UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

Sistema multi-agente para la coordinación de UAVs mediante negociaciones concurrentes

David Frutos Talavera

Julio, 2017

SISTEMA MULTI-AGENTE PARA LA COORDINACIÓN DE UAVS MEDIANTE NEGOCIACIONES CONCURRENTES

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Tecnologías y Sistemas de Información

TECNOLOGÍA ESPECÍFICA DE
TECNOLOGÍAS Y SISTEMAS DE INFORMACIÓN

TRABAJO FIN DE GRADO

Sistema multi-agente para la coordinación de UAVs mediante negociaciones concurrentes

Autor: David Frutos Talavera

Director: Dr. David Vallejo Fernández

Julio, 2017

David Frutos Talavera

Ciudad Real – España

E-mail: David.Frutos.Talavera@gmail.com
Teléfono: 686 133 022

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL	SECRETARIO
Fdo.:	Fdo.:	Fdo.:

Resumen

En este Trabajo de Fin de Grado (TFG) se plantea la realización de un mecanismo de coordinación orientado a los Unmanned Aerial Vehicle (UAV)s o más comúnmente llamados: drones. Dicho proyecto surge a causa de la creciente popularidad que están obteniendo los UAVs y a la necesidad, futura, de mecanismos de coordinación para proporcionar de servicios.

Por este motivos, este proyecto busca desarrollar y diseñar un sistema de coordinación concurrente mediante mecanismos de subastas flexibles centrado en la coordinación de los UAVs. En este proyecto se describe un servicio de vigilancia realizado por UAVs dónde, gracias a un sistema de subastas, se compite por la obtención de un contrato de vigilancia. El sistema define un entorno dónde interactúan múltiples agentes de dos tipos diferentes: los compradores y los vendedores. Los compradores, pueden ser representados por un persona real o por un UAV, mientras que los vendedores siempre son representados por un UAV. Estos agentes compiten, según sus roles, por la venta u obtención de un servicio.

Tanto para la realización de este proyecto como para las pruebas y casos de estudio pertinentes se han utilizado cuatro UAVs. Estos UAVs son del modelo *Cargo* de la empresa *Parrot* y su inteligencia ha sido programada gracias a su capacidad de conexión vía Bluetooth Low Energy (BLE). Gracias a esto, se ha podido desarrollar un entorno dónde los UAVs son controlados mediante un Ground Control Station (GCS) e interactúan entre ellos y el comprador mientras realizan sus ofertas y contra ofertas.

Gracias a este proyecto se espera obtener el diseño de un sistema multi-agente para la coordinación de UAVs mediante negociaciones concurrentes. Con este diseño se espera sentar las bases y conseguir que, en un futuro no muy lejano, los servicios realizados por UAV puedan ser coordinados de manera eficiente teniendo en cuenta las características de dichos UAVs como por ejemplo la capacidad de carga, velocidad de vuelo y su autonomía.

Abstract

This final course work is about the execution of a mechanism of coordination based on the Unmanned Aerial Vehicle (UAV) - more commonly known as drones. This project is created due to the UAV's emerging popularity and the future necessity of coordination mechanisms for services.

Due to these reasons, this project tries to develop and design a concurrent coordination system through mechanisms of flexible auctions focused in the UAVs coordination. This project describes a surveillance service made by UAVs where thanks to a bidding system, it is possible to compete for a surveillance contract. The system defines an environment where multiple agents can interact: sellers and buyers. Buyers can be represented by a real person or a UAV, while the sellers are always represented by a UAV. These agents compete, according to their roles, for selling or securing a service.

Four UAVs have been used for this project execution as well as the testing and relevant study cases. These UAVs are the models Cargo from Parrot company and their intelligence has been programmed thanks to the connection capacity via Bluetooth Low Energy (BLE). Thanks to this, an environment where the UAVs are controlled through a Ground Control Station (GCS) is created and they interact among themselves and the buyer, while making offers and counteroffers.

Thanks to this project we are expecting to obtain a design of a multi-agent system for the coordination of UAVs by means of simultaneous negotiations. With this design it is expected to establish the bases and also achieve in a not too distant future, that the services performed by the UAV can be coordinated in an efficient manner whilst taking into account the UAVs characteristics , such as the charge capacity, speed flight and autonomy.

Agradecimientos

Este documento está dedicado a todas las personas que me han ayudado a lo largo de los años y en especial durante la elaboración de este proyecto.

