- **Thread administrador de video** (sólo para AR.Drone 2.0), se encarga de controlar la grabación del stream de video HD, y del encapsulamiento en formato .mp4 y .mov.
- **Thread de control**: recibe peticiones de otros threads que desean recibir mensajes confiables desde el drone. Además se encarga de chequear los mensajes de confirmación del

drone.

- **Conjunto de threads para el AR.Drone Academy**, los cuales se encargan de obtener imágenes y subirlas al servidor de AR.Drone Academy.

Cada uno de estos threads se comunica con el drone al momento de su creación. Para esto utilizan las utilidades contenidas en VPCOM, que se encargan de reconectar con el drone cada vez que sea necesario. Estos threads y las inicializaciones consecuentemente requeridas, son creados y administrados por la función `ardrone_tool_main` (en el módulo **ardrone_tool**)

En resumen todo lo que el programador necesita hacer es completar con código propio las funciones callback llamadas por los threads correspondientes. No obstante queda a cargo del programador el trabajo de protección de secciones críticas que pudieran provocar errores de concurrencia

Dentro del SDK suministrado por Parrot se encuentran varios ejemplos de código abierto, en los que se usa la librería en distintas plataformas (Windows, Linux, iOS, Android). Se recomienda al programador utilizar estos ejemplos como plantilla para el desarrollo de aplicaciones propias.

Dentro de la librería se especifica un módulo llamado **ARDrone Control Engine** creado para ser utilizado en dispositivos iOS. No se profundizará en este módulo dado que su contenido es accesorio y no aporta nada fundamentalmente nuevo a la librería.

4.2.2. Ciclo de vida de una aplicación creada utilizando la Librería en C de Parrot

Para poder crear una aplicación que utilice la librería de Parrot para comunicarse con el drone, es requisito entender el ciclo de vida impuesto por esta librería. Se entenderá como ciclo de vida a la mínima cantidad de eventos que ocurran entre el lanzamiento y la finalización de la aplicación cliente. La figura 8 diagrama el ciclo de vida de la aplicación cliente y su relación con los componentes de la librería de Parrot. En ésta es claro apreciar el comportamiento asociado a la función main() (ardrone_tool_main), que cuando es invocada inicia el ciclo de vida de la aplicación, que en su interior contempla el llamado tanto de funciones internas de la propia librería (región verde de la figura), como de funciones especiales que fueron redefinidas por el programador al interior de su aplicación (región celeste). Estas funciones especiales tienen nombres con sufijo 'custom' (a excepción de `ardrone_tool_exit)` y corresponden a *weak functions* declaradas dentro de la librería en C de Parrot. Se entenderá por *weak function* a una función que cuenta con una definición por defecto al interior de la librería, pero que puede ser redefinida por el desarrollador dentro de su aplicación cliente. Tal como se aprecia en la figura 8 cada *weak function* es invocada en un momento específico del ciclo de vida de la aplicación. En la librería de Parrot estas funciones son en total 5:

- `ardrone_tool_init_custom`
- `ardrone_tool_update_custom`
- `ardrone_tool_display_custom`
- `ardrone_tool_shutdown_custom`

- `ardrone_tool_exit`

A grandes rasgos la comunicación con el drone siempre pasará por las siguientes tres etapas:

1. Inicialización: se establece la comunicación con el drone y se ejecutan los procesos necesarios para que el cliente inicialice procesos, threads o variables.
2. Event Loop: corresponde al período durante el cual el drone se encuentra ya conectado al cliente, reporta su estado y recibe instrucciones. Durante toda esta estapa se ejecutan los threads definidos por el cliente, los que se encuentran a la escucha de eventos gatillados por el drone. Esta etapa finaliza cuando se recibe la señal de finalización de la comunicación (ya sea por pérdida de conexión o término intencional de ésta).
3. Finalización: se finalizan los threads de la aplicación y se termina la comunicación con el drone.

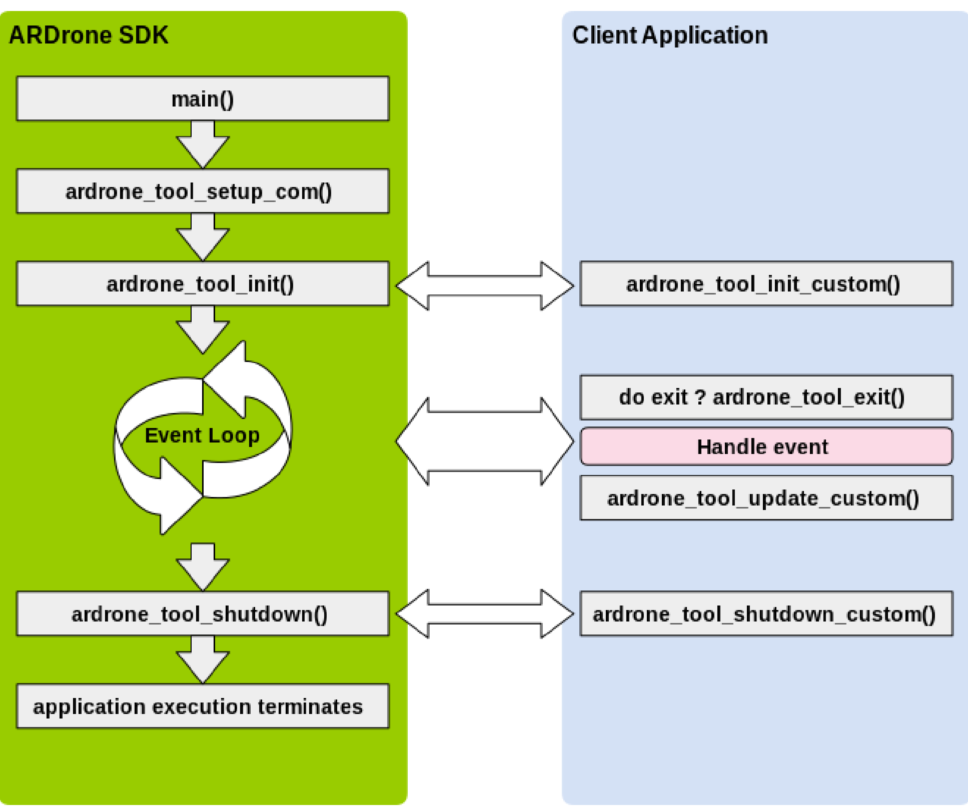


Figura 8: Ciclo de vida de una aplicación cliente que utiliza la librería en C de Parrot

4.2.3. Creación de una aplicación de ejemplo utilizando la Librería en C de Parrot

Para crear una aplicación utilizando esta librería, Parrot recomienda copiar la aplicación SDK Demo (incluida en el SDK) y modificarla del siguiente modo:

- Crear un nuevo thread y agregarlo a la estructura `THREAD_TABLE` para así permitirle enviar y recibir comandos desde el drone de manera independiente.
- Llamar la función `ardrone_tool_main` desde la aplicación. Este llamado dará inicio al ciclo de vida de la aplicación. En la figura 8 corresponde a la función main() de la región verde.

- Crear los navdata handlers necesarios y agregarlos a la `ardrone_navdata_handler_table`.
- Redefinir las *weak functions* deseadas, en caso de querer ejecutar código en momentos específicos del ciclo de vida de la aplicación.

A continuación se presenta un ejemplo en base a SDK Demo. Todo el código relevante se ha dispuesto en una clase: `ardrone_testing_tool` en el archivo `ardrone_testing_tool.c`. En este ejemplo el cliente hace despegar al drone del piso y luego de 3 segundos lo aterriza.

En los siguientes párrafos son explicadas las secciones más importantes del código de la clase `ardrone_testing_tool`.

```
01 /* ardrone_testing_tool.c */
02 #include <ardrone_api.h>
03 #include <ardrone_tool/UI/ardrone_input.h>
04
05 //VP_SDK
06 #include <VP_Api/vp_api_thread_helper.h>
07
08 //For sleep function
09 #include <unistd.h>
```

En el código anterior se hace include a las siguientes clases de ARDroneSDK:

- `ardrone_api.h` provee acceso a la función `ardrone_tool_main()`, que debe ser llamada dentro del método main de esta clase, ya que inicializa la conexión WiFi y todas las demás comunicaciones con el drone. Define además las *weak functions*, algunas de las cuales serán reimplementadas en líneas posteriores.
- `ardrone_input.h` otorga acceso a funciones de ARDroneLib que controlan el vuelo del drone. Una de ellas se utilizará más adelante en este ejemplo. Un resumen de las funciones disponibles para controlar el drone es incluido en la sección 4.2.4.
- `vp_api_thread_helper.h` permite acceder a la tabla de threads, mediante el uso de macros, y registrar allí la función que define el comportamiento del robot, en este caso despegue y aterrizaje.
- `unistd.h` se incluye para poder utilizar la función sleep dentro del thread que controla el despegue y aterrizaje del drone, de este modo depués de enviar la instrucción de despegue, el thread esperará 3 segundos antes de enviar la instrucción de aterrizaje.

```
10 PROTO_THREAD_ROUTINE(takeoff_land, data);
11 DEFINE_THREAD_ROUTINE(takeoff_land, data)
12 {
13     PRINT("\nTaking Off ...");
14     ardrone_tool_set_ui_pad_start(1);
15     sleep(10);
16     PRINT("\nLanding ...");
17     ardrone_tool_set_ui_pad_start(0);
18
19     return C_OK;
20 }
```

`PROTO_THREAD_ROUTINE` es una macro que se encarga de inicializar la función `takeoff_land` con su argumento data (que en este ejemplo no es utilizado) como un componente de la aplicación. A continuación `DEFINE_THREAD_ROUTINE`, que también es una macro, define el comportamiento asociado a la función `take_off`, que más tarde se ejecutará como un thread independiente dentro de la aplicación. La función `ardrone_tool_set_ui_pad_start(arg)` envía al drone el comando correspondiente a despegar si recibe como argumento el número `1`, o para aterrizar si recibe un `0`. Es importante notar que el comportamiento asociado al thread `takeoff_land` tiene una duración determinada y por tanto no se ejecutará durante todo el tiempo que la conexión con el drone se mantenga. Tal comportamiento es atípico en la implementación de threads que interactúen con el drone, ya que por lo general éstos entran en un ciclo que dura mientras la conexión con el drone no haya sido terminada

.

```
21 int main(int argc, char** argv)
22 {
23     return ardrone_tool_main(argc, argv);
24 }
```

En las líneas anteriores se define el `main` de la clase, dentro suyo se llama la función `ardrone_tool_main`, que como ya se ha dicho, inicializa todas las conexiones necesarias para interactuar con el drone.

```
25 /* The delegate object calls this method during
26 initialization of an ARDrone application */
27 C_RESULT ardrone_tool_init_custom(void)
28 {
29   /* Start all threads of your application */
30   START_THREAD( takeoff_land, NULL );
31   return C_OK;
32 }
```

Tal como se comentó al inicio de esta sección, la librería ARDroneLib, específicamente en el archivo `ardrone_tool.h`, se definen 5 *weak functions*, que pueden ser reimplementadas por el usuario para ejecutar labores determinadas en momentos precisos del ciclo de vida de la aplicación. Una de estas funciones es reimplementada en las líneas anteriores: `ardrone_tool_init_custom`.

Será ejecutada cuando se inicialice la conexión con el drone y se encargará de iniciar el thread `takeoff_land`.

```
33 /* The delegate object calls this method when the event loop exit */
34 C_RESULT ardrone_tool_shutdown_custom(void)
35 {
36   /* Relinquish all threads of your application */
37   JOIN_THREAD( takeoff_land );
38   return C_OK;
39 }
40
41 /* The event loop calls this method for the exit condition */
42 bool_t ardrone_tool_exit()
43 {
44   return exit_ihm_program == 0;
45 }
46
47 C_RESULT signal_exit()
48 {
49   exit_ihm_program = 0;
50   return C_OK;
51 }
```

`ardrone_tool_shutdown_custom` y `ardrone_tool_exit` también son implementaciones de *weak functions* definidas en `ardrone_tool.h`.

```
52 /* Implementing thread table in which you add routines of
53 your application and those provided by the SDK */
54 BEGIN_THREAD_TABLE
55   THREAD_TABLE_ENTRY( takeoff_land, 20 )
56 END_THREAD_TABLE
```

Finalmente se agrega el thread requerido a la `THREAD_TABLE` mediante el uso de las macros `BEGIN_THREAD_TABLE`, `THREAD_TABLE_ENTRY` y `END_THREAD_TABLE`.

4.2.4. Resumen de funciones importantes de la Librería en C de Parrot para controlar el drone

Las funciones más importantes que sirven para controlar los movimientos del drone son sólo tres: una para despegue/aterrizaje, una para controlar las inclinaciones y velocidades del drone, y una para hacer que el drone entre o salga del estado de emergencia.

El estado de emergencia es un estado definido por el firmware del drone. Cuando el drone entra al estado de emergencia sus motores se detienen inmediatamente. El drone puede entrar en este estado por petición del cliente o bien su firmware puede hacerlo automáicamente en respuesta a un suceso externo (por ejemplo bloqueo de hélices, inclinación excesiva en roll o pitch, entre otros).

La única manera de mover al drone en el aire es mediante el control de sus inclinaciones (pitch y roll), de su velocidad angular (yaw) y de su velocidad vertical (gaz). Los valores para tales variables no son binarios, todos ellos pueden ser ajustados sutilmente para obtener el movimiento deseado. La documentación del SDK de Parrot denomina «progressive commands» a aquel tipo de AT commands que altera los valores de pitch, roll, yaw y gaz del drone.

En versiones anteriores de la API el número de funciones disponibles para este fin era muy superior, pero muchas de éstas han sido deprecadas. Aquellas vigentes a la fecha son:

`ardrone_tool_set_ui_pad_select`

Resumen: Envío de señal de emergencia / Envío de señal de recuperación desde emergencia.

Argumentos: `int value` : flag de emergencia.

Descripción: Cuando el drone está volando o en estado listo para volar, este comando se debe usar con $value = 1$ para comenzar a enviar la señal de emergencia al drone, cuyo efecto será detener los motores y caer.

Cuando el drone se encuentra en estado de emergencia, este comando debe usarse con $value = 1$ para hacer que el drone intente volver al estado normal.

Una vez que se ha comenzado a enviar la señal de emergencia, el estado del drone debe monitorearse mediante la *navdata*, y se debe esperar hasta que haya cambiado de estado efectivamente. Cuando esto ocurra, este comando debe llamarse con $value = 0$ para cesar el envío de la señal de emergencia.

`ardrone_tool_set_ui_pad_start`

Resumen: Despegue / Aterrizaje.

Argumentos: `int value` : flag de despegue.

Descripción: Hace que el drone despegue ($value = 1$) o aterrice ($value = 0$).

Si durante una sesión de vuelo el drone entra en estado de emergencia, el cliente deberá llamar esta función con argumento 0 para evitar que el drone intente despegar apenas haya salido de este estado.

`ardrone_at_set_progress_cmd`

Resumen: Mueve al drone.

Argumentos:

`int flags` : flag que habilita el uso de progressive commands.
`float phi` : ángulo izquierda / derecha $\in [-1{,}0; +1{,}0]$.
`float theta` : ángulo frontal / trasero $\in [-1{,}0; +1{,}0]$.
`float gaz` : velocidad vertical $\in [-1{,}0; +1{,}0]$.
`float yaw` : velocidad angular $\in [-1{,}0; +1{,}0]$.

Descripción:

Esta función hace que el drone se mueva por el aire. Son ignorados por completo por el drone si los recibe mientras se encuentre detenido en el piso.

El drone es controlado mediante el envío de cuatro parámetros:

- un ángulo de inclinación izquierda / derecha, donde 0 es el plano horizontal y los valores negativos corresponden a inclinación hacia la izquierda (*roll*).
- un ángulo de inclinación frontal / trasero, donde 0 es el plano horizontal y los valores negativos corresponden a inclinación frontal (*pitch*).
- una velocidad vertical (*gaz*).
- una velocidad angular (*yaw*).

Para permitir al usuario mayor control, los parámetros entregados mediante esta función no son valores absolutos, sino que corresponden a un porcentaje de un valor máximo para cada tipo de movimiento, especificado mediante la configuración del drone. En consecuencia, los parámetros enviados deberán ser siempre valores tipo *float* $\in [-1{,}0; +1{,}0]$.

4.3. Comunicación con el AR.Drone sin utilizar la Librería en C de Parrot

Parrot especifica en su SDK el protocolo utilizado para intercambiar mensajes con el AR.Drone 2.0. Esto permite crear librerías que posibiliten la comunicación con el drone, desde cero y en distintos lenguajes. Evitando así el uso de la librería oficial de Parrot.

Múltiples librerías para controlar el AR.Drone 2.0 han sido creadas y publicadas en formato open source. Las encontramos en diversos lenguajes: Java [50], C++ [51], Scala [52], Python [53], Ruby [54], Javscript [55]. Uno de los argumentos que ciertos desarrolladores entregan para justificar su trabajo es que la librería oficial de Parrot es demasiado compleja para sus propósitos. Otro argumento encontrado se refiere a la preferencia que tienen la mayoría de los desarrolladores por lenguajes de programación de más alto nivel que C. Por último, se ha concluido que un requerimiento importante para una API de este estilo es que permita el desarrollo de prototipos rápidamente.

Una librería completa para comunicar al cliente con el drone deberá ser capaz de lidiar con los siguientes requerimientos:

- Establecer una conexión WiFi exitosa con el drone.
- Enviar AT commands al drone (para configuración y maniobras de vuelo).
- Recibir navdata proveniente desde el drone.
- Recibir y procesar el stream de video proveniente desde el drone.

4.3.1. Consideraciones para envío de AT commands

Los AT commands son strings enviados al drone para controlar sus acciones. El firmware del drone está programado para recibir estos paquetes por protocolo UDP, a través del puerto 5556 (se exige el uso del mismo puerto de envío y recepción durante la conexión UDP/IP).

Para lograr una experiencia de control fluida, Parrot recomienda enviar AT commands cada 30 milisegundos. Por otro lado para evitar que el drone considere la conexión perdida, durante toda la sesión de conexión el cliente deberá enviar comandos con una separación de menos de dos segundos entre sí.

Sintaxis de los AT commands

Estos strings deben construirse en con caracteres ASCII de 8 bits, con un caracter de retorno (valor de byte $0D_{(16)}$), denotado <**CR**> como delimitador de línea.

Un comando consistirá en los tres caracteres «**AT***» seguido por el nombre de un comando, un signo igual, un número de secuencia y opcionalmente una lista de argumentos separados por comas, cuyo significado dependerá del comando.

Un paquete UDP podrá contener uno o más comandos separados por caracteres de fin de línea (valor de byte $0A_{(16)}$), no obstante un comando no podrá ser dividido en diferentes paquetes UDP.

Por ejemplo, el siguiente es un comando AT válido:

AT*PCMD_MAG=21625,1,0,0,0,0,0,0<**CR**>

Consideraciones especiales deben ser tomadas al momento de codificar enteros con signo, floats y strings. Todos estos detalles se especifican en la Guía Para el Desarrollador incluida en el SDK de Parrot.

Secuencia de envío de los AT commands

Para evitar que el AR.Drone procese comandos antiguos, a cada comando se le asigna un número de secuencia, el que se incorpora en su string como el primer entero después del signo "igual". El drone no ejecutará comandos cuyo número de secuencia sea inferior al número del último comando válido recibido. Además, en caso de perder la conexión en el puerto UDP-5556 o de recibir un AT command con número de secuencia 1, el drone reiniciará el contador de números de secuencia a 1.

En fin, para comunicarse exitosamente con el drone se deberá enviar siempre el primer comando con número de secuencia igual a 1 y enviar siempre comandos con números de secuencia ascendentes.

En la Tabla 1 se muestra un resumen de los AT commands.

AT command	Argumentos	Descripción
AT*REF	input	Despegue/Aterrizaje/Emergencia
AT*PCMD	flag, roll, pitch, gaz, yaw	Mueve al drone
AT*FTRIM	-	Establece la referencia para el plano horizontal (el drone debe estar sobre el suelo)
AT*CONFIG	key, value	Modifica la configuración del drone
AT*CONFIG_IDS	session, user, application ids	Identificadores para los comandos AT*CONFIG
AT*COMWDG	-	Reestablece el watchdog de comunicaciones del drone
AT*CALIB	device number	Indica al drone que calibre su magnetómetro (debe encontrarse volando)

Tabla 1: Resumen de los AT commands definidos por el firmware del AR.Drone 2.0

4.3.2. Consideraciones para la recepción de navdata

Los datos de navegación (navdata) informan periódicamente al cliente sobre el estado del drone (altitud, ángulos de inclinación, cámara, velocidad, etc.). Son enviados por el drone a través del puerto UDP-5554.

Sintaxis de navdata

La información es almacenada dentro de los paquetes en formato binario y consiste en varias secciones de bloques de datos llamados “options”. Cada option contiene un header de 2 bytes que lo caracteriza, un entero de 16 bits que indica el tamaño del bloque, y múltiple información guardada en forma de enteros o floats de 32 bits o como arreglos. Todos estos datos son almacenados en formato *little-endian* [56].

La siguiente tabla muestra la estructura típica de un paquete de navdata.

Header 0x55667788	Drone state	Sequence number	Vision flag	Option1			...	Checksumblock		
				id	size	data	...	cks id	size	cks data
32-bit int	32-bit int	32-bit int	32-bit int	16-bit int	16-bit int	... ...	... ...	16-bit int	16-bit int	32-bit int

Tabla 2: Estructura de un paquete de navdata

Sincronización entre cliente y drone

El cliente podrá verificar que el número de secuencia (presente en el header de los paquetes de navdata) es creciente. En caso de que el drone pierda conexión con el cliente, el primero reiniciará el número de secuencia de estos paquetes y entrará en un estado de advertencia. Será responsabilidad del cliente sacar al drone de este estado mediante el envío de AT commands específicos.

4.3.3. Consideraciones para la recepción del stream de video

A diferencia del AR.Drone 1.0, la versión 2.0 utiliza códecs de video estándar y encapsulamiento a medida para el envío del stream de video a través de la red. El AR.Drone 2.0 envía dos tipos de stream de video: un stream de video en vivo y uno de grabación.

Códecs de video

El AR.Drone 2.0 utiliza el perfil básico del códec H264 (MPEG4.10 AVC) para transmisión y grabación de video de alta calidad.

Los siguientes valores pueden ser adaptados para el stream de video en vivo:

- FPS : Entre 15 y 30
- Bitrate : Entre 250kbps y 4Mbps
- Resolución : 360p (640x360) o 720p (1280x720)

Para el stream de grabación de video, estos parámetros están fijos en 720p, 30FPS y 4Mbps (no pueden modificarse).

Mientras el drone se encuentre grabando video, el hardware H264 deja de estar disponible. Para cubrir este caso el AR.Drone 2.0 utiliza un codificador MPEG4.2 implementado por software para la codificación del stream de video en vivo. Este codificador puede ser ajustado con los siguientes parámetros:

- FPS: Entre 15 y 20
- Bitrate: Entre 250kbps y 1Mbps

Encapsulamiento del video en la red

Para ser transmitido por la red, los cuadros de video (frames) son divididos en 'slices' y enviados en paquetes con encabezados específicos, que contienen información importante acerca del cuadro de video. Estos encabezados son llamados PaVE (Parrot Video Encapsulation) y su estructura es la siguiente:

```
typedef struct {
  uint8_t signature[4]; /* "PaVE" - usado para identificar
    el inicio del frame */
  uint8_t version; /* Version del codigo */
  uint8_t video_codec; /* Códec del frame */
  uint16_t header_size; /* Tamaño de parrot_video_encapsulation_t */
  uint32_t payload_size; /* Cantidad de datos de este PaVE */
  uint16_t encoded_stream_width; /* ej: 640 */
  uint16_t encoded_stream_height; /* ej: 368 */
  uint16_t display_width; /* ej: 640 */
  uint16_t display_height; /* ej: 360 */
  uint32_t frame_number; /* Posición del frame en el stream actual */
  uint32_t timestamp; /* En milisegundos */
  uint8_t total_chuncks; /* Actualmente no se usa */
  uint8_t chunck_index ; /* Actualmente no se usa */
  uint8_t frame_type; /* Puede ser I-frame o P-frame -
    parrot_video_encapsulation_frametypes_t */
  uint8_t control; /* Comandos especiales, como el fin del stream */
  uint32_t stream_byte_position_lw; /* Posición en bytes del frame
    en el stream de video codificado - lower 32-bit word */
  uint32_t stream_byte_position_uw; /* Posición en bytes del frame
    en el stream de video codificado - upper 32-bit word */
  uint16_t stream_id; /* Identifica paquetes que deben estar juntos */
  uint8_t total_slices; /* Número de 'slices' del frame actual */
  uint8_t slice_index ; /* Posición del slice actual en el frame */
  uint8_t header1_size; /* Sólo para H.264 : Tamaño del SPS
    inside payload */
  uint8_t header2_size; /* Sólo para H.264 : Tamaño del SPS
    inside payload */
  uint8_t reserved2[2]; /* Padding para alinear la info en 48 bytes */
  uint32_t advertised_size; /* Tamaño anunciado de los frames */
  uint8_t reserved3[12]; /* Padding para alinear la info en 64 bytes */
} __attribute__ ((packed)) parrot_video_encapsulation_t;
```

Transmisión del stream de video en la red

El stream de video del AR.Drone 2.0 es transmitido a través del socket TCP-5555. Tan pronto como un cliente se conecta a la red, el drone comienza la transmisión de video.

Un frame puede ser enviado mediante varios paquetes TCP, y por ende debe ser reconstituido por la aplicación en el lado del cliente, antes de poder decodificarlo utilizando el decodificador de video.

Stream de grabación de video

Este stream utiliza el socket TCP-5555 para transmitir frames codificados en H264 de 720p. Este stream se deshabilita cuando la aplicación del cliente no está grabando. Utiliza el mismo tipo de encapsulamiento (PaVE) que el stream de video en vivo.

5. Live Robot Programming

El concepto de Live Programming no es nuevo, se encuentra ya referenciado en trabajos de Tanimoto de la década de 1990 [57]. No obstante, últimamente ha atraído la atención gracias a la muy comentada charla de Bret Victor en CUSEC'12 [58]. En ella se muestra un sistema que está siempre activo, en el cual la imagen de un árbol, que es generada programáticamente, refleja al instante cada edición realizada sobre su código. Así por ejemplo, cuando el programador edita el parámetro correspondiente al largo de las ramas, en la imagen éstas se encogen o estiran a medida que el nuevo valor es digitado. Este efecto se logra gracias a que cada acción de edición causa cambios inmediatos en la computación del programa y por ende en sus valores de salida.

Live programming propone que el programador se beneficia enormemente al tener una conexión inmediata con el programa que está escribiendo, ya que esta modalidad estrecha la distancia entre los eventos de creación de código y la observación de su comportamiento con la consecuente retroalimentación. En programación para robots, la falta de mecanismos de retroalimentación inmediata es evidente, ya que una de sus mayores dificultades radica en la gran brecha que separa los eventos de creación del código y la realización de pruebas en un simulador o en un robot real. Entre ambos sucesos el programador debe compilar el programa y cargarlo al hardware del robot o al simulador. En caso de detectar errores en el comportamiento del robot, el programador debe rastrear el error mentalmente de vuelta a su código. Este proceso interrumpe el hilo de pensamiento relacionado al diseño y corrección del código y mella la continuidad de la labor, eficiencia y probabilidad de éxito del programador.

En este contexto el estudiante de doctorado Miguel Campusano y el profesor Johan Fabry, ambos integrantes del Departamento de Ciencias de la Computación (DCC) de la Universidad de Chile, proponen un nuevo lenguaje de programación para robots que implementa Live Programming. Este lenguaje, denominado LRP (Live Robot Programming) [1], permite programar sobre un robot o simulador que se encuentra ejecutando el código mientras éste está siendo escrito.

En las siguientes secciones se explicará brevemente el diseño del lenguaje y su sintaxis, se hablará sobre las pruebas de uso realizadas a la fecha y los requerimientos para utilizarlo en un nuevo robot.

5.1. Diseño de LRP

LRP (Live Robot Programming) es un lenguaje basado en máquinas de estado anidadas. Comparte esta característica con varios otros exitosos lenguajes de programación para robots, pero aquello que lo diferencia de trabajos previos es que se enfoca en implementar Live Programming.

En LRP el comportamiento del robot es modelado mediante una máquina de estados, dentro de la cual cada estado representa simbólicamente una etapa de su comportamiento, y cada transición, un cambio de estado originado por un evento que la gatilla. En una máquina de estados anidada, cualquier estado puede contener en su interior una máquina de estados completa, cuyos estados a la vez podrían contener otra máquina y así sucesivamente. De esta forma, cada etapa en el comportamiento del robot podría ser detallada en tantos niveles de profundidad como se deseara.

Cada máquina de LRP puede definir *variables* con scope estático, cuyo alcance abarca la totalidad de la máquina, más las máquinas que tenga anidadas en todo nivel.

Dentro de una máquina, es posible definir *acciones*. Una acción es un trozo de código escrito en el lenguaje de programación Smalltalk, que se ejecuta atómicamente. En particular, un estado puede contener tres tipos de acciones:

- *onentry*, se ejecuta cuando el estado pasa a ser el estado activo.
- *running*, se ejecuta mientras el estado se encuentre activo, una vez por ciclo de ejecución.
- *onexit*, se ejecuta cuando el estado deja de estar activo y sólo una vez que la máquina de estados anidada, en caso de existir, ha sido descartada. Esto ocurrirá sólo después de que haya sido ejecutada la acción onexit del estado activo de la máquina anidada.

Una acción tiene acceso a todas las variables en su scope, esto es, las variables de la máquina en que se define, más las de la máquina que la define a ella y así hasta la raíz de la jerarquía.

Una máquina también puede contener eventos. Éstos son acciones específicas que determinan condiciones para que ocurran las transiciones o, dicho de otra manera, los cambios de estado. El evento ocurrirá si el resultado de su evaluación es true.

Dos estados pueden estar unidos por una transición dirigida. De este modos las transiciones definen el flujo de la máquina. Cuando ocurre un evento, las transiciones con origen en el estado activo son inspeccionadas para revisar si deben ejecutarse según sus condiciones. Esta inspección se inicia en la máquina raíz y continúa hacia el interior de la jerarquía. Si más de una transición, con origen en el estado activo, cumple su condición de ejecución, se ejecuta sólo la que fue definida primero léxicamente. LRP especifica 4 tipos de transiciones:

- normal: son las descritas en el párrafo anterior.
- epsilon: ocurren automáticamente cuando su estado padre es activado.
- timeout: ocurren automáticamente cuando un tiempo límite es alcanzado.
- wildcard: carecen de estado inicial, consideran como su estado inicial a todos los estados de la máquina en la que se definen.

En la figura 9 se muestra el editor de LRP, que incorpora en su sección central una visualización de las máquinas de estado según su jerarquía. Cuando una de estas máquinas es seleccionada, su estructura en detalle se despliega en la sección derecha de la ventana. Esta visualización indica el estado actualmente activo y las últimas transiciones gatilladas, junto con las variables en el scope de la máquina seleccionada. Todo el sistema cambia en sincronía con la edición del código.

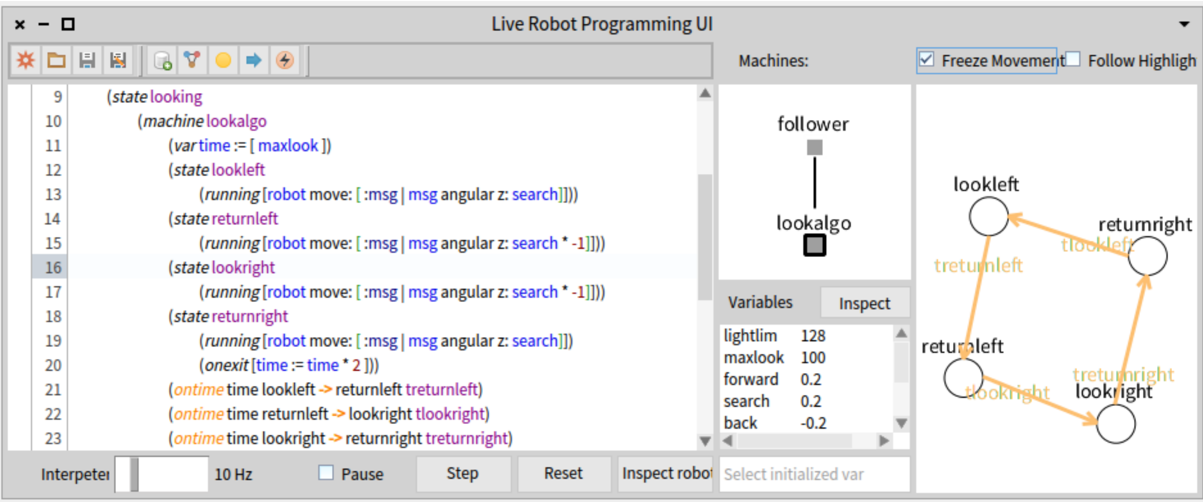


Figura 9: Editor de LRP

LRP se comunica con el robot mediante una variable global llamada *robot* que actúa como una interfaz de acceso a sus sensores y propiedades. Esta variable actúa como puente entre LRP y la API del robot en cuestión. Gracias a esta manera de proceder LRP es capaz de funcionar sobre prácticamente cualquier robot que provea una API de comunicación. La programación del puente entre LRP y la API de un robot tiene especial importancia para este trabajo, pues corresponde a una etapa en el desarrollo de esta solución. La sección 5.3 ofrece una descripción más completa de este puente .

5.2. Sintaxis de LRP

Para ilustrar la sintaxis de LRP se presenta la implementación del comportamiento lógico de un robot que sigue una línea negra pintada en el piso. Se trata de un robot con un sensor de luz frontal y un parachoques delantero sensible a la presión, que utiliza el middleware ROS [3]. El robot deberá seguir la línea hasta que choque con un obstáculo, dar la media vuelta y seguir la línea en sentido contrario hasta encontrar otro obstáculo, después de lo cual se finaliza la ejecución el programa. Este programa de ejemplo fue creado por integrantes del DCC, antes de la ejecución de este trabajo.

A grandes rasgos este código crea una máquina llamada *follower*, que define lo siguiente:

- 5 estados (state): *moving, looking, bumpback, bumpturn, end*
- 4 eventos (event): *outofline, intheline, bumping, ending*
- 6 transiciones (on): *tlooking, tmoving, tend*, y 3 transiciones anónimas
- Es importante notar que looking es un estado complejo, que contiene una máquina llamada lookalgo en su interior. La figura 10 ilustra la máquina follower: sus estados y transiciones según son desplegadas la interfaz de LRP.

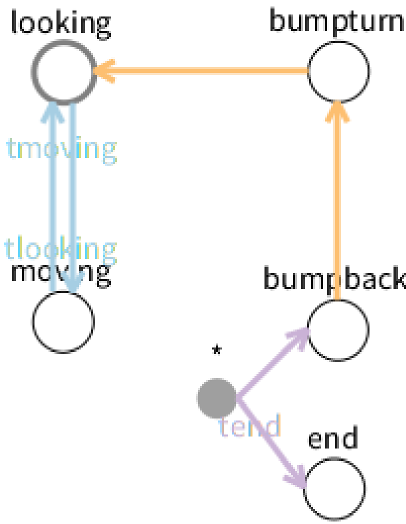


Figura 10: Diagrama de la máquina follower generado por LRP

A continuación se presenta el código asociado a este ejemplo. Éste ha sido dividido en secciones que son explicadas por separado.

```
1 (var lightlim := [128]) (var maxlook := [100]) (var forward := [0.2])
2 (var search := [0.2]) (var back := [-0.2]) (var turn := [1])
```

Las primeras dos líneas definen variables globales, que están disponibles para ser accesadas y editadas a lo largo de todo el código. Estas variables sirven para establecer parámetros que afectarán el comportamiento del robot. Por ejemplo la variable lightlim determina el umbral de luz detectado por el sensor, que distingue el blanco del negro.

```
3 (machine follower
4   (state moving (running [robot move: [ :msg | msg linear x: forward]]))
5   (on outofline moving -> looking tlooking)
6   (on intheline looking -> moving tmoving)
7   (event outofline [robot light data > lightlim + 10])
8   (event intheline [robot light data < lightlim - 10])
```

En la línea 3 la máquina follower es definida, las siguientes líneas crean elementos en su interior. La línea 4 define el estado moving y dentro de éste, la acción running asociada, que se ejecutará mientras este estado se encuentre activo. Notar que la definición de running es un bloque de código Smalltalk, en el cual se envía un mensaje a un objeto llamado robot, que actúa como interfaz de comunicación con el robot controlado. Esta variable tiene especial importancia y se describirá con más detalle posteriormente. Las líneas 7 y 8 crean eventos. El evento outofline ocurre cuando la lectura del sensor de luz del robot supera el valor lightlim + 10, es decir, cuando el sensor percibe blanco. El segundo evento, intheline, ocurre cuando la lectura del sensor del robot es inferior a lightlim - 10: esto sucede cuando el sensor de luz percibe negro. Las líneas 5 y 6 definen transiciones, la primera, llamada tlooking, lleva desde el estado moving a looking y

ocurre cuando el evento outofline se gatilla; la segunda, llamada tmoving, lleva desde el estado looking a moving y ocurre cuando el evento intheline es gatillado.

```
9    (state looking
10     (machine lookalgo
11       (var time := [ maxlook ])
12       (state lookleft
13         (running [robot move: [ :msg | msg angular z: search]]))
14       (state returnleft
15         (running [robot move: [ :msg | msg angular z: search * -1]]))
16       (state lookright
17         (running [robot move: [ :msg | msg angular z: search * -1]]))
18       (state returnright
19         (running [robot move: [ :msg | msg angular z: search]])
20         (onexit [time := time * 2 ]))
21       (ontime time lookleft -> returnleft treturnleft)
22       (ontime time returnleft -> lookright tlookright)
23       (ontime time lookright -> returnright treturnright)
24       (ontime time returnright -> lookleft tlookleft))
25     (onentry (spawn lookalgo lookleft)))
```

Desde la línea 9 hasta la 25 se define el estado compuesto looking. Dentro suyo, definida entre las líneas 10 y 24, se encuentra la máquina lookalgo (llamado así por look-algorithm) y, en la línea 25, su acción onentry, según la cual cuando el estado looking es activado, se genera la máquina lookalgo con estado inicial lookleft. La definición de la máquina lookalgo es análoga a follower y no será detallada.

```
26   (var nobump := [true])
27   (event bumping [robot bumper data == 1 & nobump])
28   (event ending [robot bumper data == 1 & nobump not])
29   (on bumping *-> bumpback)
30   (on ending *-> end tend)
31   (state bumpback
32     (onentry [ nobump = false])
33     (running [ robot move: [ :msg | msg linear x: back]]))
34   (state bumpturn
35     (running [ robot move: [ :msg | msg angular z: search]]))
36   (ontime 1000 bumpback -> bumpturn)
37   (ontime 3000 bumpturn -> looking)
38   (state end)
39)
40(spawn follower looking)
```

Esta última sección de código es la responsable del comportamiento de choque, retroceso, y giro para retornar. Como ya se dijo, el robot deberá seguir la linea hasta chocar con un obstáculo,

luego deberá dar la media vuelta y seguir la linea hasta volver a chocar con un obstáculo. Los dos eventos más relevantes para este comportamiento son el primer y el segundo choque, los cuales son definidos en las líneas 28 y 28 y son llamados bumping y ending correspondientemente. La variable nobump, definida en la linea 26, indicará si el primer choque ya ha sucedido o no. Así, la courrencia de los eventos bumping y ending dependerán de la variable nobump y de la lectura del sensor de choque del robot (bumper).

Después del primer choque, antes de que el robot de la media vuelta, es necesario que retroceda un pequeño trecho. Para esto se define el estado bumpback en la linea 31, que se activa cuando el evento bumping ocurre, sin importar cual sea el estado activo. Esto es así porque la transición asociada es de tipo wildcard (ver linea 29). El estado bumpback se mantendrá activo sólo durante un segundo, ya que se encuentra conectado al estado bumpturn mediante una transición tipo timeout (linea 36). El estado bumpturn es el responsable que girar el robot en 180 grados aproximadamente. El robot girará durante 3 segundos y luego pasará a buscar la linea y avanzar. Esto puede verse en la transición tipo timeout de la linea 37. Cuando el estado end se activa, el robot ha chocado con el segundo obstáculo y ha dejado de avanzar.

La última linea de código (linea 40) especifica que cuando la ejecución se inicie, la máquina follower sea creada con estado inicial looking.

5.3. Puente entre LRP y la API del robot

Como se ha dicho anteriormente LRP tiene el potencial de funcionar con cualquier robot que provea algún tipo de API de control. Esto es así gracias a que se vale de una variable global llamada `robot`, utilizada para efectuar todas las comunicaciones con el robot, ya sea leyendo sus sensores, enviándole instrucciones, etc. La variable robot y toda la infraestructura asociada a su uso, es definida al interior del llamado Puente entre LRP y la API del robot. Así, siempre y cuando se cuente con un puente, será posible usar LRP con todo robot, incluso si éste provee una API de comunicación que no fue diseñada para ser usada en la modalidad Live Programming.

A la fecha se han programado dos puentes para LRP: uno que lo comunica con ROS y otro con Ev3 Lego Mindstorms. Gracias a éstos se han efectuado pruebas en 3 tipos de robots distintos: Turtlebot, PR2 y Lego Mindstorm.

Para poder utilizar LRP con el drone, será necesario programar un puente entre la API del drone y LRP.

Ya que la infraestructura de LRP para los puentes fue diseñada pensando en la flexibilidad, la creación de un nuevo puente consiste simplemente en concretizar una clase abstracta, denominada `LRPAbstractBridge`, que contiene un único cuya implementación es obligatoria: `openUI`, que es responsable de inicializar el puente y poner a disposición la pseudovariable `robot`.

5.4. Pharo Smalltalk

LRP ha sido implementado utilizando el lenguaje de programación Pharo Smalltalk, que se caracteriza por estar completamente orientado a objetos y por proveer un poderoso ambiente de

programación enfocado en la simplicidad y en la retroalimentación instantánea. En Pharo todas las interacciones ocurren a través del envío de mensajes a objetos, los cuales son responsables de su propio comportamiento ante la recepción de un mensaje.

LRP se favorece de la retroalimentación en tiempo real que provee Pharo, pues este último ofrece manipulación inmediata de objetos y actualización en tiempo real, lo cual permite al programador trabajar sobre un ambiente que está siempre activo y que reacciona al instante.

En vista de lo anterior el puente entre la API en Pharo del drone y LRP debe ser programado en Pharo, de lo contrario el intérprete de LRP no podría usarlo. Dada esta situación, se ha optado por implementar la API de comunicación con el drone en este mismo lenguaje, para así evitar problemas de compatibilidad entre los módulos.

Capítulo II.

Descripción de la Solución

Luego de la investigación preliminar y a partir de los objetivos generales definidos en la introducción de este documento, se ha resuelto que para permitir el control del drone desde el lenguaje de programación LRP, es necesario proveer una plataforma de conexión compuesta por dos componentes principales:

1. Una API de comunicación con el drone implementada en Pharo Smalltalk, cuya principal función es ofrecer mecanismos para comunicarse con el drone desde Pharo. Debe ser capaz de establecer y mantener los canales de conexión con el drone durante el tiempo requerido por el usuario. Adicionalmente debe contener métodos para recibir y decodificar los datos de navegación provenientes desde el drone. Por último, debe ofrecer funciones para enviar comandos que controlan al robot.

2. Una aplicación tipo puente, necesaria para utilizar la API en Pharo desde LRP. Este puente debe proveer métodos que controlan al robot y que pueden ser utilizados desde LRP.

Se determina adicionalmente que para validar la solución sean creados programas de ejemplo que controlen al drone desde LRP, y que naturalmente utilicen la infraestructura implementada durante este trabajo de memoria. La creación de estos programas de prueba busca adicionalmente comprobar la efectividad de utilizar LRP para controlar un robot aéreo con las características del drone.

Una visión general de los componentes de la solución completa y su interrelación es expuesta en el diagrama de la figura 11. En él toman lugar el drone y el cliente como dos grandes sistemas que se comunican a través de la red WiFi suministrada por el firmware del robot. Al interior del sistema operativo del cliente se ejecuta la máquina virtual de Pharo, y dentro de ésta habitan todas las componentes de la solución.

En el diagrama se muestra que el drone envía y recibe datos hacia y desde el dispositivo cliente utilizando cuatro canales de comunicación definidos por protocolos y puertos específicos. La misión de interpretar los datos recibidos y de crear los comandos que se enviarán al drone recae sobre la API de comunicación. Su implementación resulta prioritaria y se define como el primer paso de la implementación del sistema, dado que el diseño y funcionalidad del resto de los componentes depende de ella.

La aplicación puente, referida en la figura como LRP Bridge, se ejecuta también dentro de Pharo. Sólo las funciones definidas al interior de este puente estarán disponibles para ser utilizadas desde LRP. La función del puente es, en consecuencia, adaptar las funciones de comunicación con el drone, enmascarándolas mediante una interfaz apropiada para LRP.

La componente Parrot SDK video receiver representa un programa escrito en C, que es parte del SDK de Parrot para el AR.Drone 2.0. Esta herramienta decodifica y reproduce el stream de video proveniente del drone, y es utilizada desde la API como una componente externa.

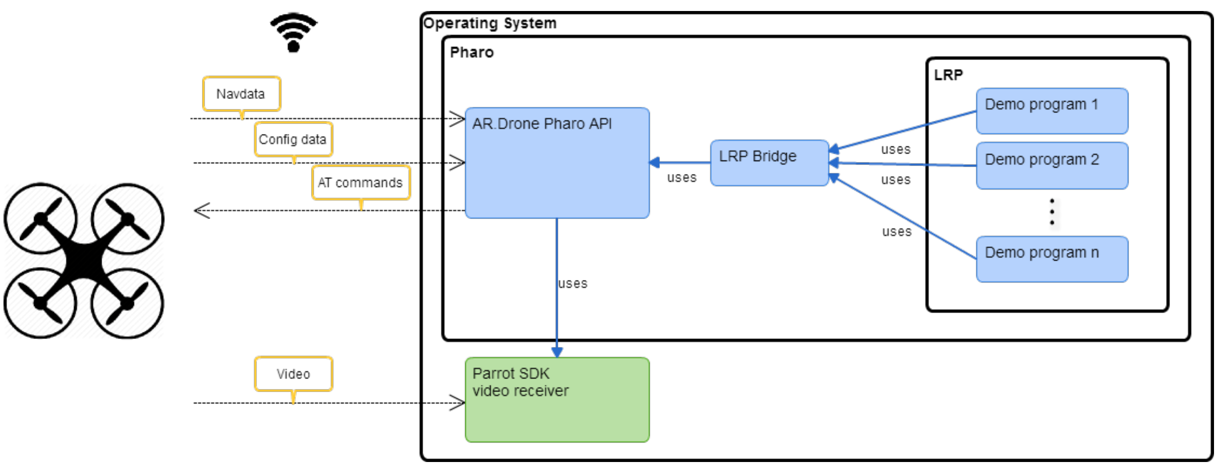


Figura 11: Diagrama de componentes de la solución

6. API en Pharo para comunicación con el AR.Drone

La descripción de esta API abarcará las secciones 6, 7 y 8 de este documento. La sección 6 presenta una descripción general de los objetivos de la API, describe su diseño e incluye consideraciones que son relevantes para comprender las secciones posteriores. La sección 7 descompone la API para poder explicar su estructura y funcionamiento en profundidad. Luego la sección 8 contiene toda la información necesaria para poder utilizar la API como un usuario estándar, sin necesidad de conocer sus mecanismos internos o los protocolos de comunicación exigidos por el drone.

Como se indica en la sección 4.3, existen numerosas librerías para comunicarse con el AR.Drone, pero ninguna de ellas permite hacerlo desde el lenguaje y ambiente de programación Pharo. Interactuar con el drone desde Pharo es el primer paso para posteriormente hacerlo desde el lenguaje de programación para robots LRP.

Las funcionalidades de la API quedaron determinadas por los requerimientos que exigidos por los programas de ejemplo creados en LRP, con los cuales se pretendía comprobar la factibilidad de utilizar LRP en un robot como el drone. Tuvo prioridad la implementación del envío de comandos que controlaran el movimiento y la configuración del drone; junto con la implementación de funciones que posibilitaran la recepción de datos publicados por el robot. Estos datos son relevantes para el cliente porque informan acerca del estado actual del robot, lo cual le permite adaptar su comportamiento de manera acorde.

En términos generales, la API encapsula las siguientes labores:

- Establecer comunicación inalámbrica con el drone, y comprobar que el envío y recepción de paquetes se realice correctamente.
- Mantener activa la conexión con el drone y recuperarla en caso de que se pierda. Notificar en caso de que la reconexión no sea posible.
- Recepcionar los datos de navegación enviados desde el drone, decodificarlos y encapsularlos en objetos representativos que puedan ser utilizados por el usuario, ya sea directamente a través de la definición de un bloque de código que se ejecute cada vez que un paquete sea recibido y que utilice los datos de dicho paquete; o indirectamente, consultando el estado del drone, a través de mensajes ad hoc ofrecidos por la API.
- Enviar comandos de control de movimiento y automatizar el envío de mensajes cuya emisión depende del estado interno del drone (por ejemplo, envío de comando para que el drone vuelva a su estado normal, cuando se encuentra en estado de emergencia).
- Proveer métodos que informen acerca del estado interno del drone y de su estado de conexión con el cliente.
- Notificar la ocurrencia de excepciones, ya sean de red, de autenticación o relacionadas con el estado interno del drone. Por ejemplo: una excepción es arrojada si se intenta despegar cuando el drone tiene batería insuficiente. Esta labor considera la incorporación de un mecanismo complementario para que el usuario pueda reaccionar adecuadamente ante cada

excepción a través de la definición de funciones (bloques de smalltalk) tipo callback asociadas a una excepción particular, las cuales serán ejecutadas cada vez que dicha excepción ocurra.

6.1. Diseño y arquitectura

El diseño de la API considera muy importante la simplicidad de uso por parte del usuario. Múltiples comportamientos y rutinas de comunicación requeridas por el drone se han encapsulado en funciones de alto nivel y en procesos que se ejecutan en segundo plano, de modo que el usuario final pueda programar comportamientos en el robot sin siquiera conocer los protocolos específicos exigidos por el firmware del drone.

La arquitectura del sistema se erige alrededor de una clase principal llamada ARDrone. Ésta contiene los métodos públicos de la API, los que en conjunto bastan para que cualquier usuario pueda controlar al robot e informarse acerca de su estado interno. El resto de las clases de la aplicación son instanciadas o utilizadas indirectamente por los métodos de la clase ARDrone.

La figura 12 esquematiza las principales componentes de la librería clasificadas de acuerdo a su función. Una componente no está necesariamente asociada a una clase particular, más bien simboliza un sub sistema con un rol determinado. En términos generales el esquema muestra al drone y a la API comunicándose mediante 3 canales (navdata, AT commands y configdata).

En dicho esquema se puede ver cómo en la API en Pharo los paquetes son recibidos y enviados siempre a través de los Adapters, y que la única componente que los utiliza es Process Managers. Esta última emplea las componentes Drone Internal State y Drone Motion State para decidir qué paquetes enviar y cuándo, a la vez que modifica el Drone Internal State de acuerdo a la información que recibe desde el drone. Por otro lado Drone State Manager reacciona de acuerdo al contenido de Drone Internal State y anuncia cambios de estado del drone que son relevantes.