Quiero darle las gracias, en especial, a toda mi familia y en especial a mis padres y a mi hermano por darme el apoyo que me han dado desde siempre y por *«obligarme»*, dependiendo del día, a continuar con mis tareas y a seguir dándolo todo aunque las cosas no fuesen del todo favorables.

También me gustaría dedicarle este documento a una persona muy especial para mí, que apareció de repente y me hizo una persona mucho más feliz otorgándome su tiempo, paciencia y cariño. Por esto quiero darle las gracias a Merche que espero que me sigas aguantando muchos años más. Próximo destino San Juan...

Gracias también a todos mis amigos por los ánimos ofrecidos y las ayudas dadas.

Otra persona a la que le tengo que dar las gracias y que no puede faltar es mi tutor David Vallejo. Muchas gracias por las horas echadas y la ayuda ofrecida.

David Frutos Talavera

A mis seres queridos

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Listado de acrónimos

TFG	Trabajo de Fin de Grado
SOA	Service-Oriented Architecture
SMA	Sistema multi-agente
GCS	Ground Control Station
UAV	Unmanned Aerial Vehicle
RUR	Rossum's Universal Robots
IA	Inteligencia Artificial
SMR	Sistemam Multi-Robot
AUV	Autonomous Underwater Vehicle
ROV	Remotely Operated Vehicle
RPV	Remotely Piloted Vehicle
ADAS	Advanced Driver-Assistance Systems
LARNYX	Long-range gun with Lynx engine
BTT	Basic Training Target
LORAN	Long Range Navigation
GPS	Global Positioning System
DASH	Drone Anti-Submarine Helicopter
BLOS	Beyon Line of Sight
DFCS	Digital Flight Control System
UAS	Unmanned Aerial System
OACI	Organización de Aviación Civil Internacional
RPA	Remotely-Piloted Aircraft
RPAS	Remotely-Piloted Aircraft System
RPS	Remote Pilot Station
UMA	Unmanned Aircraft
APV	Automatically Piloted Vehicle

UTA	Unmanned Tactical Aircraft
UCAV	Unmanned Combat Air Vehicle
ROA	Remotely Operated Aircraft
MUAV	Micro Unmanned Aerial Vehicle
VANT	Vehículo aereo no tripulado
MALE	Medium Altitude, Long Endurance
HALE	High Altitude, Long Endurance
MAVLink	Micro Air Vehicle Link
LGPL	Lesser General Public License
SITL	Software In The Loop
PC	Personal Computer
HW	Hardware
POSIX	Portable Operating System Interface UNIX
SDK	Software Development Kit
AOSE	Agent Oriented Software Engineering
FIPA	Foundation for Intelligent Physical Agents
FIPA-ACL	Agent Communication Language
JIAC	Java Intelligent Agent Compontentware
AAP	April Agent Platform
RAE	Real Academia Española
SO	Sistema Operativo
CPU	Central Processor Unit
API	Application Programming Interface
GDI	Graphics Device Interface
SFU	Windows Services for UNIX
CWI	Centrum Wiskunde & Informatica
OO	Orientación a Objetos
GIL	Global Interpreter Lock
FIFO	First In First Out
MPI	Interfaz de Paso de Mensajes
WWW	World Wide Web
TOD	Task Oriented Domains

WOD	Worth Oriented Domains
S	Strategies
PS	Percentage of Success Matrix
PO	Pay Off Matrix
EU	Expected Utility
RAD	Rapid Application Development
CASE	Computer Aided Software Engineering
XP	eXtreme Programming
DSDM	Dynamic Systems Development Method
SW	Software
BLE	Bluetooth Low Energy
LTS	Long Term Support
GPL	General Public License
WYSIWYG	What You See Is What You Get
IDE	Integrated Development Environment
HTTP	Hypertext Transfer Protocol
FTP	File Transfer Protocol
OWL	Ontology Web Language
XML	Extensible Markup Language
REST	Representational State Transfer
PIL	Python Imaging Library
JSON	JavaScript Object Notation
UUID	Universally Unique Identifier
BASH	Bourne again shell

Capítulo 1

Introducción

En los últimos años el uso de los Unmanned Aerial Vehicle (UAV) o comúnmente llamados **drones de uso civil** ha aumentado drásticamente. Antes de continuar con este capítulo definiremos un UAV o dron como un vehículo aéreo que no dispone de piloto y que suele ser controlado o de forma remota por un piloto o por un programa dotado de cierta inteligencia que es capaz de llevar a cabo misiones [OdM15] (véase § 3.1.3)