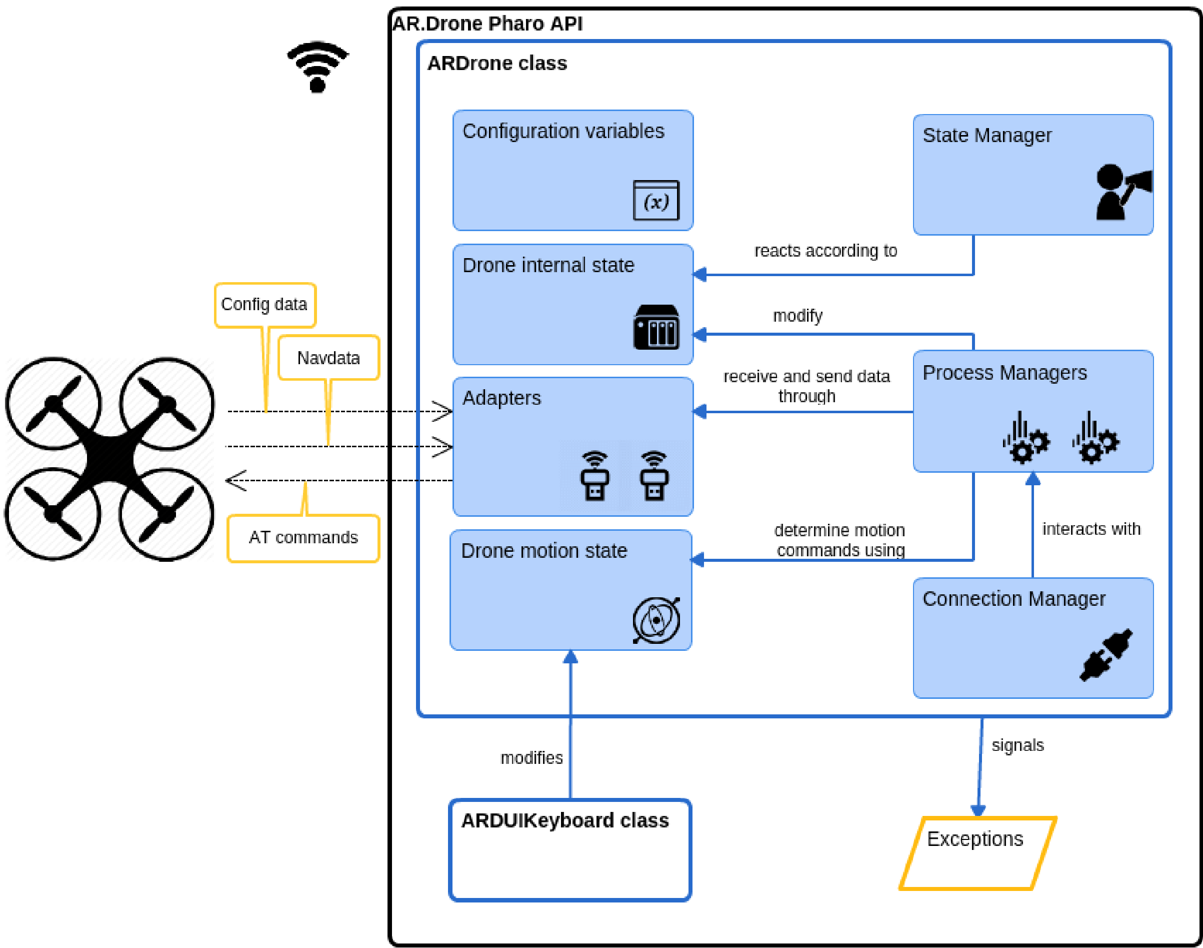


Figura 12: Diagrama de componentes de la API

A continuación se ofrece una breve descripción de cada componente:

- Los Adaptadores (Adapters) administran los puertos de comunicación (sockets) con el drone. Se utilizan para enviar y recibir mensajes. Adicionalmente se encargan de agregar un número de secuencia adecuado a cada paquete enviado.
- Las variables de configuración (Configuration variables) son un conjunto de variables de instancia con valores fijos, especificados por los requerimientos del firmware del drone. Estos valores son utilizados por el resto de las componentes de la librería. Especifican, por ejemplo: la IP del drone, los puertos de comunicación, etc.
- El estado interno del drone (Drone internal state) es una clase que representa el estado interno del robot según los datos de navegación informados por éste. Sus valores son actualizados conforme los datos de navegacion arriban. Esta clase tiene gran relevancia, pues el comportamiento de varias otras componentes del sistemas dependen del estado del robot.
- El administrador de estado (State Manager) reacciona ante los cambios en el estado interno del drone, y se encarga indicarlos mediante anuncios de Pharo (*announcers*)
- El estado de movimiento del drone (Drone motion state) contiene los niveles de inclinación y velocidad deseados para el drone en cada momento. Los valores almacenados en esta componente determinan los comandos de movimientos que serán enviados al drone por el proceso correspondiente.

- Los administradores de procesos (Process Managers) crean y administran procesos livianos (threads), cada uno de los cuales cumple una labor específica relacionada con el envío de comandos de acuerdo al estado interno del drone y a las acciones del usuario, y con el procesamiento de datos recibidos. Esta componente encapsula a 4 clases que serán detalladas más adelante.
- El administrador de conexión (Connection Manager) es el responsable de iniciar y mantener viva la conexión con el drone, de realizar los intentos de reconexión cuando el enlace se pierda, y de notificar cuando sea imposible conectarse.

Para poder presentar una descripción detallada de la API según sus componentes, es necesario introducir algunos conceptos relevantes. Por ello se adjunta continuación la sección 6.2, que prepara al lector para comprender a cabalidad la descripción en detalle de la API contenida en la sección 7.

6.2. Observaciones y conceptos relevantes

6.2.1. Observaciones sobre la configuración de los datos de navegación

Tan pronto como el cliente establece conexión con el drone y envía un paquete UDP con «algunos bytes», como dice la documentación de Parrot, a través del puerto de navdata, el drone inicia el envío de datos de navegación en un stream que se mantiene activo durante toda la sesión.

Los posibles modos en que el drone envía navdata son los siguientes:

- Modo Bootstrap (o de arranque): cada paquete enviado por el drone contendrá sólo su número de secuencia y el status del drone.
- Modo Demo: cada paquete enviado por el drone podrá contener tanta información de navegación como el cliente haya solicitado. Si el cliente no especifica los datos que requiere, el drone enviará por defecto datos de su posición y velocidad, y de detección de tags. La tasa de envío rondará los 15 paquetes por segundo.
- Modo Debug: cada paquete contendrá toda la información de navegación disponible (full mode). Ésta puede incluir información de debugging que no es útil para vuelos rutinarios. En este modo se enviarán alrededor de 200 paquetes por segundo.

Luego de iniciar, el drone partirá enviando paquetes en modo Bootstrap. El cliente puede cambiar a modo Demo o Debug, enviando un comando al puerto de comandos, del tipo:
AT*CONFIG=605,"general:navdata_demo","TRUE"<**LF**>
para modo Demo, o el mismo comando con parámetro "FALSE" para modo Debug (ver Observaciones sobre el envío de comandos en la siguiente sección).

Mediante el envío de comandos de configuración, puede especificarse también el tipo y cantidad de datos de navegación que serán informados por el drone. Como se vio en la sección 4.3.2, la estructura de los paquetes de navegación consiste en un encabezado seguido de n opciones.

Las opciones ofrecidas por el drone son en total 29. Algunas informan en detalle las mediciones brutas de los sensores del drone (magnetómetro, acelerómetro y giroscopio); otras ofrecen

estimaciones sobre variables del entorno, como la velocidad del viento, presión del aire, etc; ciertas opciones informan acerca del estado interno del firmware del drone; algunas incluyen detalles técnicos acerca del video; etcétera. En conclusión la mayoría de las opciones que el drone envía a través de los datos de navegación, contienen información altamente específica, que no se necesita para efectos del proyecto descrito en el presente documento. Para efectos de este trabajo, sólo se han utilizado dos opciones: «Demo» y «Vision Detect».

Hay que notar que el nombre «Demo» es utilizado para referirse a dos conceptos distintos: el Modo Demo y la Opción Demo; ambos no deberán confundirse. El «Modo Demo» se refiere a una opción de configuración del drone, según la cual los datos de navegación se envían a una tasa de 15 por segundo, y el tipo de datos que cliente quiera recibir es configurable. La «Opción Demo» se refiere a un contenido específico incluido al interior de cada paquete de navegación.

La opción «Demo» está habilitada por defecto en el Modo Demo. Informa acerca de la inclinación, velocidades, estado de vuelo, fps del stream de video y nivel de batería del drone.

La opción «Vision Detect» debe ser habilitada por el cliente mediante el envío de un paquete de configuración. Incluye información acerca de los tags detectados, con un máximo de 4 tags a la vez; y para cada uno de ellos informa:

- Tipo.
- Coodenadas X e Y de su centro en la imagen del video, considerando (0,0) la esquina superior izquierda y (1000, 1000) la esquina inferior derecha.
- Alto y ancho de la caja de detección que lo contiene, en la imagen del video.
- Distancia a la que se encuentra de la cámara en centímetros. Mediante pruebas se ha determinado que la distancia informada es aproximadamente un 15 % menor que la distancia real.
- Ángulo de orientación, sólo para el tipo de marca «Oriented Roundel».
- Cámara en la que fue detectado.

LECCIÓN APRENDIDA

> El Modo Demo es uno de los tres modos en que el drone puede enviar datos de navegación. Por otro lado la Opción Demo se refiere a un contenido específico que está contenido en los paquetes de navegación, cuyo envío está habilitado por defecto.

6.2.2. Observaciones sobre el envío de comandos

Para que firmware del drone pueda interpretar los comandos recibidos, éstos deben respetar un formato y codificación determinados, descritos en la documentación oficial de Parrot. De acuerdo

a ello, un comando de control o AT command consiste en un string en formato ASCII de 8 bits, conformado por un prefijo (según el tipo de comando), un número de secuencia, un conjunto de argumentos separados por comas (0x2C), y un caracter de fin de línea <**LF**> (0x0A).

El siguiente es un ejemplo de AT command:

AT*PCMD=605,1,0,0,0,0<**LF**>

En éste, **AT***PCMD es el prefijo, a continuación el entero 605 es el número de secuencia, y los valores posteriores son los argumentos del comando. El fin del comando queda determinado por el caracter de fin de linea <**LF**>.

Los argumentos que pueden enviarse al interior de un comando pueden ser de tres tipos:

- Un entero con signo, incorporado como el string correspondiente a su representación decimal.
- Un valor String propiamente tal, que debe ir entre comillas dobles.
- Un float, cuya representación como String no podrá enviarse directamente como argumento, sino que deberá ser procesado extrayendo la palabra de 32 bits codificada con los bits que determinan su valor e interpretándola como un entero con signo, cuya representación decimal se integrará al comando para representar al float. Por ejemplo el float -0.8, que según el formato IEEE-754 es guardado en memoria como la palabra de 32 bits *BF4CCCCD*, queda representado por valor que resulta de interpretar esa palabra como un entero con signo, en este caso -1090519040.

Respecto al número de secuencia que debe contener cada comando, es requisito que comandos posteriores posean números de secuencia mayores a aquellos que los anteceden. De este modo se evita que el drone procese instrucciones antiguas por error.

Dentro de la API, se asigna la responsabilidad de construir los AT commands a una clase llamada ARDCommand. No obstante, la labor de asignar números de secuencia crecientes y consecutivos escapa a las responsabilidades de esta clase, y se delega a los Adaptadores UDP, los que justo antes de enviar cada mensaje lo modifican, asignándole el número de secuencia que corresponde. Como existen diversas instancias de adaptadores y éstos son utilizados paralelamente por diversos procesos, la variable que determina la secuencia de los comandos se almacena como variable de instancia de la clase principal ARDrone, y su accesor, que a su vez la incrementa, se encuentra protegido ante concurrencia.

7. Descomposición de la API en Pharo según componentes

En la sección 6.1 se ha presentado el diseño de la API en Pharo utilizando una clasificación funcional de sus partes. Esta clasificación dio lugar a la definición de 7 «componentes funcionales», cuyas interacciones y descripciones fueron expuestas brevemente e ilustradas en la figura 12. En la presente sección cada componente funcional es descrita en detalle. A cada componente se le ha dedicado una subsección.

7.1. Configuration variables

Esta componente representa valores estáticos almacenados en la clase ARDConstant, y que están determinados de antemano por el firmware del drone. El diseño contempla que estos valores sean obtenidos directamente a través de métodos accesores, ya que de este modo pueden ser sugeridos por Pharo mientras el usuario escribe código. Por el motivo anterior se ha decidido no utilizar arreglos para implementar las enumeraciones necesarias para la API, y en su lugar se ha almacenado cada valor de cada enumeración en una variable de clase. Esta decisión hizo crecer la cantidad de constantes presentes en la clase. Para organizarlas se han clasificado en protocolos, y se les ha nombrado utilizando prefijos compartidos por aquellas constantes que son utilizadas para propósitos similares. Por ejemplo las variables que definen posibles animaciones para los LEDs del drone se han agrupado bajo el protocolo «led animations» y comparten el prefijo «ledAnimation». Entre ellas se encuentran ledAnimation_BLINK_RED, ledAnimation_GREEN, ledAnimation_FIRE, etc.

7.2. Adapters

La componente Adapters representa a la totalidad de adaptadores utilizados para intercambiar mensajes con el drone. Cada adaptador es una instancia de la clase ARDAdapterUDP o de ARDAdapterTCP según corresponda. Un adaptador contiene:

- Una instancia de *Socket* configurada con la IP del drone, un puerto de acuerdo a su rol (navdata, AT command o configdata), y una variable que indica si escucha las IP Multicast o no.
- Una variable *delay,* sólo para los Adaptadores UDP, correspondiente al mínimo intervalo de tiempo que debe transcurrir entre el envío de dos mensajes consecutivos.
- Una variable *mutex* para permitir el uso del adaptador desde distintos procesos ligeros, específicamente para proteger la sección crítica de código encargada de enviar un único paquete y de respetar el delayInterval requerido.

Concretamente, la clase ARDrone utiliza tres Adapters: dos Adapters UDP y un Adapter TCP:

- *navdataAdapter* (UDP) : utilizado para recibir datos de navegación. Su configuración particular es la siguiente:
    - Socket:

    - IP: 192.168.1.1
    - port: 5554
    - isMulticast: TRUE (Este dato no se menciona en la documentación oficial de Parrot, y fue obtenido a partir de investigación externa)
  - delay: 30 milisegundos
- *commandAdapter* (UDP): utilizado para enviar AT commands. Su configuración es la siguiente:
  - Socket:
    - IP: 192.168.1.1
    - port: 5556
    - isMulticast: FALSE
  - delay: 30 milisegundos
- *configurationAdapter* (TCP) : utilizado para recibir la configuración interna del drone. Su conformación es la siguiente:
  - Socket:
    - IP: 192.168.1.1
    - port: 5559
    - isMulticast: FALSE

LECCIÓN APRENDIDA

Los datos de navegación son transmitidos por el drone utilizando el método multicast, por lo que el Socket que los reciba debe estar configurado para escuchar dichas transmisiones.

7.3. Process Managers

Es la componente más intrincada de la API. Contiene cuatro clases: Connection Manager, Navdata Manager, Command Manager y Configuration Manager, todas ellas se instancian sólo una vez al interior de ARDrone y comparten la responsabilidad de crear y administrar procesos que efectúan en conjunto todas las tareas de comunicación con el drone.

7.3.1. Connection Manager

La creación de esta componente se hizo necesaria porque era frecuente que durante las pruebas de vuelo se perdiera conexión con el drone por algunos segundos, lo que afectaba a los procesos encargados de enviar y recibir datos. Contar con un administrador de conexión permitió adaptar el comportamiento de estos procesos de acuerdo al estado de conexión con el drone.

Connection Manager administra el estado de conexión del drone, el cual es accesible a través del método connectionState en la API pública de la clase ARDrone. Los posibles estados de conexión son: connection_CONNECTED, connection_CONNECTING y connection_DISCONNECTED.

Esta componente corresponde a una clase que crea un proceso encargado de administrar la conexión entre el drone y el cliente, esto es: inicializarla, detectar cuando se ha perdido y reconectar automáticamente. Además se encarga de mantener la conexión activa mediante el constante envío de paquetes al puerto de comandos. Cada una de las funciones anteriores es descrita detalladamente en los siguientes párrafos:

Inicializar la conexión con el drone: Equivale a indicarle al drone que comience o retome la transmisión de navdata en modo Demo, y además, a reiniciar el número de secuencia de los comandos AT que se envían desde el cliente. La documentación de Parrot indica que para ordenarle al drone que comience a emitir navdata, basta con enviar un mensaje con «algunos bytes» al puerto de navdata, pero no menciona cuáles son esos bytes. A través de investigación adicional, que incluyó la revisión de distintas APIs para comunicarse con el drone, se determinó éstos deben ser el char home (0xR1) seguido del char null (0xR0), ambos en un mismo mensaje.

LECCIÓN APRENDIDA

> Para que el drone comience a transmitir datos de navegación, es necesario enviar específicamente los bytes (0xR1) (0xR0) juntos en un solo mensaje al puerto de navdata.

El cliente esperará 20 milisegundos para recibir de un paquete de navegación. Si nada es recibido el intento de conexión actual se considerará fallido y se realizará un nuevo intento. En caso de que un paquete sea recepcionado, el cliente utilizará la información contenida en su interior para determinar si el drone se encuentra o no en modo Boostrap. Según sea el caso, ocurrirá una de las siguientes opciones:

1. El drone se encuentra en modo Bootstrap, lo que quiere decir que los paquetes enviados sólo contendrán su número de secuencia y el status del drone. Esto ocurrirá sólo si el drone no ha sido configurado aún por ninguna aplicación cliente para enviar navdata en modo Demo o Full. Sólo mientras el drone se encuentre en modo Bootstrap será posible configurarlo para que envíe navdata, ya sea en modo Demo o Full. La API en Pharo sólo soporta el modo

Demo. El comando de configuración se envía al interior de un AT command apropiado al puerto de comandos.

2. El drone ya sido configurado anteriormente por una aplicación cliente para enviar navdata en modo Demo o Full. En este caso es imposible hacer que el drone vuelva al modo Bootstrap. El drone continuará enviando navdata en el modo ya configurado. Si dicho modo resulta ser Full, la API arrojará la excepción DroneConfigurationAttemptFailed, considerará el intento de conexión actual como fallido y además detendrá los posibles intentos de conexión automáticos posteriores. En este caso el usuario deberá reiniciar el drone, desconectando y volviendo a conectar su batería, para así permitirle entrar en modo Bootstrap.

LECCIÓN APRENDIDA

Sólo es posible establecer el modo de envío de datos de navegación, ya sea en modo Demo o Full, cuando el drone se encuentra en modo Bootstrap. Si el cliente u otra aplicación ha configurado anteriormente un modo de envío distinto al deseado, no será posible volver a modo Bootstrap para configurarlo. En este caso la única solución será reiniciar el drone.

Detectar cuando la conexión con el drone se ha perdido: Si el drone no recibe paquetes por más de 2 segundos dejará de transmitir datos al cliente y entrará en modo de vuelo «hover», a una altura de un metro desde el piso. En este caso el cliente deberá volver a inicializar la conexión. Para cubrir este caso, junto con aquel en que la recepción de navdata es interrumpida por interrupciones en la señal inalámbrica, el cliente implementa en su interior un mecanismo tipo watchdog que se encarga de detectar y alertar los eventos de pérdida de conexión con el drone. El cliente considerará la conexión con el drone como perdida, habiendo transcurrido dos segundos desde que el último paquete de navegación fue recibido. Para cumplir con lo anterior, el Connection Manager contiene una variable cuyo valor se establece en dos segundos cada vez que un paquete de navegación arriba al cliente, y que decrece a medida que transcurre tiempo desde la recepción del último paquete de navdata. Si esta variable llega a ser cero, se arrojará la excepción DroneConnectionInterrupted. Según la configuración por defecto de la API, este evento dará pie al inicio de un ciclo de intentos de reconexión descrito inmediatamente a continuación.

Intentar reconectar automáticamente cuando sea necesario: Cada vez que la conexión con el drone haya sido interrumpida, es decir, cuando se arroje la excepción DroneConnectionInterrupted, Connection Manager intentará reestablecer la conexión automáticamente, haciendo intentos consecutivos de inicialización de conexión, hasta que se alcance un límite de tiempo establecido. Este límite tiene un valor predefinido de 5 segundos, sin embargo, es posible cambiarlo usando el mensaje de la API pública configReconnectTimeout. El valor de esta variable puede ser establecido incluso en cero, para el caso en que el usuario no desee que el sistema realice ningún intento de reconexión automático. La conexión se considera

definitivamente perdida, en el lado del cliente, si el límite de tiempo de reconexión vence. En este caso, se arroja una excepción de tipo DroneConnectionTimeout.

Cada intento de conexión realizado equivale a un intento de inicialización de conexión, tal como se describe algunos párrafos atrás.

Mantener la conexión activa enviando mensajes permanentemente al puerto de comandos: Si el drone no recibe paquetes desde el cliente por más de 50 milisegundos, dará alerta por watchdog problem. En este caso el cliente deberá enviar el comando AT*COMWD para reestablecer el watchdog del drone, y además deberá resetear su propio contador del número de secuencia de comandos AT que envía. Para evitar que el drone alerte por watchdog problem, el cliente envía permanentemente el comando AT*COMWD cada 40 milisegundos al drone. En caso de que por problemas en la señal, la alerta por watchdog problem ocurra de todos modos, el cliente reaccionará ante este evento reestableciendo su contador de número de secuencia para comandos AT, y continuando con el envío del comando AT*COMWD cada 40 milisegundos.

7.3.2. Navdata Manager

Es una clase que se encarga de recibir y decodificar los datos de navegación provenientes del drone. Para ello crea un proceso que cumple estas tareas enviando y recibiendo mensajes a través del adaptador para navdata. Hay que recordar que el stream de navdata contiene datos acerca de la sesión de vuelo, y no acerca de la configuración interna del drone.

Este proceso se ejecuta en un ciclo infinito de frecuencia aproximada 15 Hertz (coincidiendo con la frecuencia de envío de paquetes del drone en modo Demo). En cada iteración el contenido del paquete recibido es utilizado para actualizar los valores de un objeto representativo del datagrama, que contiene en su interior información respecto del estado interno del drone, de su posición y velocidad (Opción Demo), y de la detección de tags (Opción Vision Detect). Este objeto datagrama es utilizado inicialmente tan sólo para almacenar la representación, en el lado del cliente, del estado interno del drone. Sin embargo, para mayor flexibilidad, se provee al usuario un método en la API pública, que permite definir un bloque de código tipo callback, que se ejecutará al interior de cada ciclo y que deberá recibir como argumento al objeto que representa el datagrama recibido. De este modo, el usuario puede trabajar directamente con el objeto datagrama, y así conocer todos sus campos, incluso aquellos que no son expuestos a través de métodos de la API pública.

Cuando la conexión con el drone sufre una interrupción, el proceso receptor de navdata es suspendido, y permanece así durante el tiempo que demore el reestablecimiento de la comunicación. Cuando la conexión no puede recuperarse, este proceso se termina, al igual que ocurre cuando el usuario de la API decide finalizar la sesión con el drone (mediante el envío del mensaje público ARDrone >> terminateSession).

7.3.3. Command Manager

Command Manager crea y administra dos procesos encargados de enviar AT commands a través del adaptador para comandos.

El primer proceso envía comandos de movimiento al drone de modo que la experiencia del usuario que los utiliza sea fluida. Aunque para el usuario final de la API el detalle sobre la estructura de los comandos de movimiento no sea relevante, lo es para quién desea comprender su funcionamiento interno y ciertas peculiaridades asociadas al modo en que el drone reacciona ante ellos.

Los comandos de movimiento corresponden a AT commands, específicamente a progressive commands, y tienen la siguiente estructura:

AT*PCMD=605,1,0,0,0,0<**LF**>

Su prefijo es siempre «**AT***PCMD», y sus argumentos, de izquierda a derecha, son los siguientes:

- Argumento 1: número de secuencia
- Argumento 2: flag que habilita (1) o deshabilita (0) el uso de progressive commands
- Argumentos 3 y 4: inclinaciones deseadas para roll y pitch
- Argumentos 5 y 6: velocidades deseadas para gaz y yaw

Los valores de inclinación y velocidad contenidos en estos comandos son determinados por los valores contenidos en el objeto Motion State, que representa en cada momento el estado de movimiento deseado para el drone. Motion State incorpora valores para roll, pitch, gaz y yaw.

Utilizando este objeto intermedio se evita que el usuario de la API sea responsable del envío directo de comandos de movimiento, liberándolo así de la necesidad de controlar la tasa de envío, problemas de retraso en el envío, etc. El usuario sólo podrá modificar el estado del objeto Motion State, y será el proceso que envía comandos de movimiento aquel que decida cuáles comandos enviar y cuándo hacerlo, basándose en el estado del objeto Motion State.

Las reglas que sigue este proceso para enviar comandos de movimiento son las siguientes:

- Los comandos de movimiento se enviarán sólo mientras el drone se encuentre volando. Este dato es obtenido a partir de navdata.
- Mientras el objeto Motion State contenga en su interior algún valor no nulo, se enviarán comandos de movimiento a una tasa de 30 Hz (frecuencia sugerida por la documentación de Parrot para una experiencia de vuelo fluida). Estos comandos contendrán los valores deseados de inclinación y velocidad, y contendrán en su argumento 2 (flag de comandos progresivos) un valor igual a 1.
- Cuando el objeto Motion State pase de contener algún valor no nulo a contener sólo valores nulos, se enviarán 3 comandos de estabilización al drone, es decir 3 comandos de movimiento que tengan valor 0 para el flag de comandos progresivos. Éstos harán que el drone se

estabilice más rápidamente, al forzarlo a entrar en modo hover. Se envían 3 comandos, y no sólo uno, para disminuir la probabilidad de que el drone no reciba la instrucción de estabilización debido a problemas en la comunicación.

- Durante el resto del tiempo que el objeto Motion State sólo contenga valores nulos, no se enviarán comandos de control de movimiento al drone, de este modo se prioriza el envío de comandos que no sean de movimiento.

La frecuencia de envío de los comandos de movimiento, mientras deban enviarse, intenta ser siempre la misma: 1 comando cada 30 milisegundos. Esta frecuencia se ve ligeramente disminuida cuando el segundo proceso de esta clase utiliza el mismo adaptador UDP para enviar sus propios comandos. No obstante la tasa de salida de los comandos enviados por el segundo proceso es muy inferior a la tasa de envío de los comandos de movimiento, y por ende su efecto sobre la frecuencia de envío del primero, es despreciable.

Controlar correctamente los momentos en que el drone entra en estado hover es muy importante, dado que mientras se encuentre en este estado, su firmware utilizará un algoritmo de estabilización de vuelo que se vale de las imágenes provenientes de la cámara inferior. Cubrir esta cámara de manera que el drone no pueda utilizar sus imágenes para estabilizarse, tiene un impacto dramático sobre la capacidad del drone de mantenerse volando sobre un punto fijo en el piso. Se obtiene un efecto similar si el cliente se mantiene enviando comandos de movimiento al drone durante todo el tiempo si éstos no establecen el flag de comandos progresivos en cero, incluso si contienen sólo valores nulos para las inclinaciones y velocidades del drone.

LECCIÓN APRENDIDA

Enviar al drone comandos de movimiento que contengan sólo valores cero para *yaw*, *pitch*, *roll* y *yaz*, pero que establezcan un valor distinto de cero para el flag de comandos progresivos, impide que el drone entre en modo hover, lo que afecta negativamente su capacidad de mantenerse estable sobre un punto definido.

Así, existen dos maneras de hacer que el drone entre en modo hover. La primera y más eficiente es forzarlo enviándole un comando de movimiento que contenga un valor cero para el flag de comandos progresivos. La segunda es cesar el envío de comandos de movimiento. En este caso el drone tardará un par de segundos en entrar en modo hover.

LECCIÓN APRENDIDA

La manera más eficiente de hacer que el drone entre en modo hover, es enviarle un comando de movimiento con valor cero para el flag de comandos progresivos.

El drone informa que su estado es «volando» desde el momento que se despega del piso, sin embargo se recomienda al programador evitar enviar comandos de movimiento durante el tiempo que dure el despegue, es decir, desde que el drone abandona el suelo hasta que estabiliza su vuelo en modo hover a aproximadamente un metro de altura. Enviar comandos de movimiento durante el despegue provoca inestabilidad en el vuelo del drone.

LECCIÓN APRENDIDA

Se recomienda no enviar comandos de movimiento durante el proceso de despegue del drone.

El segundo proceso creado por Command Manager se encarga de enviar el resto de comandos AT que no son de movimiento, por ejemplo aquellos que animan los LEDs del drone, o aquellos modifican su configuración interna. Para cada comando distinto, se ha implementado un mensaje en la API de Command Manager, que se encarga de crear el comando correctamente y de agregarlo a la cola de comandos. Mediante estos mensajes, los comandos son enviados cada vez que el usuario lo solicite o cuando sea necesario para la API, de acuerdo al estado interno del drone. Por ejemplo, para intentar reconectar, o para mantener la conexión viva mediante el envío constante del comando que resetea el watchdog del drone.

El envío de estos comandos, que no son de movimiento, no es directo. Éstos se ordenan utilizando una cola compartida y protegida con un mecanismo de exclusión mutua para que múltiples procesos ligeros encolen comandos. La función del proceso se reduce entonces a desencolar dichos comandos y enviarlos utilizando el adaptador para comandos.

Los mensajes implementados para enviar comandos que no son de movimiento, se han clasificado utilizando protocolos, cuyo nombre hace referencia al tipo del comando que envíen. Como los comandos que modifican la configuración interna del drone son muy numerosos, se les ha dispuesto en protocolos distintos, que los ordenan según una clasificación propuesta por Parrot en la documentación oficial. En este trabajo se utilizan principalmente comandos de configuración asociados a:

- El modo de vuelo del drone (clasificación CONTROL según Parrot)
- La detección de tags (DETECT)

- El contenido de los datos de navegación y requerimientos para el envío del archivo de configuración interna del drone (GENERAL)
- La animación de LEDs (LEDS)
- La configuración de video (VIDEO)
- Al establecimiento de variables apropiadas para el modo Multiconfiguración del drone (CUSTOM)

Notas sobre la Multiconfiguración El drone es capaz de almacenar múltiples valores para sus parámetros de configuración, permitiendo así que distintos usuarios puedan utilizar el mismo drone sin necesidad de establecer al inicio todos los valores de configuración requeridos. Para efectos de la API en Pharo, basta con establecer que una configuración específica corresponde a un set de valores establecidos para la configuración, y que se asocia a una combinación de 3 identificadores: uno de aplicación, uno de usuario y uno de sesión.

Cada vez que la API establece conexión con el drone, le envía los comandos apropiados para establecer un identificador de aplicación, de usuario y de sesión; con ello el drone cargará el archivo de configuración apropiado en caso de existir, y almacenará los cambios efectuados sobre su configuración en el archivo correspondiente. Los identificadores utilizados por la API se encuentran establecidos en el código de los mensajes pertenecientes al protocolo «api-configuration CUSTOM» de la clase ARDCommandManager.

La utilización del soporte para multiconfiguración no es opcional si se desea modificar la configuración interna del drone, ya que para que surtan efectos los comandos AT*CONFIG que modifican ciertos parámetros (en especial aquellos asociados al video), es requisito enviar inmediatamente antes de cada uno de ellos un comando de encabezado AT*CONFIG_IDS que incluya los identificadores asociados a la aplicación, usuario y sesión.

7.3.4. Configuration Manager

Dado que modificar la configuración interna del drone es una tarea necesaria, corroborar que los comandos de configuración enviados hayan surtido efecto surge como una necesidad. Esto es así porque existen variables de configuración cuyo estado no puede ser inferido directamente a partir del comportamiento del drone. Un ejemplo es la configuración del entorno en el que el drone está volando: «indoor» u «outdoor»; esta configuración repercute sobre las velocidades e inclinaciones máximas del drone, sin embargo, es muy difícil inferir su valor actual con sólo observar el vuelo del drone.

Debido a lo anterior, se ha incluido en la API el módulo Configuration Manager, cuya labor es proveer un mecanismo sencillo para consultar los valores actuales de la configuración interna del drone.

Configuration Manager ofrece escencialmente dos métodos en su API: el primero sirve para recuperar el archivo de configuración desde el drone en un momento específico y el segundo para consultar el valor de un parámetro de configuración específico. El archivo de configuración, que

el drone envía a través del canal TCP puerto 5559, consiste en un string multilínea en formato ASCII que contiene todos los parámetros de configuración del drone y sus correspondientes valores. Cada parámetro ocupa una linea con formato *Parameter_name = Parameter value.*

```
...
general:ardrone_name     = RyChCopter
general:flying_time      = 758
general:navdata_demo     = TRUE
general:navdata_options = 105971713
general:com_watchdog     = 2
general:video_enable     = TRUE
general:vision_enable    = TRUE
...
```

Tabla 3: Extracto del archivo de configuración enviado por el drone

Mediante pruebas de uso se descubrió que en ciertas ocasiones el drone no envía el archivo de configuración completo, sino que envía un string cuyo inicio es válido, pero que se encuentra interrumpido en algún punto posterior arbitrario. Es por ello que Configuration Manager valida el string recibido, chequeando que contenga el valor asociado al parámetro «rescue:rescue», correspondiente al parámetro de la última línea.

La API implementa en Connection Manager un mecanismo automático de obtención del archivo de configuración que se ejecuta cada 500 milisegundos. Los datos contenidos en este archivo son almacenados al interior de Configuration Manager, de modo que el cliente pueda consultarlos libremente en cualquier momento. El evento de obtención del archivo de configuración por primera vez luego de que la conexión con el drone ha sido establecida (o recuperada automáticamente) es anunciado mediante el announcement DroneConfigurationFileReceivedForTheFirstTime. Por otro lado, sin importar si el archivo ya ha sido recibido antes, cada evento de recepción de un nuevo archivo es anunciado a través del announcement DroneConfigurationFileReceived.

Notas sobre la configuración de los modos de vuelo El drone soporta 3 modos de vuelo preestablecidos:

- «FREE FLIGHT» es el modo de vuelo normal, el drone se mueve sólo de acuerdo a los comandos de movimiento recibidos.
- «HOVER ON TOP OF ROUNDEL» el drone se mueve siguiendo la posición del tag «oriented roundel» a través de su cámara vertical. (no se utiliza en este trabajo)
- «HOVER ON TOP OF ORIENTED ROUNDEL» idéntico al modo anterior, excepto por el hecho de que el drone se mantiene apuntando en la misma dirección que apunta el tag.

Los modos de vuelo pueden establecerse modificando el parámetro de configuración «control:flying_mode». No obstante, para habilitar los modos «hover on top of...» el programador deberá configurar adicionalmente la detección de tags de manera apropiada. Esto implica establecer el parámetro de configuracioń «detect:detect_type» en el modo «ORIENTED_COCARDE_BW» y sólo después de esto modificar el parámetro «control:flying_mode».

En todos estos modos de vuelo el drone continúa obedeciendo los comandos de control de movimiento que recibe.

Notas sobre la configuración de detección de tags En versiones iniciales de la API de Parrot para controlar al drone, la configuración de detección de tags dependía sólo del parámetro de configuración: «detect:detect_type». Las opciones para éste se encuentran almacenadas como una enumeración llamada CAD_TYPE, en el archivo ardrone_api.h que forma parte del SDK de Parrot. De las 15 opciones disponibles para este valor, a la fecha sólo 4 son válidas:

- NONE: ningún tag es detectado
- VISION_V2: el Shell tag es detectado mediante la cámara frontal. Es la opción por defecto
- MULTIPLE_DETECTION_MODE: permite detectar múltiples tags a la vez en ambas cámaras
- ORIENTED_COCARDE_BW: sólo es necesario para habilitar el modo de vuelo «HOVER ON TOP OF (ORIENTED) ROUNDEL»

En versiones posteriores de su API, Parrot introduce los parámetros de configuración «detect:detections_select_(v, h)» que determinan qué tipos de tags serán detectados en las cámaras vertical y horizontal. En la documentación de la API de Parrot, se hace hincapié en no habilitar la detección de un mismo tipo de tag en dos cámaras distintas, ya que esto puede producir resultados inesperados. Además, para habilitar la detección múltiple de tags, recomienda establecer primero el parámetro de configuración «detect:detect_type» en modo de detección múltiple (MULTIPLE_DETECTION_MODE) y después definir los tipos de tags que se desea detectar en cada cámara alterando el valor de los parámetros «detect:detections_select_(v, h)».

Lo que Parrot no menciona, y que puede producir confusión en el programador, es que al momento de habilitar la detección de un tipo de tag en alguna cámara utilizando «detect:detections_select_(v, h)», el valor para el parámetro «detect:detect_type» cambia automáticamente a MULTIPLE_DETECTION_MODE.

LECCIÓN APRENDIDA

Cuando se configuran detecciones de tags específicos alterando los parámetros de configuración «detect:detections_select_(v, h)», el parámetro «detect:detect_type» es establecido automáticamente en modo de detección múltiple.

Por otro lado ocurre también que cuando se establece el valor ORIENTED_COCARDE_BW para el parámetro «detect:detect_type», los valores para «detect:detections_select_(v, h)» son modificados automáticamente por el firmware del drone a valores apropiados para que

este modo de vuelo funcione, por lo tanto, en este caso el programador no deberá alterar la configuración de este parámetro, ya que si lo hace, el drone entrará en modo de detección múltiple: MULTIPLE_DETECTION_MODE.

LECCIÓN APRENDIDA

Cuando se configura el parámetro «detect:detect_type» con el valor ORIENTED_COCARDE_BW, los valores para el parámetro «detect:detections_select_(v, h)» son modificados automáticamente por el firmware del drone. El programador no deberá modificar estos valores manualmente si desea conservar el modo de detección ORIENTED_COCARDE_BW.

Existe un último punto relacionado al modo de detección ORIENTED_COCARDE_BW. La información que el drone publica sobre cada tag detectado incluye su cámara de origen. Normalmente se utiliza el valor 0 para referirse a la cámara horizontal y 1 para la vertical. Sin embargo la cámara de origen para un tag oriented roundel detectado en la cámara vertical del drone valdrá 2 siempre que el parámetro «detect:detect_type» se encuentre configurado con el valor ORIENTED_COCARDE_BW.

LECCIÓN APRENDIDA

La camara de origen de un tag Oriented Roundel detectado por la cámara vertical valdrá 2 siempre que el parámetro «detect:detect_type» se encuentre configurado con el valor ORIENTED_COCARDE_BW.

7.4. Drone Internal State

Esta componente representa la existencia de dos instancias de objetos datagrama, que almacenan el contenido de el último y el penúltimo paquete de navdata recibido. Por ende, mientras sólo uno de ellos representa el último estado interno informado por el drone en cada momento, ambos pueden ser utilizados para detectar cambios ocurridos en dicho estado. Naturalmente el contenido de estos objetos es actualizado al momento de recibir cada paquete de navegación.

Es relevante que el estado interno del drone sea conocido por la librería en cada momento, ya que el comportamiento de los procesos que envían comandos de control y configuración necesitan adecuar su comportamiento de acuerdo al estado del robot. Por ejemplo, el proceso que envía

comandos de movimiento al drone sólo debe hacerlo mientras el drone se encuentre volando, por lo que necesita ser notificado cada vez que el drone despega o aterriza.

Adicionalmente a las clases que representan los datagramas recibidos, se ha creado una clase dedicada al procesamiento de la información ligada a la detección de tags. A continuación se la describe en detalle.

Vision Detector Analyst Esta clase facilita la consulta de información sobre detección de tags, proveyendo métodos que responden las consultas más habituales que podría tener un usuario de la API. Además, disminuye el ruido relacionado al número de tags detectados momento a momento. El drone informa sobre el número de tags detectados en cada paquete de navdata que emite. Aunque la mayor parte de estos paquetes indica un número acertado de tags detectados, entre éstos se encuentran algunos que informan un número inferior al real. Para solucionar este problema, se ofrece al usuario dos mensajes para saber si un tag está siendo detectado momento a momento:

- El mensaje detected retornará exactamente lo informado en los paquetes de navegación recibidos. Por ende es suceptible a entregar falsos negativos.
- El mensaje detectedWeighted utiliza un arreglo que almacena la información sobre el tag recibida en los últimos 3 datagramas enviados por el drone, y usa estos valores para calcular un promedio de detección. Esta función es menos propensa a entregar falsos negativos, pero por otro lado introduce un pequeño retraso en el mecanismo de detección.

Como ya se ha dicho, el drone tiene la capacidad de detectar a la vez múltiples tags mediante sus dos cámaras, sin embargo, existen limitaciones asociadas, que se detallarán en los siguientes párrafos.

La primera limitación, documentada por Parrot en su guía para el desarrollador, indica que no se debe habilitar la detección de un mismo tipo de tag en ambas cámaras, ya que ésto podría causar problemas en los algoritmos. Se hicieron diversas pruebas en las que esta advertencia fue ignorada y no surgió ningún problema, no obstante se optó por respetar esta advertencia y habilitar por defecto sólo la detección del tag Oriented Roundel en la cámara vertical y la del Shell tag en la cámara horizontal.

La siguiente limitación fue descubierta en pruebas de uso, y se refiere a que aunque el drone es incapaz de detectar múltiples tags de tipo Shell Tag frente a una misma cámara, siempre reporta la presencia de sólo uno de ellos. Esto no ocurre con el tag Oriented Roundel, ya que hasta cuatro marcas de este tipo pueden ser detectadas por una sola cámara a la vez. Por otro lado, si se activa la detección de ambos tags en una misma cámara, el drone reportará la presencia de a lo más un Shell Tag y de máximo 4 Oriented Roundel.

Finalmente, surge una última complicación derivada de que el drone presenta los datos de detección de tags separadamente en arreglos de 4 elementos. Por ejemplo, al interior de cada paquete de navegación, dentro de la opción «Vision», existe un arreglo de largo 4 llamado «distance» que contiene la distancia a la que se encuentran cada uno de los tags detectados por el drone a la vez. Además del arreglo distance, existen otros arreglos que indican posición, orientación, longitud, altura, cámara de origen y el tipo de los tags detectados. Los datos asociados

a un mismo tag detectado ocuparán un único índice dentro de cada uno de estos arreglos. No obstante, el índice asociado a cada tag detectado no se mantiene constante en el tiempo, puede variar en caso de que alguno de ellos deje de ser detectado (aunque sea por un período muy breve de tiempo). Esto ocurre porque el drone siempre llena los arreglos ocupando los índices menores primero, incluso si esto implica que el índice de un tag que está siendo detectado a la vez que algún otro, cambie abruptamente al momento en que su compañero deja de ser detectado. Producto de lo anterior, se ha tomado la decisión de limitar el soporte ofrecido por la API en Pharo para la detección de tags, estableciendo que ésta soporta oficialmente sólo un tag detectado en cada cámara. Por lo tanto, se soportará la detección de dos tags de distinto tipo a la vez sólo para el caso en que estos sean detectados en cámaras distintas.

Cabe mencionar que el drone antepone en orden dentro de los arreglos al Shell tag respecto del Oriented roundel.

LECCIÓN APRENDIDA

El drone entrega información acerca de los tags detectados mediante arreglos de 4 elementos, en los que cada tag detectado ocupa el mismo índice en todos los arreglos. Sin embargo el índice asociado a cada tag puede cambiar dependiendo del número de tags detectados en cada momento.

La manera en que la API en Pharo identifica el tipo de tag asociado a cada detección no utiliza la información contenida en el arreglo «type» de la opción «Visión». Esto porque en modo de detección múltiple este valor sólo varía en función de la cámara de origen, pero no en función del tipo de tag detectado. Se utiliza en cambio el ángulo de orientación del tag detectado para identificar de qué tipo de tag se trata. Si este valor es exactamente cero, se asume que se trata de un Shell tag. En caso contrario, se trata de un Oriented roundel. Como los ángulos de orientación se entregan en formato float de 32 bits, es razonable asumir que para el caso del tag Oriented roundel casi nunca se informará un cero absoluto como su ángulo de orientación.
Considerando lo ya expuesto sobre la detección de tags, se decidió incorporar a la API en Pharo una clase llamada ARDDetectedTag que representa un tag de un tipo definido (puede ser Shell u Oriented Roundel) que puede ser detectado por sólo una de las cámaras del drone. Así, una instancia de esta clase tendrá un «tipo» asociado, que dependerá del tipo de tag y de la camara donde se espera su detección. La API define dos tipos para los objetos de esta clase:

- detectedTag_SH_IN_CH: Shell tag detectado en la camara horizontal.
- detectedTag_OR_IN_CV: Oriented roundel detectado en la camara vertical.

La clase ARDDetectedTag es responsable de procesar la información relacionada a la detección de este tag: su posición, orientación, estado de la detección, etc.

El Vision Detector Analyst contiene dos instancias de la clase anterior, una por cada tipo. Su responsabilidad es extraer la información necesaria para actualizar ambas instancias a partir de los datos de navegación crudos recibidos en cada paquete.

7.5. State Manager

State Manager representa la capacidad de la API de anunciar cuando el estado interno del drone sufre cambios relevantes y cuando los paquetes de navegación son recibidos. Esto último es especialmente útil para determinar el estado de la conexión con el drone.

Para detectar cambios en el estado interno del drone, se comparan los estados informados en los dos últimos datagramas recibidos. Posteriormente se notifica si existen diferencias específicas entre ambos.

Entre otros, los siguientes anuncios se han definido en la API:

- DroneLanded: el drone ha aterrizado.
- DroneTookOff: el drone ha despegado.
- DroneInEmergencyMode: el drone ha entrado en modo de emergencia.
- DroneConnectionStart: se ha establecido conexión con el drone.
- DroneConnectionStop: imposible establecer conexión. Esto ocurre una vez que el plazo para realizar intentos de reconexión ha vencido.
- DroneConnecting: la conexión se ha interrumpido. Los intentos de reconexión comienzan.
- DroneConfigurationFileReceived: se ha recibido el archivo de configuración desde el drone, la API intenta traer este archivo cada 500 milisegundos.
- DroneConfigurationFileReceivedForTheFirstTime: se ha recibido el archivo el configuración por primera vez desde que se (re)estableció la conexión.
- DroneNavdataPacketReceived: se ha recibido un paquete de navegación. La API intenta recibir un paquete cada 20 milisegundos.

7.6. Drone Motion State

Se refiere a una única clase: ARDMotionState. Ésta contiene variables de instancia que almacenan los ángulos de inclinación (pitch y roll), velocidad angular (yaw) y velocidad vertical deseados para drone. Todos estos valores son de tipo float, pertenecen al rango [-1, 1] y representan un porcentaje del valor máximo configurado para la inclinación o velocidad correspondiente en el drone.

Estas variables de movimiento son enviadas como argumentos al interior de los comandos de control de movimiento. Uno de los procesos de la instancia ARDrone intenta despachar estos comandos cada 30 milisegundos. De esta manera, para controlar los movimientos del drone el usuario deberá modificar el valor de estas variables. Los métodos públicos de la API ofrecen una manera directa de hacer esto.

Este diseño separa las peticiones de control generadas por el usuario, del envío de comandos hacia el drone; lo cual garantiza independencia de la interfaz de control del movimiento con respecto a la API de comunicación. De este modo la API es flexible para ser utilizada con cualquier interfaz de usuario (por ejemplo joystick o teclado).

7.7. Interfaces de usuario

7.7.1. Interfaz ARDUIMotionState

Como una herramienta auxiliar para el usuario de la API, se ha incluido un modelo gráfico que refleja en tiempo real el estado del objeto Motion State. De este modo, devela parcialmente el contenido de los comandos de movimiento que se envían al drone (es parcial ya que no representa los niveles de inclinación o velocidad enviados).

Contar con esta herramienta le permite al programador observar intuitivamente el efecto que tiene su código sobre el objeto Motion State, independientemente del efecto que los comandos de movimiento hayan tenido sobre el drone. Esto es útil porque puede ocurrir que los valores enviados de inclinación o velocidad sean muy discretos, o bien que los comandos de movimiento se envíen por un tiempo muy breve, y que el efecto sobre el drone sea difícilmente observable. Adicionalmente esta interfaz de usuario le permite al programador probar su código sin siquiera hacer despegar al drone, puesto que aunque la API no efectúe el envío de los comandos de movimiento mientras el drone no se encuentre volando, el objeto Motion State cambiará su estado de todos modos, lo cual será reflejado en la interfaz.

La figura 13 muestra la apariencia de esta interfaz. Se identifican dos figuras de cruz, estilo control pad, cada una de las cuales se compone de cuatro cuadrados. Cada par de cuadrados opuestos representa el estado de una variable de inclinación o velocidad. El color azul indica que la variable asociada al par contiene un valor no nulo, mientras que su signo lo define cuál de las figuras del par se encuentra coloreada.

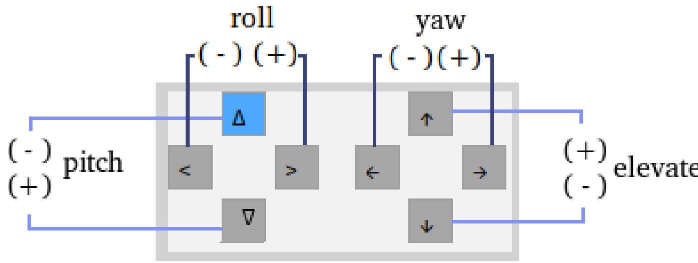


Figura 13: Interfaz de usuario ARDUIMotionState reflejando inclinación hacia abajo de la nariz del drone

7.7.2. Interfaz ARDUIKeyboard

Adicionalmente a la API se ha definido a, la clase **ARDUIKeyboard**, que corresponde a una interfaz de usuario para controlar el drone con el teclado y que se integra con la interfaz ARDUIMotionState recién descrita. En ésta se asignan dos teclas a cada variable de movimiento, una para volverla positiva y otra para volverla negativa. El valor asignado a todas las variables se ha fijado en 0.5, ya sea positivo o negativo. Por ejemplo, al presionar la tecla 'S', el valor de pitch se establecerá en -0.5 y por ende el drone «bajará la nariz» durante todo el tiempo que esta tecla sea presionada. Al dejar de presionarla, la variable pitch volverá a ser 0. Lo que ocurre

entonces es que los eventos del teclado modifican el objeto ARDMotionState, mientras que éste paralelamente es accedido por el proceso encargado de enviar comandos de movimiento al drone.

En las Tablas 4, 5 y 6 se presenta un resumen de los controles para el teclado. En ellas los comandos han sido clasificados por tipo, según produzcan movimiento, piruetas o animaciones LED. Junto a cada comando, se incluye una breve descripción de su efecto en el drone. En la figura 14 se muestra la distribución de estos controles en el teclado. Configurar los comandos según el teclado latinoamericano busca facilitar el uso de esta herramienta a estudiantes del DCC, ante la posibilidad de que alguno deseara continuar este trabajo.

Los comandos de animación LED y los de animación de vuelo (pirueta) requieren la especificación de parámetros de duración y, sólo para LED, de frecuencia. Por otro lado, los comandos de movimiento requieren que se especifique el porcentaje de la velocidad o inclinación máximas asociadas al movimiento. La clase ARDUIKeyboard contiene valores por defecto para todos los parámetros anteriores:

- 2 segundos para la duración de piruetas.
- 5 segundos para la duración de animaciones LED y 2 Hz para su frecuencia.
- 40 % de la velocidad o inclinación máximas para los movimientos.

El usuario puede cambiar estos valores para su propia instancia, utilizando los mensajes al interior del protocolo *configuring* en la clase ARDUIKeyboard.

Un video de un vuelo de prueba, controlado por teclado, utilizando una versión temprana de esta API de comunicación puede ser encontrado en el siguiente link de YouTube: `http://bit.ly/ARDronePharo` (Ver figura 15).