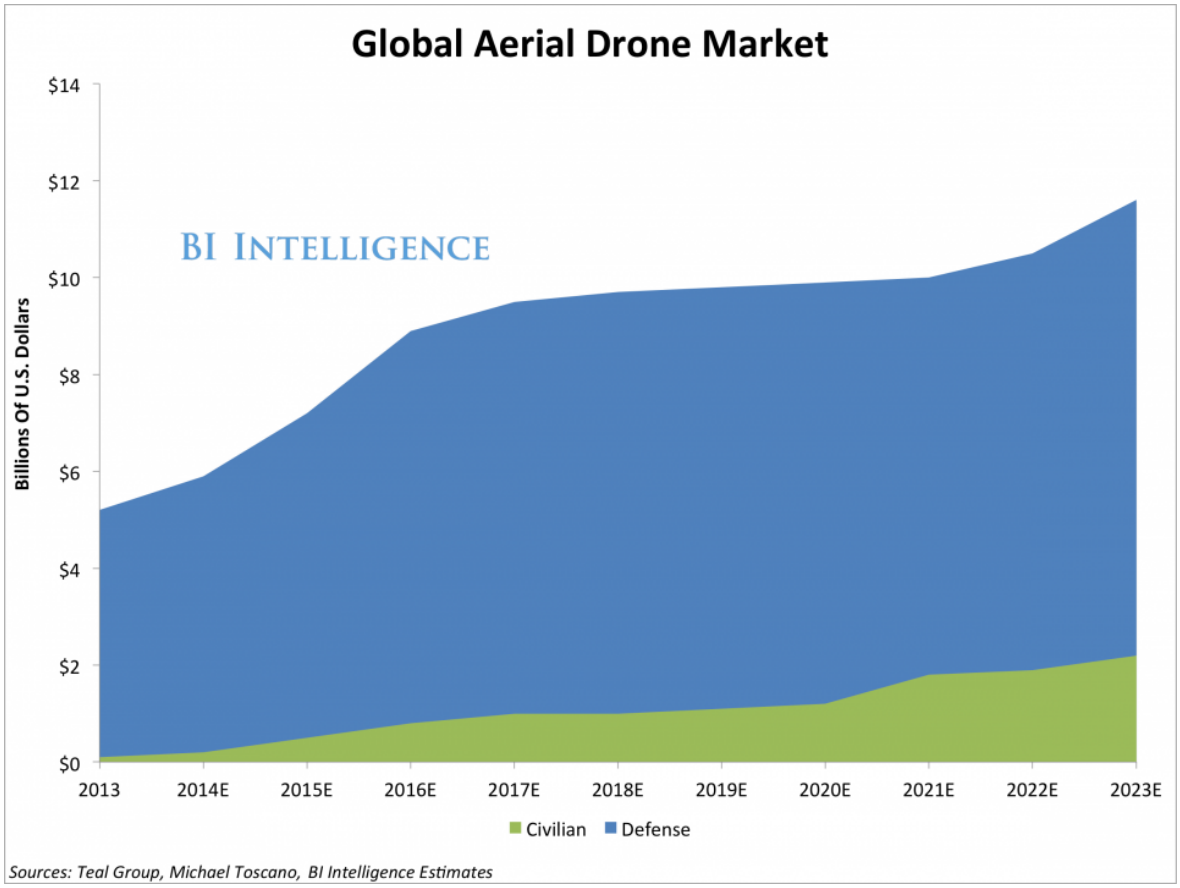

Figura 1.1: Estimación del mercado de UAV militares y civiles. `http://www.businessinsider.com/the-market-for-commercial-drones-2014-2`

Las capacidades de dichos UAVs ha ido incrementandose con el paso de los años desde simples aeronaves pre programadas hasta los UAVs actuales capaces de realizar un reconoci-

miento del terrero y un modelado posterior en 3D.

Existen decenas de utilidades que pueden ser desempeñadas por los UAVs civiles (véase § 3.1.3.3) y dado que su uso está creciendo (véase Figura 1.1) se espera que el numero de aplicaciones también aumente con el tiempo ya que es una tecnología emergente.

La mayoría de estas aplicaciones **requieren de una coordinación** y gestión de los recursos utilizados. Dado que los UAVs tienen una autonomía y capacidad es necesario poder gestionar qué tareas o servicios realizan y que dichas tareas siempre supongan el mayor beneficio para él propietario del servicio. Con el tiempo estas utilidades requerirán de una arquitectura que permita su coordinación para realizar distintas tareas o servicios.

Supongamos que un servicio o una tarea pueda ser realizada por un número indeterminado de UAV y que dichos servicios puedan ser contratados por otros UAVs o por clientes externos, en este punto, se requerira que los UAVs puedan coordinarse mediante un sistema descentralizado que permita a los UAVs comunicarse entre ellos directamente, además dicho sistema podra disponer de un sistema de negociaciones basado en subastas con posibilidad de cancelación de acuerdos bajo penalización. En otras palabras, se pretende realizar una prueba de concepto de un diseño de un sistema multi-agente para la coordinación de UAV mediante negociaciones concurrentes basadas en subastas flexibles.

1.1 Contexto

Para poner en contexto al lector de este TFG es necesario explicar una serie de conceptos antes de continuar, aunque serán detallados en el capitulo § 3.

Es necesario explicar que existen distintos tipos de UAV. Existen los que llevan un ordenador incorporado que los dota de cierta inteligencia y pueden maniobrar solos de forma inteligente; los que se pueden controlar desde una estación de tierra y su inteligencia reside en dicha estación, y, los que directamente son pilotados por un humano remotamente. Este TFG se centra en los UAVs que poseen cierta inteligencia.

Cabe destacar que la mayoría de las aplicaciones realizadas por UAV, **debido a la leyes**, son realizados por operarios que pilotan el UAV remotamente, pero aun así existen situaciones donde realizan servicios autónomos y son pilotados con cierta inteligencia sobre todo en tareas relacionadas con el transporte o con la vigilancia. Este proyecto está basado en una Service-Oriented Architecture (SOA) o Arquitectura Orientada a Servicios citehuhns2005service que, como tal, es un paradigma de arquitectura orientado a diseñar y desarrollar sistemas distribuidos. SOA define, a grandes rasgos, los siguientes términos: *Servicio*, *orquestación*, *sin estado*, *proveedor* y *consumidor*.

También es necesario explicar que un Sistema multi-agente (SMA) [Wei06] es un sistema compuesto por múltiples agentes inteligentes que interactúan entre ellos, siendo un agente una entidad inteligente que existe dentro de un contexto y que es capaz de comunicarse. Un

agente en un sistema multi-agente posee ciertas características: *Autonomía*, *visión local* y descentralización.

En los sistemas multi-robots suele usarse un enfoque de multi-agentes, siendo cada robot un agente del sistema. Los UAVs no distan mucho de un robot por lo que utilizar este enfoque para realizar este proyecto es de lo mas lógico.

Una vez se tiene a los UAVs ofreciendo servicios como agentes es necesario coordinarlos. Para esta coordinación de servicios existen dos enfoques, la **centralizada** y la **descentralizada**. En la centralizada una estación de tierra (agente coordinador) controla, centralmente, la coordinación de una serie de UAV que realizan una tarea o servicio. La descentralizada, la que será utilizada en este TFG, el UAV posee cierta independencia, y ya sea mediante su ordenador de a bordo o una estación de tierra, es capaz de coordinarse directamente con otros UAVs para realizar su servicio.

Cabe destacar que se ha optado por una coordinación descentralizada dado que existe una tendencia a aumentar cada vez más la inteligencia de los UAVs y poco a poco están siendo desarrollados con mayor inteligencia.

Cuando varios agentes compiten por el control de un recurso o para ofertar un servicio es necesario inculcarles la capacidad para negociar dicho recurso por eso los mecanismos de negociación son una de las posibles formas que se utilizan para guiar las interacciones entre los agentes. Estos mecanismos se utilizan para ver que sujeto o agente realiza el servicio de manera más optima para el cliente. Entre los existentes podemos encontrar diversos métodos: el regateo, la formación de coaliciones, la votación y las subastas [DAM01].

Este TFG se centrara en las subastas dado que, según el criterio del desarrollador, son las que más se asemejan a un sistema típico de venta de servicios, donde el cliente quiere que se le realice el trabajo de la manera más barata posible y el vendedor quiere ganar el mayor dinero posible. Las subastas pueden ser mejoradas, si se tienen en cuenta un detalle, no se están subastando bienes si no servicios, servicios realizados por un agente y que no tienen que ser realizados por completo. Si se tiene claro este concepto, es posible tanto para el vendedor como para el cliente seguir realizando subastas y pujas de forma concurrente, siempre que se añada la posibilidad de romper un acuerdo ya realizado, aun a costa de pagar una penalización. Esta posibilidad de ruptura dota al mecanismo de subasta de una flexibilidad adicional. [NJ05a] [NJ03] [PFJ98]

1.2 Motivación

Como dato interesante, las dos tendencias tecnológicas con mayor auge en el 2016, según la CNN [Bur16], son los UAVs y la realidad virtual.