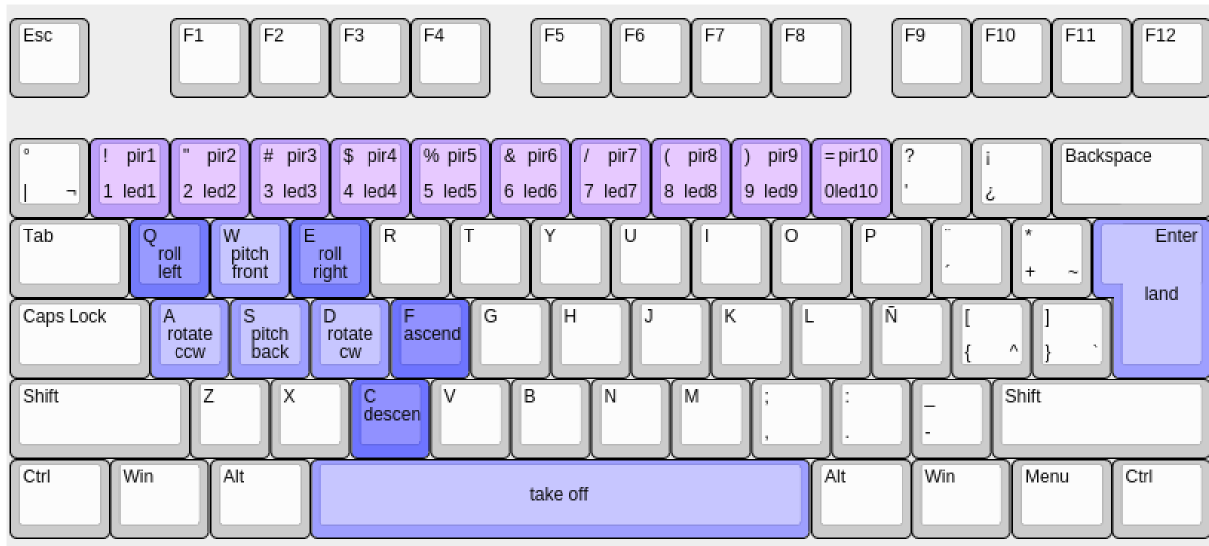


Figura 14: Distribución de comandos del teclado

tecla	acción	descripción
W	**pitch front**	baja la nariz, se mueve hacia adelante
S	**pitch back**	eleva la nariz, se mueve hacia atrás
A	**rotate ccw**	rota en sentido antihorario (observado desde arriba)
D	**rotate cw**	rota en sentido horario (observado desde arriba)
Q	**roll left**	se inclina hacia el lado izquierdo, avanza en esa dirección
E	**roll right**	se inclina hacia el lado derecho, avanza en esa dirección
F	**elevate**	aumenta altura
C	**descend**	disminuye altura
space	**take off**	despega
enter	**land**	aterriza

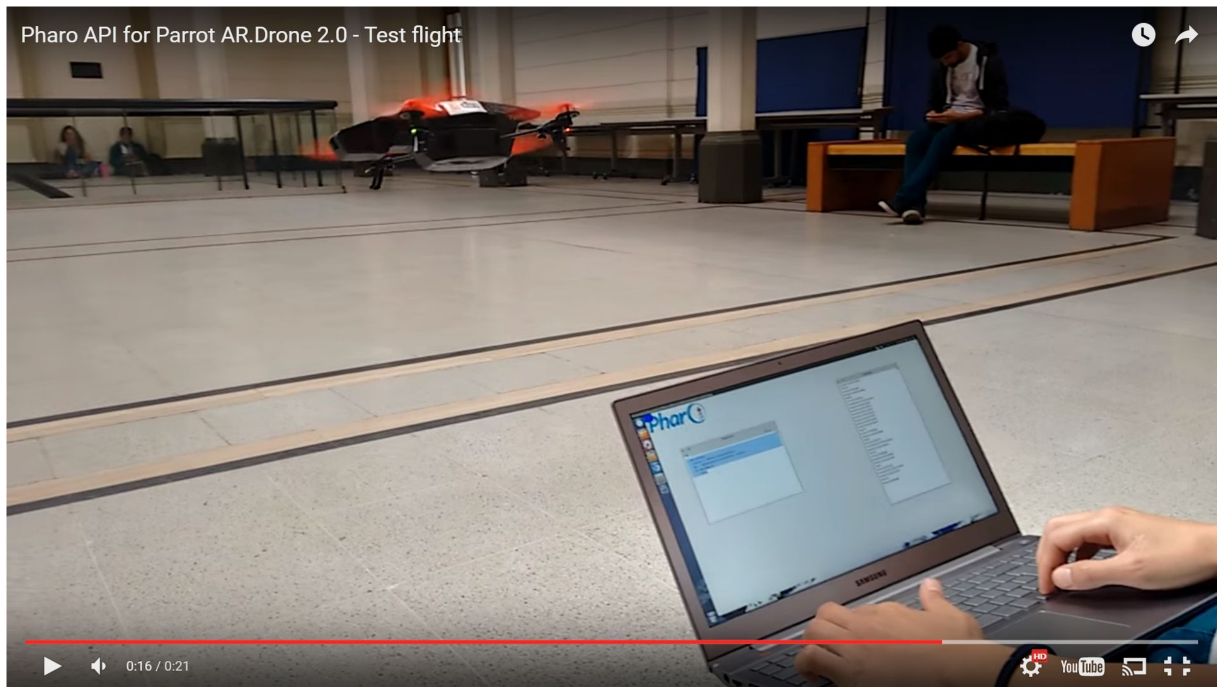
Tabla 4: Controles del teclado para movimiento

n	tecla	pirueta	descripción
1	!	**flip ahead**	giros en 360 grados hacia adelante
2	”	**flip behind**	giros en 360 grados hacia atrás
3	#	**flip left**	giros en 360 grados hacia la izquierda
4	$	**flip right**	giros en 360 grados hacia la derecha
5	%	**wave**	inclinaciones combinadas de pitch y roll que resultan en un movimiento ondulante
6	&	**double phi theta mixed**	dos inclinaciones rápidas frontales seguidas de dos inclinaciones rápidas laterales
7	/	**yaw shake**	rápidos giros de yaw, cortos y alternados en sentido horario y antihorario, que provocan un efecto de sacudida
8	(	**turnaround**	giros de 180 grados en yaw, en sentido horario
9	)	**theta 20 deg yaw 200 deg**	levanta la nariz del drone bruscamente y luego la estabiliza, lo que le hace retroceder
10	=	**yaw dance**	giros alternados y lentos en yaw, en sentido horario y antihorario

Tabla 5: Controles del teclado para Piruetas

n	tecla	animación LED	descripción
1	1	**blink green**	4 leds alternan verde y apagado, al unísono
2	2	**blink red**	4 leds alternan rojo y apagado, al unísono
3	3	**blink orange**	4 leds alternan naranjo y apagado, al unísono
4	4	**fire**	leds frontales parpadean naranjo, leds posteriores permanecen rojos
5	5	**red snake**	4 leds parpadean rojo; uno a la vez, en sentido horario
6	6	**snake green red**	4 leds alternan entre rojo, verde y apagado; desfazados
7	7	**double missile**	leds posteriores prenden naranjo, rojo y se apagan, luego los leds anteriores hacen lo mismo, durante un tiempo los 4 leds permanecen apagados.
8	8	**front left green others red**	led anterior izquierdo permanece verde, el resto permanece rojo
9	9	**front right green others red**	led anterior izquierdo permanece verde, el resto permanece rojo
10	0	**green**	4 leds permanecen verdes

Tabla 6: Controles del teclado para animaciones LED



Video disponible en `http://bit.ly/ARDronePharo`

Figura 15: Captura de video publicado en YouTube que muestra un vuelo de prueba controlado con teclado.

8. Descripción Funcional de la API en Pharo

Numerosas clases componen esta API, sin embargo al usuario promedio le bastará conocer sólo una: la clase ARDrone, ya que dentro suyo se encuentran todos los mensajes públicos necesarios para comunicarse con el drone.

8.1. Resumen de mensajes públicos

Los mensajes públicos asociados a la API en Pharo se han asociado todos a la clase ARDrone. Ya que son muy numerosos se ha decidido no incorporarlos directamente al texto, y en cambio presentarlos clasificados por el rol que cumplen en los Anexos, sección A. No obstante, para dar una visión general al lector, se ofrece a continuación la lista de los roles en los que estos mensajes han sido clasificados.

- Instancia de la clase ARDrone y conexión con el drone
- Control de movimiento
- Animación de LEDs
- Consulta de datos de navegación
- Consulta y modificación de la configuración interna del drone
- Detección de tags
- Configuración de la API y captura de excepciones

8.2. Excepciones asociadas a la API en Pharo (Exceptions de Smalltalk)

La API alerta al programador cuando se producen situaciones de error, utilizando las *Exceptions* de Smalltalk. Las siguientes excepciones son definidas dentro de la API:

- `DroneConfigurationAttemptFailed`: se lanza cuando no es posible configurar el modo de envío de navdata en Modo Demo. Esto ocurrirá cuando una aplicación anterior haya configurado este modo en Modo Debug. No se recomienda capturar esta excepción, ya que la API no será capaz de interpretar los datos recibidos en Modo Debug. El usuario deberá reiniciar el drone para resolver este problema.
- `DroneConnectionAdapterDisconnected`: se lanza cuando el adaptador WiFi del cliente no está conectado a ninguna red.
- `DroneConnectionInterrupted`: se lanza cuando el watchdog del cliente llega a cero. Si el cliente no ha especificado que el valor timeout de reconección sea cero, el sistema comenzará los intentos de reconexión hasta agotar el timeout.
- `DroneConnectionTimeout`: se lanza cuando es imposible establecer comunicación inalámbrica con el drone, luego de haber agotado el tiempo disponible para intentos de reconexión.

Es importante notar que existe un método en la API pública, en la clase ARDrone, que permite al usuario capturar las Excepciones anteriores, definiendo un bloque que se ejecutará cada vez que la Excepción se lance. Este método es
`ARDrone >> addHandlerFor: anExceptionClass do: aBlock`

8.3. Anuncios asociados a la API en Pharo (Announcements de Smalltalk)

La API contempla la creación de varios Announcements de Smalltalk que permiten la comunicación entre distintos procesos de la API. Para mayor detalle ver la sección 7.5

- DroneLanded
- DroneTookOff
- DroneInEmergencyMode
- DroneConnectionStart
- DroneConnectionStop
- DroneConnecting
- DroneConfigurationFileReceived
- DroneConfigurationFileReceivedForTheFirstTime
- DroneNavdataPacketReceived

8.4. Indicaciones de uso y ejemplo demostrativo

Para usar esta API es necesario obtener en primer lugar instanciar la clase ARDrone, a la que luego podrán enviársele todos los mensajes ya especificados. Todos los protocolos para establecer comunicación con el drone se ejecutan automáticamente. El único requisito importante para el usuario es que debe enviar el mensaje terminate a su instancia de ARDrone cuando haya finalizado la ejecución de su programa. En caso de no hacerlo, los procesos asociados a la instancia continuarán ejecutándose en segundo plano.

A continuación se muestra un breve ejemplo que expone el modo en que esta API debe ser utilizada.

```
01   | arDrone |
02   arDrone := ARDrone uniqueInstance.
03   arDrone startSession.
```

Como ya se indicó, el primer paso es instanciar la clase ARDrone. Debido a que ésta implementa el patrón *singleton*, la instancia deberá obtenerse a través del mensaje `uniqueInstance`. Para iniciar los procesos y establecer comunicación con el drone, se envía el mensaje `startSession`.

```
04   arDrone setNavdataReceiverCallback:
05   [ :datagram | Transcript show: datagram asString; cr ].
```

Las líneas anteriores especifican un comportamiento asociado a la recepción de datos de navegación. En particular determinan que cada datagrama que sea recibido, se imprima en el *Transcript*.

```
06   arDrone
07      animateLEDs: ARDCommand ledAnimation_FIRE
08      frequency: 2.5
09      duration: 5
```

El bloque anterior ilustra el envío del comando de animación de luces LED. Cabe hacer notar que el argumento que especifica el tipo de animación, corresponde a una enumeración, cuyos valores poseen nombres intuitivos y pueden recuperarse como variables estáticas de la clase *ARDCommand*.

```
10   arDrone takeOff.
11   (Delay forSeconds: 5) wait.
12   arDrone pitch: -0.5.
13   (Delay forSeconds: 1) wait.
14   arDrone pitch: 0.
15   (Delay forSeconds: 1) wait.
16   arDrone land.
17   (Delay forSeconds: 5) wait.
```

El bloque de código comprendido entre las líneas 10 y 17 manipula los movimientos del drone. En este ejemplo se hace al drone despegar, y luego de 5 segundos se le inclina frontalmente de modo que avance durante un segundo, al cabo del cual se estabiliza la inclinación frontal enviando el mensaje `pitch` con parámetro cero. Un segundo más tarde se envía la instrucción de aterrizar.

```
18   arDrone terminateSession.
```

Para finalizar se envía el mensaje `terminateSession`, y de esta manera se finaliza la ejecución de los procesos asociados a la instancia.

8.5. Disponibilidad y requisitos de la API en Pharo

El código de la API se encuentra alojado en un repositorio Smalltalkhub, bajo el nombre de proyecto «ArDronePharo». Su URL es:

http://smalltalkhub.com/#!/~CaroHernandez/ArDronePharo/

Es necesario instalar previamente en la imagen de Pharo la librería OSProcess. Esto puede hacerse ejecutando en un playground el siguiente comando:

```
Gofer new
        squeaksource3: 'coral';
        package: 'OSProcess';
        load.
```

Aquí concluye la descripción de la API en Pharo para comunicación con el AR.Drone. En seguida se da lugar a la descripción del Puente de comunicación necesario para utilizar esta API desde el lenguaje de programación para robots LRP.

9. Puente entre LRP y la API en Pharo para el AR.Drone

LRP fue diseñado como un lenguaje desacoplado de los sistemas robóticos con los que interactúa, lo que le permite potencialmente ser utilizado en cualquier robot para el cual exista una API de comunicación. No obstante, esto impone la necesidad de crear para cada robot distinto, una aplicación tipo puente que adapte los métodos de comunicación disponibles en la API y que solucione los problemas que surjan por la incompatibilidad de la naturaleza «live» de LRP, con la probable naturaleza estática de la API original.

El puente hace posible que LRP tenga acceso a una serie de métodos para comunicarse con el robot, esto se logra mediante la adaptación de los métodos provistos por la API para tal fin. Así, el puente corresponde a un caso del patrón de diseño *«Adapter»*. Desde LRP los métodos del puente son accedidos siempre a través de una pseudovariable denominada `robot`.

El diseño de LRP contempla la existencia de este puente y facilita su creación al incluir una clase abstracta denominada `LRPAbstractBridge` que contiene la estructura y la lógica requeridas por el lenguaje. Para crear un puente, el usuario simplemente deberá concretizar esta clase y en su interior implementar para un método denominado `openUIFor:`, en cuyo interior se deberá generar el código para todos los métodos que estarán disponibles desde LRP.

Muchas veces para poder conciliar la naturaleza estática de la API con el estilo dinámico de LRP, se requerirá al usuario definir una configuración inicial para ciertas variables, con el fin de poder transformarlas en variables dinámicas compatibles con Live Programming. En estos casos se deberán desplegar interfaces de comunicación que soliciten los datos requeridos cada vez que se inicie el intérprete de LRP.

9.1. Puentes implementados anteriormente

Hasta ahora se han implementado puentes para comunicar LRP con los siguientes sistemas robóticos: ROS y Ev3 Lego Mindstorms. Éstos tienen naturalezas muy distintas y en consecuencia los puentes asociados a cada uno de ellos difieren mucho entre sí, pues resuelven problemas específicos a la adaptación de cada sistema para ser utilizado en la modalidad Live Programming con LRP.

9.1.1. Puente entre LRP y PhaROS

El middleware ROS [3] presenta un enfoque de programación principalmente estático, en el que existen programas ejecutables denominados *Nodos* y canales de comunicación llamados *Tópicos*, que son utilizados por los *Nodos* para comunicarse entre sí o con el robot. Un *Nodo* puede enviar mensajes a través de un *Tópico* «publicando» en éste, y puede escuchar los mensajes publicados en un *Tópico* «subscribiéndose» al mismo.

Mientras que en ROS los *Tópicos* pueden ser creados en tiempo de ejecución, hecho que favorece su utilización en Live Programming; los *Nodos* son altamente estáticos: una vez que se definen no aceptan modificación posterior. Lo anterior implica que en ROS los programas poseen

un estado asociado, lo cual disminuye su compatibilidad con la modalidad de programación en vivo. Esto es así porque surge la necesidad de controlar la ejecución de secciones de código que modifican el estado del programa. Estas secciones son problemáticas porque si son ejecutadas reiteradamente, o si son modificadas mientras el programa se encuentra corriendo, pueden llegar a hacer que el programa caiga en un estado inválido. Por ejemplo, una sección de código que defina el comportamiento de un nodo, resultará muy problemática; ya que al ejecutarse múltiples veces, un mismo Nodo podría intentar definirse más de una vez, lo que llevará al programa a un estado inválido. Por otro lado, la modificación de esta sección de código durante la ejecución del programa, causará cambios en la definición del Nodo, provocando problemas de coherencia en el estado del programa.

PhaROS, que es un cliente para ROS en Pharo, resuelve parcialmente el problema de los Nodos estáticos, ya que posibilita su modificación en tiempo de ejecución. Sin embargo, deja sin resolver una serie de otros problemas relativos al uso de los tópicos. El primero de estos problemas es el siguiente: dado que LRP se ejecuta en un ciclo infinito, efectuar y/o modificar suscripciones en momentos puntuales, sin generar nuevas suscripciones en cada ciclo, resulta ser un importante desafío. Cuando se quiere realizar una subscripción, PhaROS crea un objeto que contiene una función callback en su interior, la cual se ejecutará cada vez que un mensaje sea recibido a través del Tópico de la subscripción. En LRP se podría ubicar el código encargado de crear las subscripciones al interior de la acción de un estado, pero como LRP ejecuta el programa permanentemente, este estado será visitado numerosas veces. Si múltiples subscripciones equivalentes son creadas, cada una de ellas llamará a su callback cuando un nuevo mensaje sea recibido, lo que sobrecargará la ejecución del programa. La situación empeora para el caso en que el programador desee modificar esta subscripción alterando su función callback, ya que la subscripción original quedará inaccesible y su callback continuará ejecutándose.

El segundo problema que queda por resolver es que la sintaxis de PhaROS para trabajar con *Nodos* y *Tópicos*, es verbosa y expone detalles de bajo nivel irrelevantes para el programador en LRP. Incluso si el programador conoce el nombre del Tópico con el que quiere trabajar, escribir el código no le resultará intuitivo. Por ejemplo, en PhaROS una subscripción se realiza del siguiente modo:

```
( aPhaROSPackageInstance controller node buildConnectionFor : aTopic )
typeAs : aType ;
for : aCallback ;
connect .
```

En este código `aPhaROSPackageInstance` representa una instancia de `PhaROSPackage`, o de una subclase. `aTopic y aType` representan respectivamente el nombre de un tópico y el tipo del mensaje que transmite. `aCallback` es la función que se llamará cada vez que un nuevo mensaje sea recibido. Finalmente el mensaje `connect` le indica a la instancia que comience la conexión con el tópico.

La sintaxis para publicar en un Tópico es similar a lo anterior, en cuanto a que no cuenta con el nivel de abstracción deseado para utilizarlo desde LRP. Idealmente se esperaría que para subscribirse o publicar en un tópico, un programador sólo necesite conocer el nombre del tópico y su tipo asociado.

Para resolver el escenario anterior se implementa un puente que comunica PhaROS con LRP, cuya principal misión es proveer un mecanismo que adapte el estilo semi-dinámico de PhaROS para funcionar en el ambiente extremadamente dinámico de LRP. El puente separa la mecánica de conexión a los *Tópicos* de la ejecución del programa en vivo, a través de una interfaz de usuario. Esta interfaz muestra los publicadores y subscriptores existentes (ver figura 16), y permite al usuario crear nuevas subscripciones o publicaciones sobre los *Tópicos* que requiera, completando información muy puntual al respecto (figura 17). Más tarde estos tópicos serán transformados en variables del objeto `robot` y podrán ser accesados dinámicamente desde el código LRP.

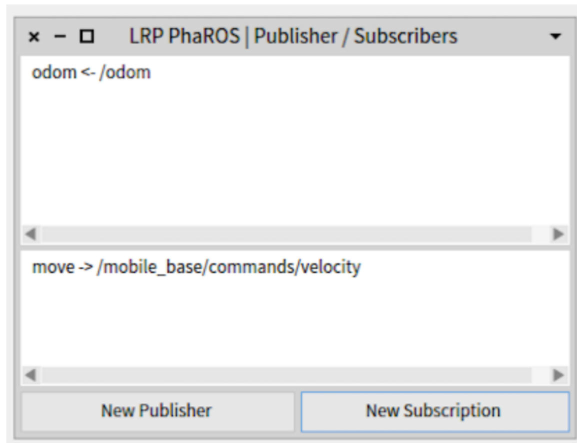


Figura 16: Interfaz de usuario que muestra Publicadores y Subscriptores

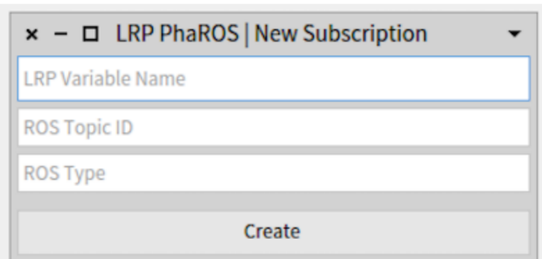


Figura 17: Interfaz de usuario para crear nueva Subscripción

El siguiente programa de ejemplo fue creado por integrantes del DCC antes de que este trabajo fuera concebido. Está escrito en LRP y utiliza PhaROS para controlar un robot robuLAB [59] (Ver figura 18). Su objetivo es hacer que el robot rote alternadamente en sentido horario y antihorario, con velocidad angular `speed`, respetando el rango definido por la variable `scope.` El comentario de la línea 02 indica que la variable p, asociada al objeto `robot`, representa la subscripción al tópico `/kompai1/pose`, el cual contiene información acerca de la posición y orientación del robot en el espacio. El comentario de la línea 03 dice que la variable motor, asociada también al objeto `robot`, representa a un publicador en el tópico `/kompai1/command_velocity`, el cual transmite información acerca de las velocidades lineales y angulares del robot en todos los ejes.

Las líneas 08 y 10 ameritan la siguiente explicación: el bloque asignado a la variable `motor`, recibe como parámetro (`msg`) un objeto que representa al mensaje enviado, cuyo tipo coincide con el tipo asociado al tópico correspondiente.

```
01;; Left right shake movement
02; p <- /kompai1/pose
03; motor -> /kompai1/command_velocity
04(var scope := [0.2])
05(var speed := [0.2])
06(machine leftright
07   (state turnleft
08      (running [robot motor: [ :msg| msg angular z: speed ]]))
09   (state turnright
10      (running [robot motor: [ :msg| msg angular z: speed negated ]]))
11   (on tooright turnright -> turnleft totheleft)
12   (on tooleft turnleft -> turnright totheright)
13   (event tooleft [robot p pose orientation z > scope])
14   (event tooright [robot p pose orientation z < scope negated])
15)
16(spawn leftright turnright)
```

Figura 18: Imagen referencial del robot robuLAB

9.1.2. Puente entre LRP y JetStorm

JetStorm es un cliente para controlar los robot Ev3 Lego Mindstorms [6] (ver figura 19) a través de una conexión TCP/IP, mediante una red WiFi. Consiste en una API que provee métodos para controlar los motores y leer los sensores del robot. El Ev3 cuenta con 4 puertos para motores y 4 para sensores.

A diferencia de PhaROS, esta API casi no posee estados intrínsecos asociados, hecho facilita su integración con Live Programming. Así, las responsabilidades del puente se limitan casi exclusi-

vamente a exponer los métodos de la API hacia LRP, lo que se logra a través de la pseudovariable `robot`.

El único estado asociado a JetStorm, corresponde al estado de la conexión WiFi con el Ev3. Si el mecanismo de conexión se incluyese al interior de un programa en LRP, la conexión se inicializaría indefinidas veces, lo cual obviamente causaría graves errores. El puente resuelve esta situación proveyendo una interfaz de comunicación con el cliente, que solicita la dirección IP del Ev3 y se encarga de establecer la conexión inalámbrica antes de comenzar a programar en LRP.

Se adjunta un ejemplo escrito en LRP, creado previamente a este trabajo de memoria. El programa controla un robot Ev3 para que sigua una una línea negra dibujada en el piso, cruzándola de un lado a otro alternadamente. En el código, puede verse en las líneas 02, 03 y 04 cómo acceder a los sensores y motores del Ev3, a través de `robot`.

En el ejemplo, los motores que controlan las ruedas del robot se activan independientemente a través de las variables `mright` y `mleft`. Activar la rueda derecha hará que el robot se mueva hacia la izquierda, y viceversa. La máquina de estados correspondiente a este ejemplo contiene 4 estados y 4 transiciones. los estados `left` y `right` representan los momentos en que el robot se encuentra a la izquierda o a la derecha de la linea respectivamente; `crossfr` y `crossfl`, a aquellos en que el robot cruza la línea desde la derecha o desde la izquierda. Las 4 transiciones de la máquina, comunican los estados anteriores de modo que el robot pase de estar a la izquierda de la línea, a cruzarla desde la izquierda, para luego estar a la derecha de la misma, para luego cruzarla desde la izquierda, repitiendo el proceso indefinidamente.

```
01;;; Line following, goes forward by always crossing the line
02(var sensor := [robot sensor3])
03(var mright := [robot motorA])
04(var mleft := [robot motorB])
05(var speed := [15])
06(machine linecross
07   (state left
08      (onentry [mright startAtSpeed: speed]))
09   (state crossfl
10      (onexit [mright stop]))
11   (state right
12      (onentry [mleft startAtSpeed: speed negated]))
13   (state crossfr
14      (onexit [mleft stop]))
15   (on black right -> crossfr rlx)
16   (on black left -> crossfl lrx)
17   (on white crossfr -> left rl)
18   (on white crossfl -> right lr)
19   (event black [sensor read = 1])
20   (event white [(sensor read = 1) not])
21)
22(spawn linecross left)
```

Figura 19: Imagen referencial del robot Ev3 Lego Mindstorms

9.2. Diseño e implementación del puente entre LRP y la API en Pharo

La API de comunicación con el drone implementada en Pharo, es similar a JetStorm en cuanto a que su estado interno más relevante durante una sesión de comunicación con el robot, es el estado de la conexión entre cliente y robot. Por ello, el puente para la API del AR.Drone comparte más similitudes con el puente para JetStorm que con el de PhaROS. No obstante lo anterior, el puente para AR.Drone contiene funcionalidades adicionales a las de JetStorm, que se hicieron necesarias a medida que se utilizó LRP para programar comportamientos en el drone.

En términos generales, el puente para AR.Drone cumple las siguientes funciones:

1. Adapta los métodos de la API como métodos del objeto `robot`.
2. Brinda una interfaz de usuario que se despliega al iniciar LRP, cuyas funciones son:
    a) Proveer un botón para intentar conectarse con el robot.
    b) Ofrecer controles básicos para mover el drone
    c) Suministrar controles para modificar aspectos clave de la configuración del drone rápidamente.
    d) Reportar el estado actual de la conexión y el nivel de batería del drone.
3. Posibilita la visualización del stream de video transmitido por el drone, utilizando una aplicación externa.

9.2.1. Especificaciones sobre la implementación del puente

A nivel de implementación, el puente para la API del AR.Drone consiste en una subclase de `LRPAbstractBridge`, que contiene una instancia de ARDrone como variable de clase. LRP

demanda que tal instancia sea única, es decir, que respete el patrón singleton. No hubo necesidad de implementar este requerimiento al interior del puente, pues la clase ARDrone ya implementa este patrón.

El método `openUI`, que es definido al interior de esta clase, se encarga de 3 labores principales.

- Su primera tarea es inicializar la instancia de ARDrone, proceso que puede realizarse automáticamente, sin la intervención de ningún usuario externo. Esto porque a diferencia del Ev3, la IP del AR.Drone se considera fija (la explicación de por qué la IP del drone se asume fija, se encuentra en la sección 4.1).
- La segunda labor de esta función es generar los métodos que estarán disponibles en LRP desde la pseudovariable `robot`. Se detalla en la sección 9.2.2 cuáles métodos públicos de la API son incluidos en el puente, a través de la creación de métodos con idéntica firma.
- La tercera y última misión de openUI es generar la interfaz de usuario que despliega información acerca del estado del drone y que además permite controlar los movimientos del drone utilizando el teclado. Esta interfaz se explica en detalle en la sección 9.2.3. Una responsabilidad especial de esta interfaz es permitir visualizar el video proveniente desde el drone. La sección 9.2.4 se dedica a explicar cómo fue implementada esta funcionalidad.

Finalmente, esta clase reimplementa el método `closeUI`, que está definido en `LRPAbstractBridge`, y que es llamado cuando la interfaz de usuario de LRP se cierra. `closeUI` simplemente envía el mensaje `terminate` al objeto `arDroneInstance`.

9.2.2. Métodos de la API en Pharo accesibles desde LRP

Los métodos públicos de la API del AR.Drone son numerosos, y se encuentran clasificados principalmente en 4 protocolos según su función: «move», «config», «state» y «tag detection». No todos ellos fueron incluídos como métodos asociados a la pseudovariable robot.