Cada año aparecen nuevas aplicaciones donde el uso de los UAVs es necesario. En 2015, en España, había más de 300 operarios de UAV registrados incluyendo empresas, autónomos

y particulares [BP15] y en 2016 ya existen más de 1.500 operarios de UAV que se dedican a distintos trabajos, desde la revisión de reformas ilegales en hogares, pasando por control de plantaciones, incluso para medición de la polución en ciudades, hasta su uso en las fuerzas de seguridad y protección civil[1].

A nivel socio-económico el uso de los UAV ha generado nuevos mercados y millones de ingresos. El gobierno Español ha recaudado gracias al uso de los UAV la cantidad de 1.200 millones de euros en impagos. Una rápida visita a Google nos muestra que hay alrededor de 150.000 resultados con la palabra *Curso de drones*. A inicios de 2016 la Comisión Europea estimaba que la industria de drones movería 15.000 millones de euros y generaría unos 250.000 empleos en todo el mundo. Citando una frase de Jaime Guillot, director ejecutivo de Drone Spain: *«El sector se empieza a profesionalizar en 2014 con la entrada de la legislación. En ese momento éramos unas 20 empresas y ahora mismo hay más de 1.000.»*.

A nivel mundial se prevé que los ingresos totales de las compañías relacionadas con los UAVs alcance los 12.000 millones de dólares en el año 2025 (véase Figura 1.2).

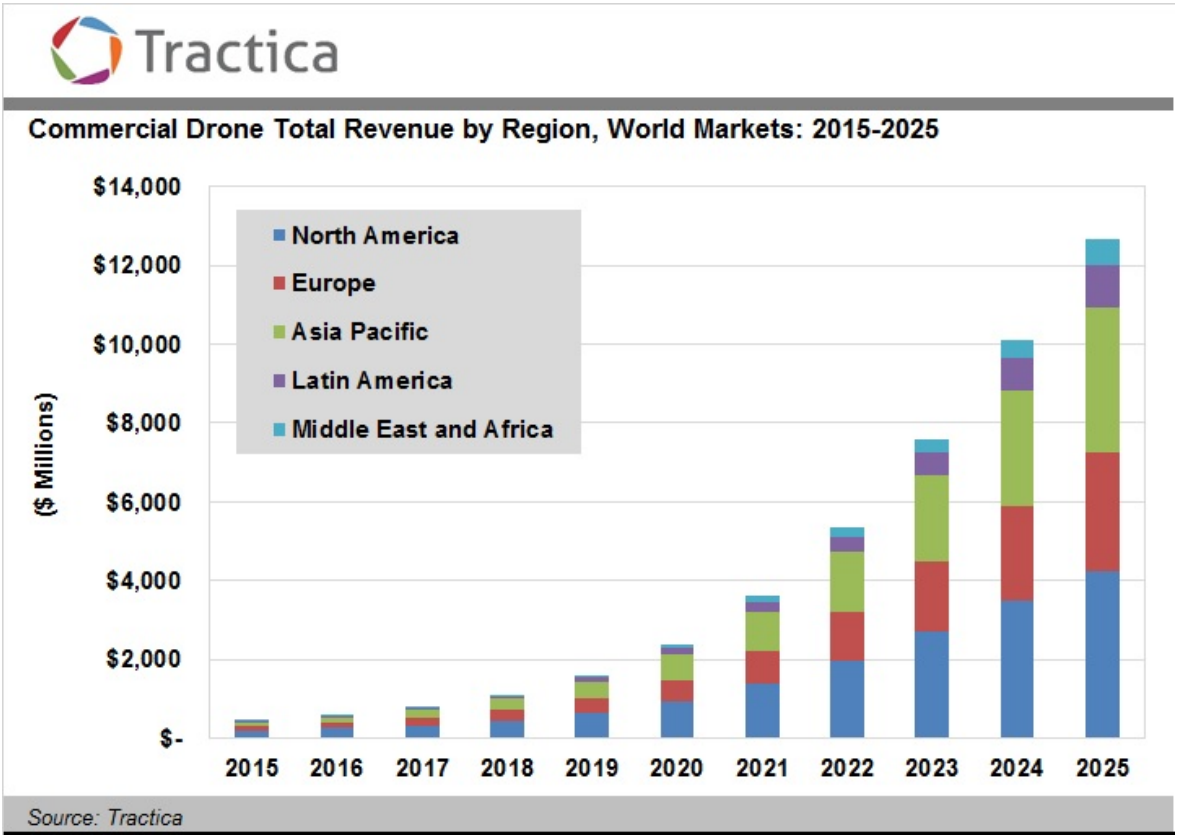

Figura 1.2: Ganancias estimadas en empresas relacionadas con los UAVs.https://www.tractica.com

No hay mucho más que decir sobre el impacto socio-económico del uso de los UAVs, las cifras hablan por sí solas solo que a través de estos datos se prevé que en un futuro sea

[1] http://dronewiki.net/los-trabajos-con-drones-que-ya-se-estan-haciendo-en-espana/

necesario diseñar una serie de mecanismos o estructuras de coordinación enfocadas a los UAVs.