- «Move» reúne los métodos dedicados a mover el drone, ejemplos de ellos son takeOff, pitch:, roll:, etc. Todos estos métodos están presentes en el puente.
- «Config» contiene métodos que alteran u obtienen valores de configuración interna del drone. Cuenta con dos métodos genéricos que permiten obtener y establecer cualquier parámetro de configuración, ya que reciben como argumento el nombre del parámetro de configuración como String. Pese a que pudiera resultar redundante se tomó la decisión de incorporar además métodos que alteraran valores de configuración específicos para el caso de ciertos parámetros que son usados con mucha frecuencia, y para algunos otros, cuya correcta configuración depende de la configuración de parámetros adicionales (por ejemplo el parámetro «control:flying_mode»).
- «State» agrupa los métodos que informan sobre el estado interno del drone, algunos ejemplos son: lowBattery, altitude, isFlying, roll, etc. Todos ellos se incluyeron en el bridge.
- «Tag detection» engloba los métodos asociados a la detección de tags. Todos ellos se incorporaron al puente.

9.2.3. Diseño de la interfaz de usuario

Al iniciar LRP, la interfaz de usuario principal asociada al puente es desplegada. Su estructura se divide en cinco secciones (ver figura 20):

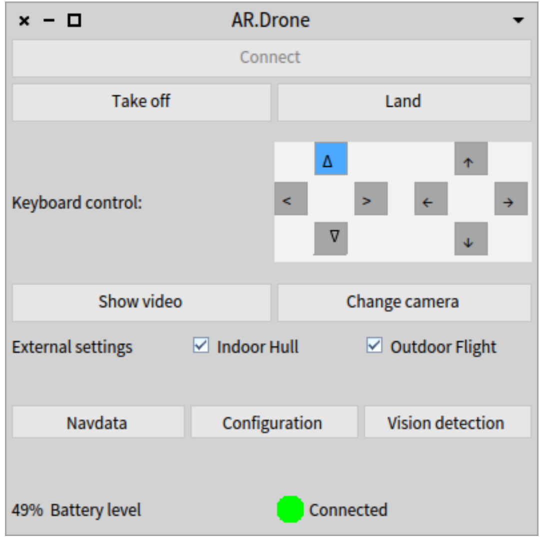


Figura 20: Interfaz de usuario del puente entre LRP y la API del AR.Drone

1. La sección superior sólo contiene el botón «Connect». Al ser presionado se intenta establecer conexión con el drone. Durante el tiempo que se intenta conectar y mientras el cliente se encuentre conectado, este botón estará deshabilitado.

2. La segunda sección contiene herramientas para mover al drone. Consta de los botones Take off y Land, y de un objeto Morph. Este último, ubicado junto al label «Keyboard control», es en realidad una instancia de la interfaz de usuario ARDUIKeyboard, descrita en la sección 7.7.2, y sirve para controlar los movimientos del drone usando el teclado. Además refleja el estado actual del objeto Motion State, tanto si éste ha sido modificado mediante eventos del teclado o bien de manera programática. Tener la capacidad de mover el drone manualmente, evita que el programador deba levantarse a reposicionar el drone después de cada prueba. El botón Land resulta especialmente útil para casos en que se desea aterrizar el drone de inmediato durante un vuelo controlado por LRP. Estos tres controles pueden usarse en cualquier momento siempre que la conexión se encuentre activa, y funcionan independientemente del estado de la máquina activa en LRP.

3. La tercera sección agrupa controles de configuración del drone. La primera fila incluye los botones «Show video» y «Change camera», que, tal como sus nombres indican, sirven respectivamente para mostrar el video del drone y para cambiar la cámara que lo transmite. Esta funcionalidad es muy útil mientras se escriben programas que utilizan la detección de tags del drone, ya que el programador puede ver lo que el drone ve. La siguiente fila contiene dos checkbox: «Indoor hull» y «Outdoor flight», ambos configuran características físicas externas. «Indoor hull» le indica al firmware del drone qué tipo de carcasa está equipada. Según sea el caso, el firmware adaptará sus algoritmos de estabilización para optimizar el vuelo. «Outdoor flight» le indica al firmware si el drone se encuentra volando en exteriores, de ser así, los valores máximos para las inclinaciones y velocidades del

drone aumentarán. Estos checkboxes de configuración permanecen deshabilitados hasta que el archivo de configuración interna del drone haya sido recibido en el cliente cuando se (re)establece la conexión. De este modo la interfaz puede obtener los valores iniciales para estos controles a partir de la configuración enviada por el drone, y así reflejar su estado inicial con transparencia.

4. La cuarta sección consiste en tan sólo una fila que agrupa tres botones. Cada botón despliega una nueva ventana que expone determinada información utilizando controles de tipo listview cuyo contenido se actualiza periódicamente. La frecuencia de actualización de los datos de cada ventana fue determinada empíricamente, buscando un valor que minimizara la taza de refresco, pero que permitiera al usuario consultar los datos prácticamente en tiempo real. Para datos cuyos valores cambian con rapidez (como por ejemplo navdata), se determinó que las actulizaciones fueran realizadas cada 250 milisegundos. Para datos menos propensos a cambiar (como los datos de configuración) se dispuso un tiempo de actualización de 800 milisegundos.

    a) Navdata: La ventana que despliega los datos de navegación se divide en dos secciones: la sección izquierda revela todo el contenido de los paquetes de navdata, esto es: encabezado, opción Demo y opción Vision Detect. Al interior del encabezado se muestra el parámetro «status», correspondiente al estado del drone, como un valor hexadecimal que no puede ser interpretado directamente por el usuario. Es por ello que en la sección derecha de esta ventana se ha expandido la información del estado del drone. (ver figura 21). La ventana y su contenido son actualizados cada 250 milisegundos

    b) Configuration: Consiste en una única lista que contiene todos los valores de configuración interna del drone informados por éste. La ventana y su contenido son refrescados cada 800 milisegundos. (ver figura 22)

    c) Vision detection: Busca ser una ayuda gráfica explícita y sencilla que indique al programador, usando colores, el estado actual de los tags detectados por el drone. Pese a que la información relativa a la detección de tags se incluye a cabalidad al interior de paquetes de Navdata, su lectura no es intuitiva y resulta poco práctica especialmente durante en los momentos de prueba. En contraste, la ventana Vision detections presenta información muy reducida, pero rápida de leer y de entender. La ventana se divide dos secciones. La sección izquierda está dedicada a la detección del tag Oriented Roundel a través de la cámara vertical. La sección derecha muestra información relativa al tag Shell detectado en la cámara horizontal. Cada una de las filas representa una condición asociada al tag, que cuando se cumple torna su fila verde, mientras que cuando no se cumple torna la fila blanca. (ver figura 23). Para considerar cumplidas a aquellas condiciones relacionadas a posición y orientación, se determinaron valores de tolerancia. Estos valores fueron ajustados mediante pruebas realizadas con cada tag, y se eligieron buscando preservar el sentido de cada condición, pero relajando la exigencia sólo lo suficiente para que su cumplimiento no fuera excesivamente difícil de lograr mientras el drone se encontrara volando. Las siguientes son las 4 condiciones consideradas en la interfaz:

        1) Detected: marca si el tag está siendo detectado a cada momento

2) X Centered: marca si el tag está centrado en la imagen respecto al eje horizontal X (con tolerancia de ±5 %)

3) Y Centered: marca si el tag está centrado en la imagen respecto al eje vertical Y (con tolerancia de ±10 % para Shell y de ±5 % para Oriented Roundel)

4) Oriented (sólo para el tag Oriented Roundel): marca si el tag se encuentra orientado en un ángulo de 0 grados (con tolerancia de ±5 %)

Hay que recordar que por razones mencionadas en la sección 7.4, la API no soporta la detección de varios tags de un mismo tipo, y recomienda además no habilitar la detección del mismo tipo de tag en ambas cámaras a la vez.

La ventana y su contenido son actualizados cada 250 milisegundos.

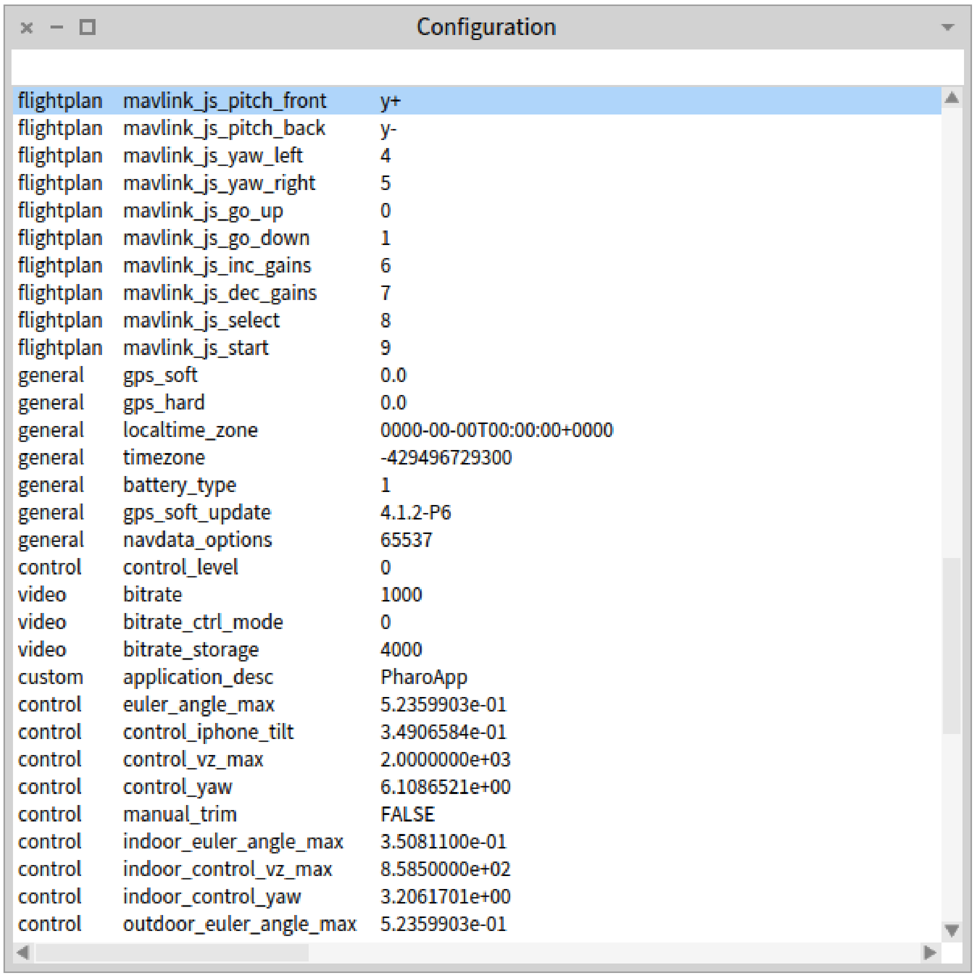
5. La última sección de la interfaz muestra dos indicadores del estado del drone que son de especial relevancia para el programador. A la izquierda se muestra el porcentaje de batería disponible, e incluye un mensaje de alerta en caso que éste haya alcanzado un nivel crítico. Este dato es importante porque cuando el firmware del drone detecta que la batería está muy baja, deja de responder ante las peticiones de despegue. A la derecha se muestra el estado de conexión actual con el drone. Se incluye un ícono que cambia de color según cambia el estado, y se acompaña con un label explicativo. El color rojo indica desconexión; amarillo, que se está intentando conectar; verde significa que la conexión está activa.

Navdata

header	16r55667788
state	16r8F8A04D0
seq	148013
vision	1
tag	0
size	148
control state	0
batery	48
pitch angle	6882.0
roll angle	-1312.0
yaw angle	126683.0
altitude	0
estimated velocity x	17.057762145996094
estimated velocity y	2.7608234882354736
estimated velocity z	0.0
tag	16
size	328
nb Detected	1
type	#(0 0 0 0)
xc	#(890 0 0 0)
yc	#(552 0 0 0)
width	#(153 0 0 0)
height	#(69 0 0 0)
distance	#(72 0 0 0)
orientation angle	#(0.0 0.0 0.0 0.0)
camera source	#(0 0 0 0)

FLY MASK	0
VIDEO MASK	0
VISION MASK	0
CONTROL ALGO	0
ALTITUDE CONTROL ALGO	1
USER feedback	0
Control command ACK	1
Camera enable	1
Travelling enable	0
USB key	0
Navdata demo	1
Navdata bootstrap	0
Motors status	0
Communication Lost	0
Problem with gyrometers	0
VBat low	0
User Emergency Landing	0
Timer elapsed	1
Magnetometer calibration state	0
Angles	1
WIND MASK	0
Ultrasonic sensor	0
Cutout system detection	0
PIC Version number OK	1
ATCodec thread ON	1
Navdata thread ON	1
Video thread ON	1
Aquisition thread ON	1
CTRL Watchdog	0
ADC Watchdog	0
Communication Watchdog	0
Emergency landing	1

Figura 21: Interfaces para el puente con LRP: Navdata



flightplan	mavlink_js_pitch_front	y+
flightplan	mavlink_js_pitch_back	y-
flightplan	mavlink_js_yaw_left	4
flightplan	mavlink_js_yaw_right	5
flightplan	mavlink_js_go_up	0
flightplan	mavlink_js_go_down	1
flightplan	mavlink_js_inc_gains	6
flightplan	mavlink_js_dec_gains	7
flightplan	mavlink_js_select	8
flightplan	mavlink_js_start	9
general	gps_soft	0.0
general	gps_hard	0.0
general	localtime_zone	0000-00-00T00:00:00+0000
general	timezone	-429496729300
general	battery_type	1
general	gps_soft_update	4.1.2-P6
general	navdata_options	65537
control	control_level	0
video	bitrate	1000
video	bitrate_ctrl_mode	0
video	bitrate_storage	4000
custom	application_desc	PharoApp
control	euler_angle_max	5.2359903e-01
control	control_iphone_tilt	3.4906584e-01
control	control_vz_max	2.0000000e+03
control	control_yaw	6.1086521e+00
control	manual_trim	FALSE
control	indoor_euler_angle_max	3.5081100e-01
control	indoor_control_vz_max	8.5850000e+02
control	indoor_control_yaw	3.2061701e+00
control	outdoor_euler_angle_max	5.2359903e-01

Figura 22: Interfaces para el puente con LRP: Configuration

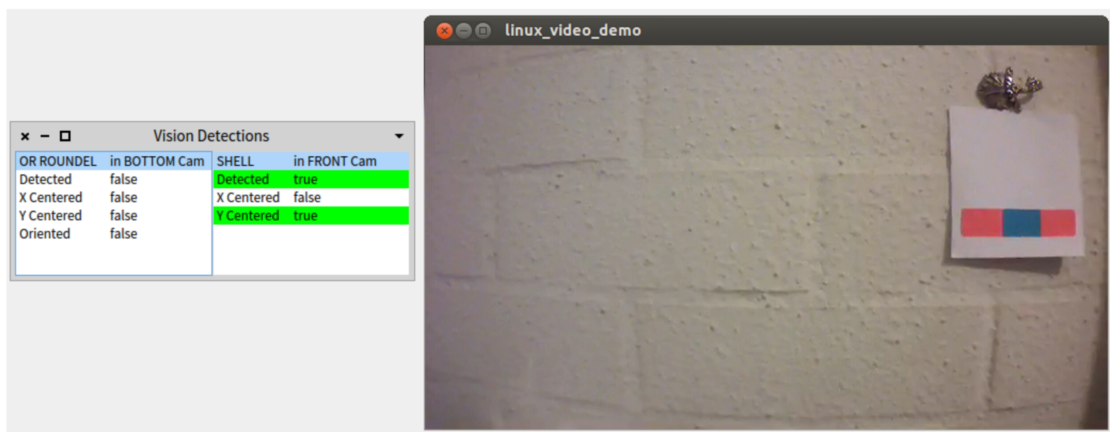


Figura 23: Interfaces para el puente con LRP: Vision Detections

9.2.4. Visualización de video

El AR.Drone 2.0 es capaz de transmitir un stream de video a través del protocolo TCP, puerto 5555. Cada uno de sus frames está dividido en *slices*, que son transmitidos en paquetes separados. El formato de encapsulamiento de los slices es propio de Parrot y se denomina PaVe (ver sección 4.3.3). Además, cada frame está codificado de acuerdo a uno de los siguientes dos

formatos de compresión de video: H264 (MPEG4.10) o MPEG4.2. Para efectos de este trabajo, implementar las herramientas necesarias para recibir, desencapsular y decodificar el video es innecesario, puesto que Parrot brinda un ejemplo al interior de su SDK, que utiliza la API oficial para recibir y desplegar el video en vivo, en el cliente.

El puente hacia LRP permite mostrar el video y cambiar la cámara que lo transmite. Para desplegar el video, se utiliza la clase de Pharo OSProcess, que permite ejecutar scripts de línea de comando y así administrar procesos al exterior de Pharo. Al interior del puente, específicamente en la clase LRPARDroneControlUI, se define una variable de instancia, llamada videoProcess-Path, que debe contener la ruta del archivo ejecutable 'linux_video_demo'. El puente incluye un valor por defecto para esta variable. El usuario del puente deberá corregir este valor para que concuerde con la ubicación que el archivo linux_video_demo tiene en su sistema.

El puente incorpora adicionalmente un único botón para cambiar la cámara que transmite. Cada vez que el botón «Change camera» es presionado, la cámara activa se intercambia con la inactiva. Esta funcionalidad no depende de ningún proceso externo, en cambio utiliza una de las funciones públicas de la API implementada en Pharo.

9.3. Programas de prueba

Se crearon programas de prueba sencillos en LRP con el objetivo de comprobar que el puente funcionara correctamente y además para completar la definición del alcance de su interfaz de usuario.

9.3.1. Square path

En este programa el drone despega, luego vuela siguiendo una trayectoria cuadrada y aterriza cuando llega al punto de partida. Consta de una máquina principal, denominada `drone`, que contiene 3 estados: `takingoff`, `flying` y `landing`. El estado `flying` contiene una máquina de estados anidada `flightpath`. Ésta implementa la lógica del movimiento cuadrado. Contiene 8 estados: 4 de ellos mueven al drone, mientras que los otros 4 estabilizan su vuelo, es decir, lo detienen sobre un punto en el aire. Las 8 transiciones en la máquina flypath y las dos en drone, son transiciones de tipo timeout, es decir, dependen del cumplimiento de un plazo temporal, y no del estado del drone.

```
01(var velocity := [0.3]) (var takingofftime := [5000])
02(var flighttime := [8000]) (var stime := [flighttime / 10])
03(machine drone
04   (state takingoff
05      (onentry [robot takeOff]))
06   (state landing
07      (onentry [robot land]))
08   (state flying
09      (machine flightpath
10         (state flyforward
11            (onentry [robot pitch:1.7*velocity negated]))
12         (state stopforward
13            (onentry [robot stabilize]))
14         (state flybackwards
15            (onentry [robot pitch:velocity]))
16         (state stopbackwards
17            (onentry [robot stabilize]))
18         (state flyright
19            (onentry [robot roll:velocity]))
20         (state stopright
21            (onentry [robot stabilize]))
22         (state flyleft
23            (onentry [robot roll:velocity negated]))
24         (state stopleft
25            (onentry [robot stabilize]))
26         (ontime stime flyforward -> stopforward)
27         (ontime stime stopforward -> flyright)
28         (ontime stime flyright -> stopright)
28         (ontime stime stopright -> flybackwards)
30         (ontime stime flybackwards -> stopbackwards)
31         (ontime stime stopbackwards -> flyleft)
32         (ontime stime flyleft -> stopleft)
33         (ontime stime stopleft -> flyforward))
34      (onentry (spawn flightpath flyforward))
35   (ontime takingofftime takingoff -> flying)
36   (ontime flighttime flying -> landing))
37(spawn drone takingoff
```

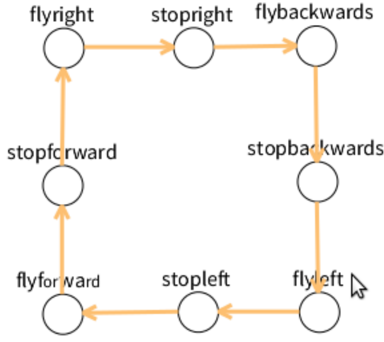


Figura 24: Diagrama de estados de la máquina anidada: flightpath

9.3.2. Tag detector

El ejemplo tag detector comprueba que la información contenida en los datos de navegación enviados por el drone se vea reflejada correctamente, y dentro de un tiempo prudente, en el estado de la máquina en LRP. Tag detector consta de sólo dos estados: `detected` y `undetected`. Cuando el drone detecta algún tag, entra en el estado `detected` y sus leds se vuelven verdes. Cuando detecta cero tags, pasa al estado `undetected` y sus leds se tornan rojos. Las variables `green` y `red`, definidas al inicio del código, representan el número de animación requerido por la función `robot animateLEDs: frequency: duration:` . Los valores para las animaciones, junto con todas las otras constantes relacionadas a ARDrone, pueden ser accesados mediante la clase `ARDConstant`.

```
01(var green := [ARDConstant ledAnimation_GREEN])
02(var red := [ARDConstant ledAnimation_RED])
03(var t := [100]) (var f := [1]) (var ntags := [[robot tagsTotalDetected]])
04(machine tagdetector
05   (state undetected
06      (onentry [robot animateLEDs: red frequency: f duration: t]))
07   (state detected
08      (onentry [robot animateLEDs: green frequency: f duration: t]))
09   (on [ntags value > 0] undetected -> detected found)
10   (on [ntags value = 0] detected -> undetected lost))
11(spawn tagdetector undetected
```

Figura 25: Diagrama de estados de la máquina: tagdetector

9.4. Disponibilidad

El código del puente se encuentra alojado en un repositorio Smalltalkhub, bajo el nombre de proyecto «LiveRobotics-ARDrone». Su URL es:

`http://smalltalkhub.com/#!/~CaroHernandez/LiveRobotics-ARDrone/`

10. Validación y Lecciones Adquiridas

10.1. Validación

Ya que la metodología aplicada durante el presente trabajo de memoria fue de tipo espiral, el proceso de validación estuvo presente desde las primeras etapas del desarrollo, realizándose iterativamente a medida que las herramientas incrementaban su complejidad. Sin embargo, el evento de validación más importante, por ser el más completo de todos los realizados, ocurrió en el VII Encuentro de Robótica del Colegio Saint George. Los siguientes párrafos presentan una narrativa de la experiencia.

El día sábado 4 de Junio de 2016, el Colegio Saint George celebró su VII Encuentro de Robótica al interior de sus instalaciones. El objetivo de este evento es acercar al área a alumnos de educación media y básica, mediante charlas, exposiciones, talleres y competencias. En esta oportunidad participaron varios colegios de distintas partes de Santiago, asistienedo tanto a los alumnos como a sus familias. Además, fueron convidadas a exponer instituciones privadas, como Knight Robotics [60], y equipos de la Universidad de Chile compuestos por alumnos del Departamento de Ingeniería Eléctrica y de Computación.

Como equipo invitado, representamos al DCC en compañía de los estudiantes de doctorado Pablo Estefó y Miguel Campusano. El equipo se ubicó en un estand, donde se ofreció información a los visitantes acerca el trabajo que realiza el DCC en el área de robótica. Se utilizó como material de apoyo un video que mostraba comportamientos programados para los robots PR2 [5], Lego Mindstorms [6] y Turtlebot [4]. Se entregaron folletos con información acerca de la carrera de ingeniería y revistas del Departamento a aquellos visitantes más interesados.

Adicionalmente se realizaron diversas demostraciones utilizando dos tipos de robots: el Turtlebot y el AR.Drone 2.0. Con este último se realizaron 4 tipos de demostraciones, dos de las cuales eran interactivas para el público:

1. Shell tag follower: A través de un programa escrito en LRP, el drone identifica el «Shell Tag» mediante su cámara frontal y lo sigue, manteniendo una distancia constante. Se trata de una demo interactiva, donde los visitantes pudieron mover el tag y usarlo para guiar al drone a su antojo. Ver figura 26.

2. Oriented roundel visitor: Usando un programa escrito en LRP, el drone se mueve de forma autónoma, y realiza un recorrido definido por dos tags de tipo «Oriented Roundel», enfrentados a 1.7 metros de distancia. El drone despega, utiliza su cámara vertical para reconocer el tag más cercano, se posiciona encima suyo y se orienta con respecto a éste, luego se mueve hacia el frente hasta identificar el segundo tag con su cámara vertical. El proceso se repite indefinidamente, de manera que el drone visita alternadamente ambos tags. Es importante notar que aunque en esta demo se utilizaron sólo dos tags, el mismo código permite utilizar tantos tags como se quiera, siempre y cuando éstos se dispongan formando un polígono regular y estén orientados apropiadamente, definiendo un ciclo cerrado. Esta demo no es interactiva, los visitantes se limitaron a observar el comportamiento del drone. Ver figura 27.

3. Oriented roundel follower: Se usa la interfaz gráfica de usuario de LRP para establecer el modo de vuelo del drone como «Hover on top of oriented roundel». Se posiciona un tag «oriented roundel» en el piso, y se le añade una cuerda para que un voluntario pueda arrastrarlo. Cuando el drone identifica el tag con su cámara inferior, se orienta respecto a éste y permanece encima suyo. Cuando el tag se mueve, el drone lo sigue. El modo «Hover on top of oriented roundel» no fue programado en LRP, sino que corresponde a un modo de vuelo preestablecido en el firmware del drone (interfaz de usuario de LRP en figura 20).

4. Exposición de vuelo y piruetas: A petición de los visitantes, se realizaron vuelos de demostración controlados por teclado, que incluyeron la realización de piruetas preestablecidas por el firmware del drone. Las piruetas que más impactaron al público fueron los flips de 360°, tanto frontales como laterales (ver piruetas en Tabla 5).

Una vasta cantidad de visitantes acudió al estand durante la jornada. Naturalmente ocurrió que los niños presentaron mayor interés en las demos, especialmente en las interactivas; mientras que los adultos, aunque también participaron de las demostraciones, se interesaban más por entender concretamente cómo y en qué se trabaja en el Departamento y preguntaban además acerca del alcance de las aplicaciones creadas. Pese a que la gran mayoría de los visitantes no se dedicaba a la computación, sus preguntas muchas veces fueron acuciosas. Incluso se dio la oportunidad de mencionar a un par de visitantes que los comportamientos del drone habían sido programados utilizando máquinas de estado.

En más de un par de ocasiones tuvimos que interrumpir las demostraciones porque todas las baterías del drone se habían agotado. Esto ocurrió pese a que contábamos con tres baterías cargadas y dos cargadores disponibles. En condiciones óptimas, una batería permite al drone mantenerse en vuelo durante 10 minutos y demora alrededor de una hora y media en cargarse.