1.3 Propuesta planteada

Una vez puesto en contexto, en este TFG se pretende **coordinar dispositivos UAVs de manera eficiente en base a su estado interno** en situaciones donde un cliente o comprador desee adquirir un servicio ofertado por dichos UAVs. En este proyecto se ha diseñado un sistema multi-agente capaz de coordinar UAVs mediante negociaciones concurrentes basadas en **subastas flexibles** para así, poder ofrecer servicios de una manera más eficiente tanto para el cliente como para el vendedor ya que serian los propios UAVs los que, a partir de sus **atributos variables** (velocidad, autonomía, capacidad de carga, etc...) y fijos (valor de reserva, penalización por cancelación de contrato, tiempo de subaste, etc...), realizarían el proceso de subasta punto a punto con el cliente.

Con el diseño de este sistema se busca:

- Coordinar gran cantidad de dispositivos UAVs.
- Minimizar la interacción entre humano y dispositivo.
- Optimizar los recursos del vendedor.

El sistema será adaptable según las necesidades de los clientes gracias a la compatibilidad que se le ha otorgado al sistema para aceptar ontologías definidas siguiendo la estructura OWL. Esta ontología define el servicio que se ofrece además de como interactúan los distintos actores y dispositivos gracias a que la ontología define una serie de clases (objetos, actores, conceptos, etc...) y relaciones entre dichas clases.

Estos dispositivos serán programados mediante **Node.js** y **Python** dado que el **SDK de *Parrot*** no tiene, a fecha actual, implementada la conexión a sus dispositivos BLE como son los UAVs modelo *Cargo* que serán utilizados en el desarrollo de este proyecto.

1.4 Estructura del documento

- **Capítulo 1: Introducción**

  Se realiza una introducción al Trabajo de Fin de Grado (TFG)

- **Capítulo 2: Objetivos**

  Se definen los objetivos propuestos y que se esperan realizar durante la ejecución de este proyecto.

- **Capítulo 3: Antecedentes**

  En este capítulo se realiza un breve repaso al estado del arte pasado y actual de los UAVs, posteriormente se entra en detalles relacionados con la programación con-

currente para terminar con una breve explicación de los mecanismos de coordinación centrándose en el documento utilizado como base para este proyecto.

- **Capítulo 4: Método de trabajo**

  En este capítulo se detallan las metodologías utilizadas existentes y la metodología escogida para la realización de este proyecto así como los medios HW y SW que se han utilizado.

- **Capítulo 5: Arquitectura**

  En este capítulo se detalla la arquitectura y los modulos desarrollados durante la realización del sistema de subastas.

- **Capítulo 6: Evolución, resultados y costes**

  En este capítulo se detallan las fases del proyecto, las pruebas evolutivas y el caso de estudio realizado detallando problemas a solucionar y resultados obtenidos. También se detalla el coste total del proyecto.

- **Capítulo 7: Conclusiones y trabajo futuro**

  En este capítulo se detallan las conclusiones de utilizar este sistema de coordinación orientado a UAV y los trabajos futuros para mejorarlos.

Capítulo 2

Objetivos

En este capítulo, describiremos detalladamente los objetivos propuestos para llevar a cabo este TFG. Partiremos de un objetivo global o general que será detallado minuciosamente y definiremos una serie de subobjetivos más específicos.

2.1 Objetivos generales

Nos surgen varias preguntas, ¿pueden los UAV ofrecer servicios?. Sí, según la definición dada en la seccion UAV (véase § 3.1.3), sí son capaces. ¿Puede un UAV ser un agente?. Posee las características de un agente por lo que si puede. ¿Dicho servicio puede ser orquestado mediante un *sistema multiagente descentralizado*?. Sí, no son excluyentes, un Service-Oriented Architecture (SOA) no específica cómo tiene que ser realizado un servicio y un Sistema multi-agente (SMA) no específica cómo tiene que ser contratado, se podría realizar una orquestación conjunta entre SOA y SMA.