Figura 26: Demostración «Tag follower» versión simple, realizada en Colegio Saint George

Figura 27: Demostración «Oriented Roundel visitor» realizada en Colegio Saint George

10.1.1. Programa Shell tag follower

El ejemplo «shell tag follower» logra que el drone siga a un tag de tipo «shell» detectado por la cámara frontal. En términos muy generales este ejemplo consiste en lo siguiente: el drone puede encontrarse en dos estados: «waiting» y «following». Se inicia en estado «waiting» y se permanece allí mientras el tag no sea detectado. Se pasa a estado «following» apenas se detecte el tag. Al interior de following pueden ocurrir dos cosas: el tag está centrado y a la distancia adecuada, en cuyo caso el drone no se mueve y permanece en estado «hover», o bien el tag no está centrado y/o se encuentra a una distancia inadecuada, caso en el cual el drone se mueve buscando centrar al tag y posicionarse a la distancia requerida.

Este ejemplo cuenta con dos implementaciones distintas. La primera busca reflejar la sencillez con la cual es posible abordar la solución, la segunda busca lograr movimientos más controlados a costa de una mayor complejidad.

La solución más sencilla será expuesta a continuación, y más adelante se mostrará aquella más compleja.

Solución Simple

Destaca por su breve extensión y gran sencillez. Los diagramas de estado generados por LRP para este código, se muestran en la figura 28. En la url `http://bit.ly/ARDroneLRP` se aloja un video con una demostración de la solución.

```
01 (var speed := [0.2])
02 (var zDistance := [120])
03 (var xTolerance := [0.35]) (var yTolerance := [0.35]) (var zTolerance := [0.4])
04 (var tag := [robot tagDetected: ARDConstant detectedTag_SH_IN_CH])
05 (var tagcentered := [[ (tag isCenteredInXWithTolerance: xTolerance) &
                          (tag isCenteredInYWithTolerance: yTolerance) &
                          (tag isAtDistance: zDistance tolerance: zTolerance)]]
```

Antes de explicar cada variable y su rol en el programa, es relevante aclarar que la posición del tag con respecto a la cámara ha sido considerada como un vector con origen en la cámara y que apunta hacia el tag. En este sistema las coordenadas X e Y corresponden a los ejes horizontal y vertical del tag en la imagen de la cámara frontal del drone, con origen en el centro de la imagen; por ende, el eje Z será el vector que represente la distancia aproximada a la que se encuentra el tag con respecto a la cámara. En este ejemplo el drone se moverá buscando posicionar al tag en el origen de los ejes X e Y y a una distancia determinada en Z. Cada dirección de movimiento se asocia a un eje y a un tipo de movimiento del drone. El eje X queda asociado al movimiento roll, el eje Y al movimiento elevate y Z a pitch.

La variable tag representa al tag detectado por el drone a través de una cámara específica. `speed` determina la rapidez máxima a la que se moverá en drone en todos los sentidos. Las variables de sufijo «Tolerance» representan un porcentaje de holgura aceptado para determinar si una coordenada se encuentra posicionada correctamente. zDistance es la distancia en centímetros

a la que se mantendrá el drone con respecto al tag (según el eje Z). Por último `tagcentered` es un bloque que al ser evaluado indica si el el tag se encuentra correctamente posicionado o no.

```
06 (machine tagFollower
07   (state waiting)
08   (state following
```

Entre las líneas 06 y 08 se define una máquina llamada tagFollower con dos estados: `waiting`, que representa el tiempo en que ningún tag esté siento detectado; y `following` que representa lo opuesto.

```
09      (machine followAlgo
10         (state positioned)
11         (state moving
12            (running [robot moveByLeftRightTilt:(speed * (tag offsetInX))
                               frontBackTilt:(-1 * speed * (tag offsetInDistance: zDistance))
                               angularSpeed: 0
                               verticalSpeed:(-1* speed * (tag offsetInY))])
13            (onexit [robot stabilize]))
14         (on [tagcentered value] positioned -> moving tout)
15         (on [tagcentered value not] moving -> positioned tin))
16      (onentry (spawn followAlgo positioned)))
```

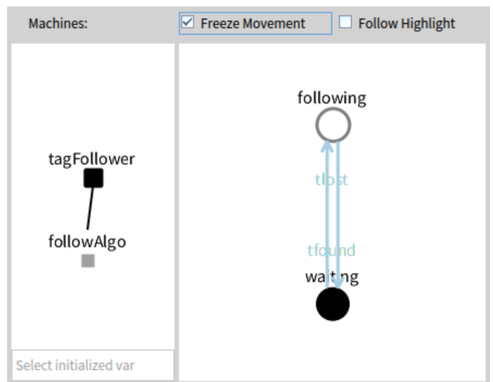
Al interior del estado following se define la máquina followAlgo, que se compone de dos estados: positioned, el cual se activará cuando el tag detectado se considere posicionado lo suficientemente cerca de la «zona central» deseada para el tag; y moving, que estará activo en caso contrario. Notar que la zona deseada para el tag, queda definida como (x,y,z) = (0, 0, zDistance).

Durante todo el tiempo que moving sea el estado activo, el drone se estará moviendo de acuerdo a la instrucción especificada en la acción running. Hay que notar que la velocidad a la que se mueva el drone en cada coordenada espacial, aumentará según lo alejado que se encuentre el tag del punto central definido según la coordenada correspondiente, la dependencia entre ambas variables es lineal.

```
17   (on [tag detected] waiting -> following tfound)
18   (on [tag detected not] following -> waiting tlost))
19 (spawn tagFollower waiting)
```

Las últimas líneas definen las transiciones asociadas a la máquina principal. Las transiciones se gatillarán dependiendo de si la detección del tag tipo shell ocurre o deja de ocurrir.

Esta solución funciona correctamente cuando la rapidez del drone se limita, si este límite se incrementa la solución deja de comportarse tan bien, ya que no hay ningún mecanismo que contrarreste lo suficiente la inercia del movimiento del drone y los intentos de estabilizar su vuelo dejan de ser efectivos.



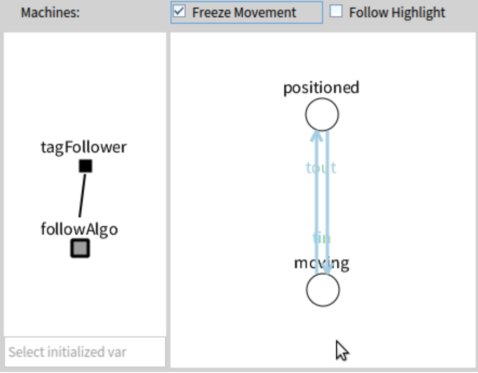


Figura 28: Diagrama de estados de las máquinas tagFollower y followAlgo, versión simple

Solución Compleja

Esta solución busca ejercer mayor control sobre los movimientos del drone, «frenándolo» cuando sea necesario. La máquina principal definida es idéntica a la de la solución anterior, sólo existen diferencias en la máquina anidada. Las diferencias se extienden específicamente entre las líneas 13 y 37 del código. Los diagramas generados por LRP para ambas máquinas se muestran en la figura 29.

Enseguida se expone el código y sólo se explican sus diferencias con respecto a la versión simple.

```
01 (var speed := [0.3]) (var zDistance := [200])
02 (var xTolerance := [0.35]) (var yTolerance := [0.35]) (var zTolerance := [0.4])
03 (var tag := [robot tagDetected: ARDConstant detectedTag_SH_IN_CH])
04 (var stopTime := [60])
05 (var rollSpeed := [0]) (var pitchSpeed := [0]) (var elevateSpeed := [0])
06 (var tagIsXCentered := [[ tag isCenteredInXWithTolerance: xTolerance ]])
07 (var tagIsYCentered := [[ tag isCenteredInYWithTolerance: yTolerance ]])
08 (var tagIsZCentered := [[ tag isAtDistance: zDistance tolerance: zTolerance ]])
09 (machine tagFollower
10    (state waiting)
11    (state following
12       (machine followAlgo
13          (state centered)
14          (state xMoving
15             (onentry [ rollSpeed := (speed * (tag offsetInX)).
16                         robot roll: rollSpeed.])
17             (onexit [ robot roll:(-1.5 * rollSpeed).
18                        (Delay forMilliseconds: stopTime) wait.
19                        robot stabilize ]))
20          (state yMoving
21             (onentry [ elevateSpeed := (-1 * speed * (tag offsetInY))
22                        robot elevate: elevateSpeed ])
23             (onexit [robot elevate:(-1.5 * elevateSpeed).
24                        (Delay forMilliseconds: stopTime) wait.
25                        robot stabilize]))
26          (state zMoving
27             (onentry [ pitchSpeed := (-1 * speed * (tag offsetInDistance: zDistance)).

28                        robot pitch: pitchSpeed])
29             (onexit [robot pitch: (-2 * pitchSpeed).
30                        (Delay forMilliseconds: stopTime) wait.
31                        robot stabilize]))
32          (on [tagIsXCentered value not] centered -> xMoving)
33          (on [tagIsYCentered value not] centered -> yMoving)
34          (on [tagIsZCentered value not] centered -> zMoving)
35          (on [tagIsXCentered value] xMoving -> centered)
36          (on [tagIsYCentered value] yMoving -> centered)
37          (on [tagIsZCentered value] zMoving -> centered)
38       )
39       (onentry (spawn followAlgo centered))
40    )
41    (on [tag detected] waiting -> following)
42    (on [tag detected not] following -> waiting)
43 )
44 (spawn tagFollower waiting)
```

El diseño de la máquina lookAlgo, contempla 4 estados: 3 de movimiento y 1 de reposo, cada estado de movimiento se corresponde con un eje. Cuando el drone se mueve en un eje lo hace con una velocidad que queda determinada por cuán alejado se encuentra el tag de la posición deseada para éste en tal eje. Cuando el tag alcanza la posición deseada, el drone «frena» el movimiento

efectuado ordenando al drone moverse en el mismo eje y con una rapidez proporcional, pero en sentido contrario durante un corto intervalo de tiempo.

Cada estado de movimiento contiene dos acciones asociadas:

- onentry se ejecuta una única vez cuando el estado se vuelve activo. En su interior se calcula la velocidad apropiada para efectuar el movimiento, la cual depende de la posición del tag en el eje correspondiente.
- onexit se ejecuta una sola vez cuando el estado deja de estar activo. Dentro suyo se realiza la maniobra de «freno» con una rapidez proporcional a la del movimiento original.

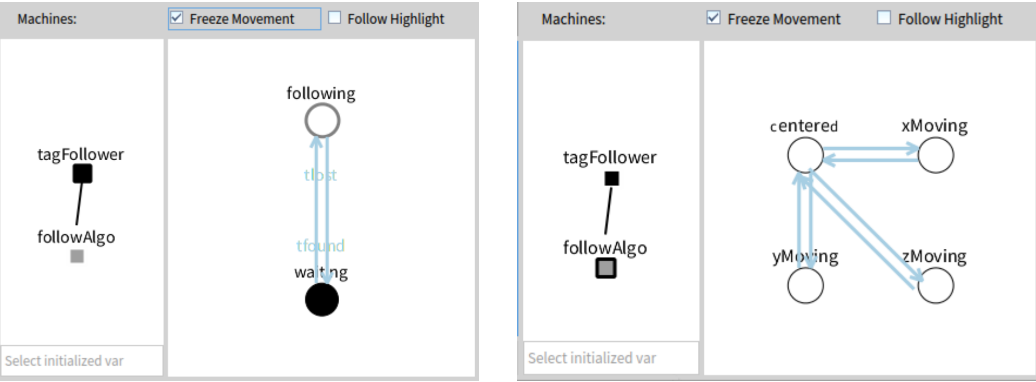


Figura 29: Diagrama de estados de las máquinas tagFollower y followAlgo, versión compleja

10.1.2. Programa Oriented roundel navigator

El ejemplo Oriented roundel navigator habilita al drone para seguir, de forma autónoma, una ruta definida por tags de tipo «Oriented roundel» dispuestos en el suelo. Hay que recordar que el drone es capaz de orientarse automáticamente según la posición de un tag de este tipo (ver sección 7.3.4, apartado «Notas sobre la configuración de los modos de vuelo»). La ruta podrá componerse virtualmente de tantos tags como se desee, siempre y cuando cada tag apunte en la dirección del que lo procede y todas las marcas se encuentren equiespaciadas.

El drone deberá despegar en las cercanías del primer tag, lo detectará con su cámara inferior, se orientará respecto a éste y luego avanzará hacia el frente lo suficiente para encontrar el siguiente tag. La distancia abarcada en cada desplazamiento es fija. Los valores que la definen han sido ajustados para tags separados por 180cm y para un ángulo máximo de inclinación del drone de 12 grados.

La figura 30 expone la máquina de estados principal, tagNavigator, junto a la máquina anidada lookAlgo. El funcionamiento de tagNavigator es sencillo: en el estado takingoff el drone despega, luego asciente hasta una altura definida. A continuación se entra en un ciclo infinito de búsqueda tag actual y orientación respecto a éste (estado looking), estabilización del vuelo (estado stabilize) y desplazamiento hacia la siguiente marca (esado move).

De todos los estados que componen la máquina principal, looking es el más complejo. Es por esto que contiene en su interior la máquina anidada lookAlgo, cuyo funcionamiento se explicará más adelante.

Un video de esta solución puede ser encontrado en la url `http://bit.ly/ARDroneLRP2`

El código es explicado a continuación.

```
01(var altitude := [700]) (var maxaltitude := [900]) (var vspeed := [0.3])
02(var torient := [5000]) (var tstabilize := [5000]) (var tlook := [5000])
03(var tmove := [700]) (var ttakeoff := [5000]) (var pitch := [-0.25])
04(var e := [20]) (var tag := [robot tagDetected: ARDConstant detectedTag_OR_IN_CV])
```

Las primeras 4 líneas definen variables útiles para el programa.

- `altitude` corresponde a la altitud a la que se elevará el drone después de despegar, y que mantendrá durante toda la ejecución, excepto cuando no sea posible vislumbrar ningún tag, en cuyo caso se elevará hasta alcanzar `maxaltitude`.
- `maxaltitude` define la altura a la que el drone se elevará cada vez que se encuentre buscando un tag, pero no pueda encontrarlo.
- `vspeed` es la rapidez vertical a la que el drone se elevará o descenderá.

Las siguientes 5 variables definen plazos temporales en milisegundos:

- `torient` es el tiempo máximo que se esperará desde que el drone detecta el tag, hasta que logra orientarse respecto a éste.

- `tstabilize` es el tiempo que se esperará desde que el drone se ha orientado respecto al tag, hasta que comienza a moverse hacia el tag siguiente.

- `tlook` indica cuánto tiempo se esperará en estado hover, hasta que algún tag sea detectado, si ninguno se detecta después de este plazo, el drone ascenderá hasta `maxaltitude`. `tmove` es el tiempo que el drone se mantendrá en movimiento hacia el frente, para navegar desde un tag hasta el siguiente.

- `ttakeoff` será el tiempo que se esperará desde que se envía la instrucción de despegue hasta que el drone se encuentre volando efectivamente.

Las últimas dos variables definidas son:

- `pitch` indica el valor de inclinación para pitch que se usará para mover al drone hacia adelante

- `e` expresa el margen de error considerado al comparar la altura real informada por el drone con las alturas deseadas. Esto para efectos de gatillar eventos.

```
05(machine tagNavigator
06   (state takingoff
07      (onentry [ robot takeOff. robot modeHoverOnTopOfOrientedRoundel. ]))
08   (state ascending
09      (onentry [ robot elevate: vspeed])
10      (onexit [robot stabilize]))
11   (state descending
12      (onentry [ robot elevate: vspeed negated])
13      (onexit [robot stabilize]))
14   (state stabilizing
15      (onexit [robot modeFreeFlight]))
16   (state moving
17      (onentry [robot pitch: pitch ])
18      (onexit [robot stabilize. robot modeHoverOnTopOfOrientedRoundel ]))
19   (state looking
```

En las líneas anteriores se parte con la definición de la máquina tagNavigator. Ésta contiene los siguientes estados:

- takingoff es el estado de partida de la máquina, cuando se activa hace que el drone despegue y entre en el modo de vuelo «Hover on top of roundel». Este modo de vuelo viene programado en el firmware del drone, y lo habilita para orientarse respecto a la posición del tag, de manera que el círculo del tag quede a su izquierda y la línea, a su derecha.

- ascending se activa una vez que el drone ha despegado, ascending eleva al drone hasta que se alcanza la altura altitude.

- descending cumple la misión de disminuir la altura del drone hasta altitude, en caso de que se encuentre volando muy alto después de haber estado buscando un tag.

- stabilizing se activa una vez que el tag ha sido encontrado y el drone se ha orientado respecto a éste. Existe sólo para dar tiempo al drone de estabilizar su movimiento antes de empezar a moverse hacia el frente para buscar el siguiente tag. Este estado se mantiene activo durante tstabilize milisegundos. Cuando se desactiva, hace entrar al robot en modo de vuelo «Free flight», lo que le permite abandonar el tag actual y avanzar hacia el siguiente.
- moving, que es el estado en el cual el drone se desplaza entre tags. Se mantiene activo durante tmove milisegundos. Cuando moving se desactiva, además de estabilizar el movimiento del robot, hace que el robot entre en modo «Hover on top of roundel» para así buscar el siguiente tag.

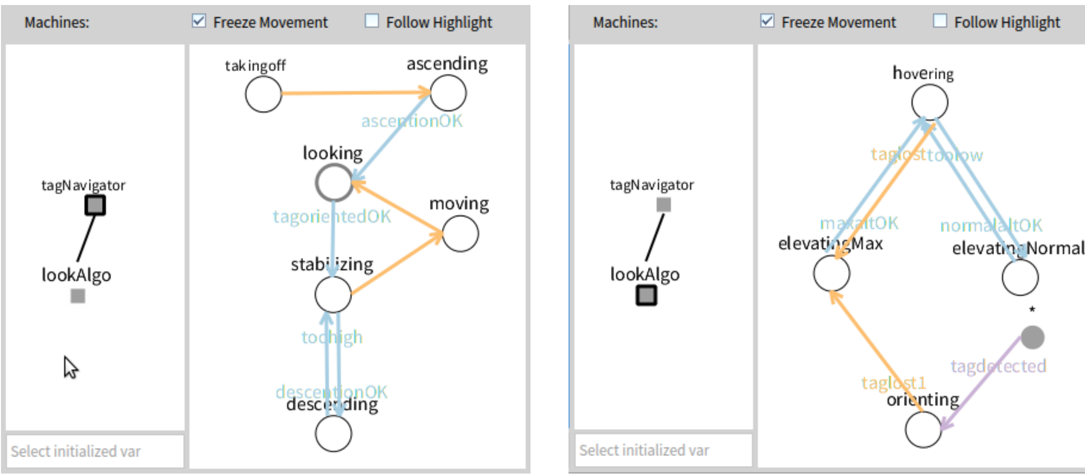


Figura 30: Diagrama de estados de las máquinas tagNavigator y lookAlgo

```
20      (machine lookAlgo
21         (state hovering)
22         (state elevatingNormal
23            (onentry [ robot elevate: vspeed. robot ledsOrange])
24            (onexit [robot stabilize. robot ledsStandard]))
25         (state elevatingMax
26            (onentry [ robot elevate: vspeed. robot ledsRed])
27            (onexit [robot stabilize. robot ledsStandard]))
28         (state orienting)
```

La máquina anidada lookAlgo contiene la lógica para detectar los tags y orientarse respecto a ellos. Mientras la máquina lookAlgo se encuentre activa, el drone estará en modo de vuelo «Hover on top of oriented roundel», y se mantendrá así hasta que se active el estado moving de la máquina principal.

La lógica de búsqueda es la siguiente: la máquina parte en estado hovering. Si transcurridos tlook milisegundos no se ha detectado ningún tag, se pasará al estado elevatingMax, donde el drone se elevará hasta maxaltitude y de este modo la imagen de su cámara inferior cubrirá un área mayor del suelo. Cuando la altura maxaltitude se alcance, volverá a activarse el estado hover, el cual permanecerá activo hasta que algún tag sea detectado.

Si en cualquier momento un tag es detectado, el estado orienting será activado de inmediato, dándole tiempo al drone para orientarse respecto a la marca en el piso.

Ahora que se conoce el funcionamiento de la máquina, comprender el rol de cada uno de sus estados es más sencillo:

- hovering: es el estado inicial de la máquina, no contiene ninguna acción, su propósito es dar tiempo al drone para detectar algún tag debajo suyo.
- elevatingNormal: en su interior el drone se eleva hasta alcanzar altitude. Se activa cuando el drone ha perdido altura producto de maniobras de orientación o movimiento.
- elevatingMax: en su interior el drone se eleva hasta alcanzar maxaltitude. Se activa cuando han transcurrido tlook milisegundos y el tag aún no es detectado.
- orienting: se activa cuando el tag el detectado, da tiempo al drone para orientarse.

```
29          (event flyingTooLow [(robot altitude + e) < altitude])
30          (event normalAltitudeReached [robot altitude >= altitude])
31          (event maxAltitudeReached [robot altitude >= maxaltitude])
32          (on flyingTooLow hovering->elevatingNormal toolow)
33          (on normalAltitudeReached elevatingNormal -> hovering normalaltOK)
34          (on maxAltitudeReached elevatingMax -> hovering maxaltOK)
35          (on [robot tag detected] *-> orienting tagdetected)
36          (ontime torient orienting -> elevatingMax taglost1)
37          (ontime tlook hovering -> elevatingMax taglost))
38      (onentry (spawn lookAlgo hovering)))
```

La máquina lookAlgo finaliza su definición con las lineas anteriores. Los eventos flyingTooLow, normalAltitudeReached y maxAltitudeReached activan respectivamente las transiciones toolow, normalaltOK y maxaltOK, todas estas transiciones guardan relación los mecanismos ya descritos para controlar la altura del drone mientras se se encuentra en estado hover. La transición tagdetected tiene importancia especial, ya es de tipo wildcard. Esto quiere decir que cuando el robot haya detectado el tag roundel, no importando el estado actual de la máquina lookAlgo, orienting pasará a ser el estado activo.

La transición temporal taglost1 indica que orienting permanecerá activo por un máximo de torient milisegundos, ya que si este plazo se cumple el estado elevatingMax será activado (es decir, se considerará que el tag como perdido y se harán esfuerzos para volverlo a encontrar). La transición taglost es similar a la anterior, pero se aplica al estado hovering y determina que éste estará activo durante un máximo de tlook milisegundos. La última línea determina que cuando el estado padre looking se active, la máquina lookAlgo se creará y su primer estado activo será hovering). Aquí finaliza el estado looking, el código restante corresponde al resto de la definición de la máquina principal tagNavigator.

```
39    (event oriented [robot tag isOrientedIn: 0 withTolerance: 0.1])
40    (event ascentionDone [(robot altitude + e/2) >= altitude])
41    (event descentionDone [(robot altitude - e/2) <= altitude])
42    (event tooHighAfterLooking [robot altitude - e > altitude])
43    (on oriented looking -> stabilizing tagorientedOK)
44    (on tooHighAfterLooking stabilizing->descending toohigh)
45    (on ascentionDone ascending -> looking ascentionOK)
46    (on descentionDone descending-> stabilizing descentionOK)
47    (ontime ttakeoff takingoff -> ascending)
48    (ontime tstabilize stabilizing -> moving)
49    (ontime tmove moving -> looking))
50(spawn tagNavigator takingoff)
```

Entre las líneas 39 y 50 se definen los eventos y transiciones de la máquina tagNavigator. Los eventos ascentionDone, descentionDone y tooHighAfterLooking participan en los mecanismos de control de la altura inicial del drone, pues mientras el estado looking no esté activo, se desea conservar al drone a altura altitude. El evento oriented, de la línea 39, es muy relevante, ya que determina el instante en que looking se desactiva y se pasa al estado stabilizing. Lo anterior se evidencia en la transición tagorientedOK de la línea 43. Hay que notar que este es el evento que evitará que la transición temporal taglost1, de la máquina anidada lookAlgo, se active y por ende se prosiga la búsqueda del tag.

La última línea del ejemplo define que el primer evento activo de la máquina principal sea takingoff.

10.2. Lecciones Adquiridas

La realización del presente trabajo de memoria hizo surgir ideas nuevas en relación a sus objetivos originales, las cuales pueden ser separadas en dos temas: conclusiones sobre el funcionamiento de LRP con el AR.Drone 2.0, y recomendaciones relacionadas al trabajo con el AR.Drone 2.0.

Respecto al primer tema se ha concluído que las herramientas visuales y la sintaxis ofrecidas por LRP aportaron positivamente a la experiencia de programación con el drone, pues facilitaron enormemente la comprensión del código durante su creación, y facilitaron además el seguimiento de su estado y detección de errores durante la ejecución. Por otro lado, se constató que es difícil aprovechar plenamente las ventajas ofrecidas por la modalidad live programming, ya que si bien ésta resultó ser muy útil para ajustar valores correspondientes a variables que se encontraran al interior del código de un programa ya creado, en la práctica resultó muy difícil programar nuevos comportamientos mientras el drone se encontrara en vuelo. Existen dos causas principales para lo anterior. En primer lugar, el drone se mueve muy rápido. Esto ocurre debido a que el umbral de inclinación del drone para lograr desplazamiento, es muy elevado, lo cual deriva en que su rapidez mínima sea muy elevada. La segunda causa se debe a que cada vez que el drone se estrella, la sesión de vuelo se interrumpe totalmente, ya que el programador debe levantarse de su lugar para reposicionar al robot y volverlo a despegar. Las dificultades anteriores son inherentes a los robots aéreos y live programming, y no se deben a ninguna particularidad de LRP o del AR.Drone 2.0.

Como un agregado a las ideas extraídas a partir de las pruebas realizadas con LRP, la experiencia completa aportó con conocimiento específico sobre el AR.Drone 2.0 que no se encontraba documentado y que ha sido destacado a lo largo de este documento en frases dispuestas al interior de recuadros azules. Este conocimiento permitirá que trabajos futuros que utilicen el drone, puedan ser elaborados con una mayor fluidez. Complementariamente se han definido recomendaciones generales para el trabajo con el AR.Drone 2.0:

- Priorizar el tiempo de autonomía del robot (duración de la batería)
- Contar con un espacio amplio, con reducida contaminación WiFi y buena iluminación (una iluminación insuficiente afectará negativamente la estabilidad del drone)
- Contar con una textura de alto contraste en el piso, pues ayuda a la estabilidad de vuelo. Es sencillo crear una textura adecuada disponiendo marcas de cruz, separadas a unos 50 cm entre sí, con cinta adhesiva de un color que contraste con el del piso.
- Priorizar en el orden del desarrollo, la implementación de funciones para aterrizar o entrar en modo de emergencia.
- Contar con un mecanismo de reconexión automática.
- Considerar el estado del parámetro de configuración «outdoor_flying», porque modifica los valores de inclinación y rapidez máxima.