Una vez respondidas estas preguntas podemos pasar a especificar los objetivos generales de este TFG.

Este proyecto, tiene como objetivo general, el diseño y desarrollo de un *esquema de coordinación descentralizada* para UAV mediante la creación de un sistema multi-agente que permita coordinador dichos UAV mediante **negociaciones concurrentes**. Una vez finalizado dicho objetivo se pretende obtener un esquema que permita:

- La comunicación descentralizada, punto a punto, entre dos *agentes* del sistema.
- Coordine los agentes mediante *hilos de negociación concurrentes*. Mantener *negociaciones concurrentes* permite al *agente* poder negociar con distintos *agentes* y otorgar *servicios* simultaneamente.
- A un *agente* llegar a un acuerdo para adquirir el servicio que ofrezca otro *agente* a un coste optimo.
- A los *agentes* cancelar el acuerdo en cualquier momento, bajo una penalización.
- Disminuir el tiempo de negociación optimizando así el uso de los UAV.
- Realización de tareas conjuntas entre distintos UAV.

El estado actual de la tecnología permite controlar el UAV desde tierra o mediante un ordenador de abordo, aun así, se pretende crear un sistema que pueda ser añadido a la inteligencia del propio UAV, ya sea controlado desde una Ground Control Station (GCS) o desde el propio ordenador de abordo. En nuestro caso, y dado que los UAVs que utilizaremos no disponen de ordenador de abordo, se separa la lógica de coordinación y comercio de la lógica de vuelo automático, permitiendo así, cambiar de UAV de forma sencilla.

2.2 Objectivos específicos

Este objetivo general, puede ser dividido en objetivos específicos que compondrán las bases para cumplimentar el objetivo general:

- **Flexibilidad:** se busca un esquema que permita adaptarse a las necesidades del escenario. La flexibilidad será obtenida de distintas maneras: la **cancelación del contrato** a partir de un sistema de pujas flexible que permita a los *clientes* y *proveedores* cancelar un contrato parcial del servicio. Este objetivo permitirá a los *agentes* realizar *negociaciones* de forma *concurrente*, incluso realizar un servicio mientras realizan negociaciones con otros *agentes*. El **cambio de roles** permitirá a los agentes realizar distintos tipos de roles, *clientes* y *proveedores* , según el momento y la necesidad. El uso de una **ontología del servicio** , a partir de un diseño general, permitirá al sistema adaptarse, de forma fácil y sencilla, a las necesidades del cliente. La posibilidad de agregar nuevas *estrategias* dotara al sistema de cierta flexibilidad.
- **Escalabilidad:** la incorporación de *hilos de negociación* permitirá al esquema gestionar un número indeterminado de *agentes*. Además, Mediante un sistema de subscripción y listado de servicios se desvinculara al sistema del número de agentes. El diseño de una ontología permitirá al sistema adaptarse a la estructura definida y poder servir para distintos servicios, como por ejemplo: servicios de vigilancia mediante UAV o servicios de mensajería mediante UAV.
- **Descentralización:** gracias a un sistema de comunicaciones que permita conexiones punto a punto se permitirá al sistema descentralizarse y a los UAV coordinarse entre ellos.
- **Análisis forense de datos:** el esquema generara una serie de datos históricos que detallara las pujas aceptadas y canceladas por parte de los clientes. Esto permitirá al cliente analizar si de verdad se están seleccionando los mejores contratos y optimizar la configuración de sus *agentes* además de usarlo para adquirir conocimiento sobre los *proveedores*.
- **Estándares abiertos:** se pretende que el sistema sea multiplataforma y capaz de funcionar en distintos dispositivos por eso se pretende utilizar estándares abiertos durante la elaboración de este proyecto. Este punto está muy relacionado con la versatilidad

que se busca en el sistema y la capacidad que tienen los UAV de ser controlados mediante un GCS o mediante un ordenador de abordo.

Capítulo 3

Antecedentes

En este capítulo se definirá la base teórica de las principales tecnologías utilizadas en el desarrollo de un sistema multi-agente para la coordinación de UAVs mediante negociaciones concurrentes. Se repasaran conocimientos sobre sistemas multi-robots, programación concurrente y toma de decisiones, conocimientos básicos para establecer el marco conceptual de este Trabajo de Fin de Grado (TFG).

3.1 Sistema multi-robot

3.1.1 Introducción

El termino **robot** fue acuñado en 1920 por el escritor Karel Capek en su novela Rossum's Universal Robots (RUR) [CS23], significaba siervo, fuerza de trabajo y fue ya en 1948, cuando el neurólogo Grey Walter construyo las denominadas tortugas de Bristol, el primer robot móvil.