Capítulo III.

Conclusión y Trabajo Futuro

11. Conclusión

La robótica, y en particular la programación de robots, es un campo muy joven que actualmente demanda la creación de herramientas convenientes para desarrolladores e investigadores del área. Recientemente investigadores del DCC han concebido un nuevo lenguaje de programación para robots llamado LRP, caracterizado por adoptar la modalidad de programación en vivo. Previo a la realización de este trabajo, LRP fue probado con diferentes robots, pero nunca con uno de características similares a las del AR.Drone 2.0 de Parrot.

Para poder hacer pruebas desde LRP con el drone, se construyó una API de comunicación y una aplicación tipo puente requerida por LRP, ambas escritas en Pharo. Las funcionalidades de la API abarcan la capacidad de establecer conexión con el drone, la recepción de sus datos de navegación y de configuración, y el envío de comandos de control a través de los cuales es posible manipular los movimientos del drone y además alterar su configuración interna. La API incorpora en su interior una interfaz de usuario que permite controlar los movimientos, establecer animaciones LED y realizar piruetas.

La principal función del puente es adaptar una serie de métodos de la API para comunicarse con el drone, de modo que estén disponibles desde LRP. Concretamente, el puente corresponde a un caso del patrón de diseño *«Adapter»*. El puente provee adicionalmente una interfaz de usuario que se despliega al iniciar el intérprete de LRP, proporcionando mecanismos intuitivos para conectarse con el drone, despegarlo, aterrizarlo, mostrar el stream de video transmitido por cualquiera de las cámaras, visualizar sus datos de navegación y configuración, entre otros.

La mayoría de las dificultades enfrentadas durante el desarrollo de la API se debieron a la presencia de características o requerimientos indocumentados, impuestos por el firmware del drone.

La experiencia permitió generar conocimiento técnico específico sobre el AR.Drone, que no siempre se encontró presente en la documentación oficial de Parrot, relevante para futuros ingenieros o investigadores que deseen trabajar con el robot. Estas ideas han sido destacadas a lo largo de todo el texto, bajo la etiqueta «Lección aprendida» y se espera que contribuyan a hacer más expedito el proceso de desarrollo en trabajos posteriores.

Los programas de ejemplo creados evidenciaron que no todas las características de la modalidad de programación en vivo pueden ser aprovechadas cuando se trabaja con un robot aéreo. Esto es así porque es difícil restringir los límites en los que el robot se mueve, porque el tiempo de autonomía del robot es muy limitado y porque la interrupción de una sesión, producto de un choque por ejemplo, implica que el desarrollador deba interrumpir su trabajo para reposicionar y despegar el robot. Sin embargo, live programming permitió ajustar las variables asociadas a programas ya creados, lo que adquiere gran valor al momento de establecer distancias que se espera que el robot recorra. Esta característica fue muy aprovechada durante las ejecuciones del ejemplo Tag Navigator, esto porque permitió ajustar durante el vuelo, la distancia recorrida por el drone según la separación de los tags. Aparte de lo anterior, la sintaxis de LRP y sus diagramas de estado animados en vivo facilitaron el proceso de desarrollo.

Al momento de dar término a este trabajo de memoria los objetivos globales propuestos para ella se consideran cumplidos. Esto porque fue posible realizar pruebas utilizando LRP con

el robot AR.Drone 2.0 y generar conocimiento nuevo a partir de la experiencia. No obstante, la experimentación con robots aéreos y Live Programming es un tema con gran potencial de desarrollo, sobre la cual aún queda conocimiento por adquirir.

12. Trabajo Futuro

El trabajo de memoria presentado en este documento puede ser extendido de diversas formas, ya sea incrementando la complejidad de las herramientas desarrolladas, o implementando nuevas herramientas complementarias que resuelvan problemas asociados al trabajo con robots aéreos.

En relación a esto último, se propone la creación de un mecanismo que limite el rango de vuelo del drone y automatice su retorno a un punto definido, utilizando el complemento GPS flight recorder compatible con AR.Drone 2.0. Si se contara con dicho complemento GPS, además sería posible extender la API para que publicara la posición del drone respecto a un sistema de coordenadas definido, a través de una interfaz pública. Estos datos podrían ser consultados desde algún programa externo que tuviese modelado un terreno 3D y que pudiese calcular rutas óptimas para mover al drone entre dos puntos establecidos.

Se propone también mejorar el sistema de detección de tags incorporado en la API implementada en Pharo, para lograr que el drone pueda seguir un tag incluso si éste se escapa del campo de visión momentáneamente. Esto podría lograrse almacenando las últimas posiciones asociadas al tag y calculando su vector velocidad en cada momento. Complementariamente, el sistema de detección de tags podría mejorarse implementando tolerancias adaptativas para considerar al tag centrado, dependientes de la distancia a la que se encuentre el tag respecto del drone.

Bibliografía

[1] J. Fabry and M. Campusano, “Live robot programming,” in *Advances in Artificial Intelligence – IBERAMIA 2014*, ser. Lecture Notes in Computer Science, A. L. Bazzan and K. Pichara, Eds. Springer International Publishing, 2014, vol. 8864, pp. 445–456. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-12027-0_36

[2] S. Ducasse, M. Denker, and D. Pollet, “Pharo’s Vision: Goals, Processes, and Development Effort [Article].”

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[4] Tutrtlebot 2: Open source robot development kit for apps on wheels. [Online]. Available: http://www.turtlebot.com/ (Accessed 2015-12-13).

[5] Willow garage: Pr2 overview. [Online]. Available: https://www.willowgarage.com/pages/pr2/overview (Accessed 2015-12-13).

[6] The lego group: Lego mindstorms education ev3. [Online]. Available: https://education.lego.com/mindstorms (Accessed 2015-12-13).

[7] Parrot: Ar.drone 2.0. [Online]. Available: http://ardrone2.parrot.com (Accessed 2015-12-13).

[8] T. Krajník, V. Vonásek, D. Fišer, and J. Faigl, “Ar-drone as a platform for robotic research and education,” in *Research and Education in Robotics-EUROBOT 2011*. Springer, 2011, pp. 172–186.

[9] Parrot for developers: Ar drone. [Online]. Available: http://developer.parrot.com/ar-drone.html (Accessed 2015-12-13).

[10] Parrot developer zone: Some developers experiences in video. [Online]. Available: http://ardrone2.parrot.com/developer-zone/ (Accessed 2015-12-13).

[11] The Gerstner Laboratory for Intelligent Decision Making and Control. Department of Cybernetics, Faculty of Electrical Engineering. Czech Technical University in Prague. ’Software for ARDrone control’. [Online]. Available: http://labe.felk.cvut.cz/~tkrajnik/ardrone/\#download (Accessed 2015-12-13).

[12] T. G. Corporation, *Press Release: UAV Production Will Total 93 Billion*, 2015. [Online]. Available: http://www.tealgroup.com/index.php/teal-group-news-media/item/press-release-uav-production-will-total-93-billion

[13] T. N. Waharte S., “Supporting search and rescue operations with uavs,” in *Emerging Security Technologies (EST), 2010 International Conference on*. IEEE, September 2010, pp. 142–147.

[14] H. Saari, I. Pellikka, L. Pesonen, S. Tuominen, J. Heikkila, C. Holmlund, J. Makynen, K. Ojala, and T. Antila, "Unmanned aerial vehicle (uav) operated spectral camera system for forest and agriculture applications," pp. 81 740H–81 740H–15, 2011.

[15] "Hollywood welcomes drones to the set," January 2015, [Online; posted 01-January-2015]. [Online]. Available: http://edition.cnn.com/videos/entertainment/2015/01/23/drones-hollywood-filming-orig.cnn/video/playlists/all-things-drones/

[16] "New use for drones: Sports photography," September 2012, [Online; posted 22-September-2012]. [Online]. Available: http://www.cbsnews.com/news/new-use-for-drones-sports-photography/

[17] Conservationdrones.org. [Online]. Available: http://conservationdrones.org/mission/ (Accessed 2015-10-25).

[18] K. Lee, "Development of unmanned aerial vehicle (uav) for wildlife surveillance," Ph.D. dissertation, University of Florida, 2004.

[19] U. C. Ofoma and C. C. Wu, "Design of a fuel cell powered uav for environmental research," in *Proceedings of the AIAA 3rdÜnmanned... Unlimited"Technical Conference, Workshop, and Exhibit, September*, 2004, pp. 20–23.

[20] D. C. English, S. Herwitz, C. Hu, P. R. Carlson, Jr., F. E. Muller-Karger, K. K. Yates, and D. Ramsewak, "Challenges in collecting hyperspectral imagery of coastal waters using Unmanned Aerial Vehicles (UAVs)," *AGU Fall Meeting Abstracts*, p. A1738, Dec. 2013.

[21] Professional society of drone journalists (psdj). [Online]. Available: http://www.dronejournalism.org/ (Accessed 2015-10-25).

[22] Matternet. [Online]. Available: http://mttr.net/company (Accessed 2015-10-25).

[23] Aerovironment, *Press Release: AeroVironment to Deploy Small Unmanned Aircraft for Federal Communications Commission Post-Disaster Communications Demonstration*, 2012. [Online]. Available: http://www.avinc.com/resources/press_release/aerovironment_to_deploy_small_unmanned_aircraft_for_federal_communications_

[24] K. V. Mellinger D., Michael N. Quadrotors. Youtube. [Online]. Available: https://www.youtube.com/watch?v=DzmPbVn1KU8

[25] R. D'Andrea. La asombrosa potencia atletica de los cuadricopteros. Youtube. [Online]. Available: https://www.youtube.com/watch?v=w2itwFJCgFQ (Accessed 2015-12-20).

[26] N. Sydney, B. Smyth, D. Paley *et al.*, "Dynamic control of autonomous quadrotor flight in an estimated wind field," in *Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on*. IEEE, 2013, pp. 3609–3616.

[27] K. Alexis, G. Nikolakopoulos, and A. Tzes, "Experimental model predictive attitude tracking control of a quadrotor helicopter subject to wind-gusts," in *Control & Automation (MED), 2010 18th Mediterranean Conference on*. IEEE, 2010, pp. 1461–1466.

[28] K. Sreenath and V. Kumar, "Dynamics, control and planning for cooperative manipulation of payloads suspended by cables from multiple quadrotor robots," *rn*, vol. 1, no. r2, p. r3, 2013.

[29] H. Khebbache, B. Sait, N. Bounar, and F. Yacef, "Robust stabilization of a quadrotor uav in presence of actuator and sensor faults," *International Journal of Instrumentation and Control Systems*, vol. 2, no. 2, pp. 53–67, 2012.

[30] T. Krajník, M. Nitsche, S. Pedre, L. Přeučil, and M. E. Mejail, "A simple visual navigation system for an uav," in *Systems, Signals and Devices (SSD), 2012 9th International Multi-Conference on*. IEEE, 2012, pp. 1–6.

[31] C. Bills, J. Chen, and A. Saxena, "Autonomous mav flight in indoor environments using single image perspective cues," in *Robotics and automation (ICRA), 2011 IEEE international conference on*. IEEE, 2011, pp. 5776–5783.

[32] T. Nguyen, G. K. I. Mann, and R. G. Gosine, "Vision-based qualitative path-following control of quadrotor aerial vehicle with speeded-up robust features," in *Proceedings of the 2014 Canadian Conference on Computer and Robot Vision*, ser. CRV '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 321–327. [Online]. Available: http://dx.doi.org/10.1109/CRV.2014.50

[33] J.-D. Fossel, D. Hennes, S. Alers, D. Claes, and K. Tuyls, "Octoslam: A 3d mapping approach to situational awareness of unmanned aerial vehicles," in *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, ser. AAMAS '13. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2013, pp. 1363–1364. [Online]. Available: http://dl.acm.org/citation.cfm?id=2484920.2485226

[34] D. Mellinger, A. Kushleyev, and K. V. A swarm of nano quadrotors. Youtube. [Online]. Available: https://www.youtube.com/watch?v=YQIMGV5vtd4 (Accessed 2015-12-15).

[35] F. Augugliaro, A. Mirjan, F. Gramazio, M. Kohler, and R. D'Andrea, "Building tensile structures with flying machines," in *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*. IEEE, 2013, pp. 3487–3492.

[36] F. Augugliaro, S. Lupashin, M. Hamer, C. Male, M. Hehn, M. W. Mueller, J. S. Willmann, F. Gramazio, M. Kohler, and R. D'Andrea, "The flight assembled architecture installation: Cooperative construction with flying machines," *Control Systems, IEEE*, vol. 34, no. 4, pp. 46–64, 2014.

[37] F. Auguglario. Building a rope bridge with flying machines. Youtube. [Online]. Available: https://www.youtube.com/watch?v=CCDIuZUfETc (Accessed 2015-12-15).

[38] M. Saska, V. Vonásek, T. Krajník, and L. Přeučil, "Coordination and navigation of heterogeneous uavs-ugvs teams localized by a hawk-eye approach," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE, 2012, pp. 2166–2171.

[39] M. Saska, V. Vonásek, T. Krajník, and L. Přeučil. Formation control with an ar-drone. Youtube. [Online]. Available: https://www.youtube.com/watch?v=JC2EPBQ8WEY

[40] M. Saska, V. Vonásek, T. Krajník, and L. Přeučil. Heterogeneous formation of mobile robots and uav. Youtube. [Online]. Available: https://www.youtube.com/watch?t=1\&v=A_K7GQ65_g4 (Accessed 2015-12-15).

[41] S. Nicholls. Ardronedocs: Tecnical specifications. Github. [Online]. Available: https://github.com/seannicholls/ARDroneDocs/wiki/Technical-Specifications (Accessed 2015-10-25).

[42] J. Rand and T. SNARC, "AR. Pwn: Hacking the Parrot AR. Drone [Article]."

[43] Aeroquad, the open source quadcopter / multicopter. [Online]. Available: http://aeroquad.com/forum.php (Accessed 2015-10-25).

[44] 3dr, software, we develop powerful software that makes using a drone easy, fun and intuitive for users and developers alike. [Online]. Available: https://3drobotics.com/software/ (Accessed 2015-10-25).

[45] Autoquad is an open source firmware project with closed source hardware. [Online]. Available: http://autoquad.org/ (Accessed 2015-10-25).

[46] Openpilot: The next generation open source uav autopilot. [Online]. Available: https://www.openpilot.org/ (Accessed 2015-10-25).

[47] Parrot for developers. ar drone. [Online]. Available: http://developer.parrot.com/ar-drone.html (Accessed 2015-10-25).

[48] Ffmpeg: Documentation. [Online]. Available: http://ffmpeg.org/documentation.html (Accessed 2015-10-25).

[49] Parrot android academy. [Online]. Available: http://ardrone2.parrot.com/ar-drone-academy/ (Accessed 2015-10-25).

[50] V. Zaliva. Java api and demo programs to control parrot's ar.drone. Github. [Online]. Available: https://github.com/codeminders/javadrone

[51] T. Krajnik. Software for ardrone control (c++). [Online]. Available: http://labe.felk.cvut.cz/~tkrajnik/ardrone/restricted.html

[52] R. Duran. Opendronecontrol [odc] is an open source software platform for developing interactive artworks and research projects with aerial robotics. Github. [Online]. Available: https://github.com/opendronecontrol/odc

[53] B. Venthur. Python-ardrone. readme. Github. [Online]. Available: https://github.com/venthur/python-ardrone/blob/master/README.md

[54] J. Weirich. Ruby api for controlling a parrot ar drone. Github. [Online]. Available: https://github.com/jimweirich/argus

[55] R. Mehner. A node.js client for controlling parrot ar drone 2.0 quad-copters. Github. [Online]. Available: https://github.com/felixge/node-ar-drone

[56] Mac developer library. memory management programming guide for core foundation. byte ordering (little-endian and big-endian). [Online]. Available: https://developer.apple.com/library/mac/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/Concepts/ByteOrdering.html (Accessed 2015-10-25).

[57] S. L. Tanimoto, "Viva: A visual language for image processing," *Journal of Visual Languages & Computing*, vol. 1, no. 2, pp. 127–139, 1990.

[58] B. V. CUSEC. Inventing on principle. Vimeo. [Online]. Available: https://vimeo.com/36579366

[59] B2b service robots robosoft, robulab. [Online]. Available: http://www.robosoft.com/products/indoor-mobile-robots/robulab/index.html (Accessed 2016-06-10).

[60] Knight robotics. [Online]. Available: http://www.knightrobotics.cl/ (Accessed 2016-06-10).

Anexos

A. Resumen de mensajes públicos de la API en Pharo

A un usuario de la API en Pharo para el AR.Drone 2.0, le bastará utilizar los mensajes públicos de la API para controlar al drone. Todos los mensajes presentados forman parte de la clase ARDrone.

A.1. Instancia de la clase ARDrone y conexión con el drone

Mensaje	Descripción	Argumentos
`uniqueInstance` (class side)	Retorna la única instancia disponible de la clase.	-
`newUniqueInstance` (class side)	Instancia la clase y guarda el objeto resultante en la variable de clase uniqueInstance. Retorna uniqueInstance.	-
`startSession`	Inicia los procesos asociados a uniqueInstance, e inicia la conexión con el drone.	-
`terminateSession`	Finaliza todos los procesos asociados a la instancia. Es responsabilidad del programador llamar a este método. De no hacerlo, quedarán procesos ejecutándose en segundo plano.	-
`resetSession`	Reinicia los procesos asociados a uniqueInstance y también la conexión.	-

A.2. Control de movimiento

Mensaje	Descripción	Argumentos
`takeOff`	Despega el drone. No tiene efecto si el drone está volando.	-
`land`	Aterriza el drone. No tiene efecto si el drone está aterrizado.	-
`elevate: aVerticalSpeed`	Establece la velocidad vertical del drone. Su valor persistirá hasta que el usuario le asigne uno nuevo.	- `aVerticalSpeed` - float $\in [-1,1]$, es un porcentaje de la velocidad vertical máxima, (+) asciende, (-) desciende
`roll: aLeftRightTilt`	Establece la inclinación lateral del drone. Su valor persistirá hasta que el usuario le asigne uno nuevo.	- `aLeftRightTilt` - float $\in [-1,1]$, es un porcentaje de la inclinación lateral máxima, (+) para inclinar a la derecha, (-) para inclinar a la izquierda
`pitch: aFrontBackTilt`	Establece la inclinación frontal del drone. Su valor persistirá hasta que el usuario le asigne uno nuevo.	- `aFrontBackTilt` - float $\in [-1,1]$, es un porcentaje de la inclinación frontal, (-) para bajar la nariz, (+) para subir la nariz
`yaw: anAngularSpeed`	Establece la velocidad angular del drone. Su valor persistirá hasta que el usuario le asigne uno nuevo.	- `anAngularSpeed` - float $\in [-1,1]$, es un porcentaje de la velocidad angular máxima, (+) gira a la derecha, (-) gira a la izquierda
`moveByLeftRightTilt: aLeftRightTilt frontBackTilt: aFrontBackTilt angularSpeed: anAngularSpeed verticalSpeed: aVerticalSpeed`	Establece de una vez la inclinación lateral y frontal, y la velocidad vertical y angular del drone. Todos los valores persistirá hasta que el usuario le asigne nuevos valores.	- `aLeftRightTilt` - `aFrontBackTil` - `anAngularSpeed` - `aVerticalSpeed` equivalentes a los argumentos de las 4 filas anteriores
`stabilize`	Hace cero todos los valores de inclinación y velocidad, estabilizando al drone.	
`animateFlight: aFlightAnimNumber duration: aDuration`	Hace que el drone ejecute una rutina de vuelo predefinida durante el tiempo determinado.	- `aFlightAnimNumber` integer (enumeración), define el tipo de pirueta - `aDuration` integer, unidad: milisegundos

Ninguno de estos métodos es bloqueante

A.3. Animación de LEDs

Mensaje	Descripción	Argumentos
`animateLEDs: aLEDAnimNumber frequency: aFrequency duration: aDuration`	Anima los LED del drone según un patrón asociado a aLEDAnimationNumber, con una frecuencia y durante un tiempo específicos.	- `aLEDAnimNumber` integer (enumeración), define el tipo de animación - `aFrequency` float, unidad: Hz - `aDuration` integer, unidad: segundos

A.4. Consulta de datos de navegación

Mensaje	Descripción	Argumentos
`lastNavdataPacketReceived`	Permite acceder al objeto ARDNavigationDatagramDemo que contiene los datos del último paquete de navegación recibido (estos datos pueden verse en la figura 21). El resto de los métodos de esta tabla es redundante, se incluyen porque son requeridos con frecuencia.	-
`batteryLevel`	Retorna un entero que indica el porcentaje de batería restante del drone.	-
`lowBattery`	Retorna un boolean que indica si el drone ha alertado por batería baja.	-
`isFlying`	Retorna un boolean que indica si el drone se encuentra volando.	-
`isInEmergencyState`	Retorna un boolean que indica si el drone se encuentra en estado de emergencia.	-
`setNavdataReceiverCallback: aBlock`	Reemplaza el proceso de recepción de navdata por otro que incorpora la evaluación del bloque aBlock cada vez que un paquete es recibido.	- `aBlock` debe recibir como parámetro un datagrama de navegación

A.5. Consulta y modificación de la configuración interna del drone

Mensaje	Descripción	Argumentos
`droneGet ConfigurationFor: aParameter`	Retorna el valor asociado al parámetro de configuración aParameter, según la configuración interna del drone.	- `aParameter` string (asociado con un parámetro de configuración, ej: 'general: num_version_config')
`droneSet ConfigurationFor: aParameter value: aString`	Modifica la configuración interna del drone, estableciendo el valor aString al parámetro aParameter.	- `aParameter` string (asociado con un parámetro de configuración, ej: 'general: num_version_config') - `aString` string, valor para el parámetro
`retreive Configuration ForDrone`	Solicita el archivo de configuración del drone. Si la recepción es exitosa el archivo se almacenará en la variable `configuration` de la instancia de `ConfigurationManager` contenida en `ARDrone`.	-
`configFlyingMode FreeFlight`	Establece el modo de vuelo en Free flight, configurando la detección de tag en modo de detección Múltiple y habilitando la detección del tipo Shell tag para la cámara horizontal, y del tipo Oriented Roundel tag para la cámara vertical.	-
`configFlyingMode HoverOnTopOf OrientedRoundel`	Establece el modo de vuelo en Hover on top of oriented roundel, configurando adecuadamente la detección de tags.	-
`changeVideoChannel`	Cambia la cámara desde la que se envía el stream de video.	-

A.6. Detección de tags

Mensaje	Descripción	Argumentos
tagsTotalDetected	Retorna el número de tags que está siendo detectado en cada momento.	-
tagDetected: aDetectedTagType	Retorna el objeto ARDDetectedTag que representa al tag de tipo aDetectedTagType que puede ser detectado a través de una camara (ver argumento aDetectedTagType). La clase ARDDetectedTag es la responsable de procesar la información de un tag detectado. Las siguientes lineas contienen mensajes compuestos, es decir, representan mensajes pertenecientes a la clase ARDTagDetected, pero que pueden utilizarse directamente. Para mayor información ver sección 7.4, apartado Vision Detector Analyst).	- aDetectedTagType, integer (enumeración). Representa un tag de un tipo específico que es detectado por una cámara particular. ARDConstant define dos valores para esta enumeración: * detectedTag_OR_IN_CV (Oriented roundel detectado en la camara vertical) * detectedTag_SH_IN_CH (Shell tag detectado en la camara horizontal)
tagDetected: aDetectedTagType detected	Retorna un boolean que es verdadero si el tag está siendo detectado por la cámara especificada.	- aDetectedTagType, ídem a tagDetected:
tagDetected: aDetectedTagType detectedWeighted	Retorna un boolean que obtiene al promediar y aproximar los estados de detección informados para el tag por los últimos 3 paquetes recibidos.	- aDetectedTagType, ídem a tagDetected:
tagDetected: aDetectedTagType distance	Retorna la distancia en centímetros a la que se encuentra el tag.	- aDetectedTagType, ídem a tagDetected:
tagDetected: aDetectedTagType height	Retorna la altura del rectángulo que encierra al tag.	- aDetectedTagType, ídem a tagDetected:
tagDetected: aDetectedTagType width	Retorna el ancho del rectángulo que encierra al tag.	- aDetectedTagType, ídem a tagDetected:

Mensaje	Descripción	Argumentos
tagDetected: aDetectedTagType orientationAngle	Retorna un float que contiene el ángulo de orientación, en grados, del tag en la imagen de la camara.	- aDetectedTagType, ídem a tagDetected:
tagDetected: aDetectedTagType isCenteredInX WithTolerance: aTolerance	Retorna un boolean que indica si el tag se encuentra centrado según el eje X, considerando un porcentaje de error definido por aTolerance.	- aDetectedTagType, ídem a tagDetected: - aTolerance float ∈ [0,1], representa un porcentaje de error aceptado
tagDetected: aDetectedTagType isCenteredInY WithTolerance: aTolerance	Ídem al anterior, pero en el eje Y.	- aDetectedTagType, ídem a tagDetected: - aTolerance float ∈ [0,1], representa un porcentaje de error aceptado
tagDetected: aDetectedTagType isAtDistance: aDistance WithTolerance: aTolerance	Retorna un boolean que indica si el tag se encuentra a aDistance centímetros del drone. Considera un porcentaje de error definido por aTolerance.	- aDetectedTagType, ídem a tagDetected: - aDistance integer, distancia en centímetros - aTolerance float ∈ [0,1], representa un porcentaje de error aceptado
tagDetected: aDetectedTagType isOrientedIn: anAngle withTolerance: aTolerance	eRetorna un boolean que indica si el tag se encuentra orientado en anAngle grados en la imagen que lo detecta. Considera un porcentaje de error definido por aTolerance.	- aDetectedTagType, ídem a tagDetected: - anAngle float, ∈ [0,360], ángulo en grados. - aTolerance float ∈ [0,1], representa un porcentaje de error aceptado.

A.7. Configuración de la API y captura de excepciones

Mensaje	Descripción	Argumentos
configReconnect Timeout: anInt	Establece el plazo para intentar recuperar la conexión con el drone cuando ésta se ha interrumpido.	- anInt integer, unidad [segundos]
addHandlerFor: anExceptionClass do: aBlock	Determina que el bloque aBlock se ejecute cada vez que una excepción de clase anExceptionClass ocurra, evitando que la excepción sea arrojada e interrumpa la ejecución del programa.	- anExceptionClass debe ser una clase que extienda exceptions - aBlock bloque de código