Ya desde su comienzo, uno de los grandes propósitos de la Inteligencia Artificial (IA) y la Robótica ha sido simular el comportamiento de los seres humanos o incluso de animales. Estos, cuando necesitan realizar tareas complejas que son impensables o muy tediosas para un solo individuo, optan por unirse a otros individuos, coordinarse y realizar la tarea o subtareas de forma conjunta. Esta solución es la que se ha tomado como base para la lógica de un **Sistemam Multi-Robot (SMR)**.

En general un SMR está caracterizado como un sistema formado por dos o más robots que comparten un entorno común [Par94]. Cabe destacar que dichos robots pueden ir desde simples robots con sensores los cuales obtienen e informan de sus datos hasta complejos robots que interactúan con el éntorno de forma compleja. También cabe déstacar que no todos los individuos del conjunto tienen que desempeñar la misma función.

Otra de las características de un SMR es que los robots del sistema deben trabajar conjuntamente para la realización de diversas tareas. Para llevar a cabo estas tareas es necesario que los robots se coordinen entre sí o que como mínimo no se interfieran mutuamente y demuestren cierto grado de cooperación [Ark92], en este caso, entendemos que existe una coordinación pasiva, de forma que los robots dél sistema no necesitan comunicarse para realizar una tarea conjunta dado que sus acciones no se ven interferidas entre sí. La coordinación

entre los robots del sistema se realiza mediante un **sistema de comunicación**. Esta puede ser: directa o intencional, donde se define un destinatario, o indirecta o no intencional, donde se envía el mensaje sin un receptor específico. Además, existen tres tipos de mecanismos de comunicación: la **no comunicación**, la **comunicación de estados** y la **comunicación de objetivos**. Estos influyen en la complejidad de la coordinación, siendo la no comunicación el mecanismo menos complejo, dado que los robots no se pasan información; la comunicación de estados tiene una complejidad media dado que los robots conocen el estado actual de los robots del conjunto; y la comunicación de objetivos es la que involucra mayor transmisión y recepción de información por lo que se pueden realizar tareas más complejas.

3.1.2 Robots terrestres

Existen multitud de modelos de robots y, por consiguiente, varios tipos de clasificaciones. Los robots terrestres están clasificados como **robots móviles**. Estos robots tienen la capacidad de moverse por el entorno pudiéndose diferenciar varios tipos dependiendo del entorno en el que se muevan y los mecanismos de desplazamiento o locomoción que utilicen.

Cabe destacar que todos los robots móviles, en mayor o menor medida, necesitan saber su posición, y los distintos métodos de transporte o locomoción añaden mayor o menor grado de incertidumbre. Por eso, es muy importante seleccionar el mejor **mecanismo de transporte**.

Existen múltiples soluciones (véase Figura 3.1), entre las cuales encontramos[1]:

- **Robots con ruedas:** son los más comunes dentro de los robots móviles. Una de las principales causas es su alta eficiencia respecto a la movilidad, aunque están restringidos a entornos con superficies planas o poco irregulares. Dentro de este tipo se pueden diferenciar según el chasis o diseño de ruedas que se utilice: diferencial, sincronizado, triciclo y coche. Dependiendo del tipo de diseño de ruedas podemos obtener una mayor estabilidad, eficiencia, coste y complejidad algorítmica. Por ejemplo, el diseño de triciclo y coche es bastante similar respecto a estabilidad y eficiencia pero el diseño de tipo triciclo es ligeramente más barato de construir.
- **Robots orugas o con cadena:** estos robots son más eficientes que los robots de ruedas en entornos con bastantes irregularidades dado que mantienen más superficie en contacto, pero esto los hace más ineficientes a nivel energético. A grandes rasgos entran dentro de los robots de ruedas con un diseño sincronizado.
- **Robots con patas o zoomórficos:** estos robots son un claro ejemplo de la búsqueda de imitar a la naturaleza. Disponen de una capacidad excepcional para adaptarse al terreno, siendo perfectos para terrenos sumamente irregulares pero con poca eficiencia, además requieren un número elevado de motores por lo que aumenta el gasto de construcción y la complejidad algorítmica.

[1] `http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/clasificacion-de-robots`

Figura 3.1: Diversos tipos de robots. De izquierda a derecha y de arriba abajo: zoomórfico, humanoide, submarino, manipulador movil, aéreo, oruga. `http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/clasificacion-de-robots/`

- **Robots humanoides o androides:** como pasa con los robots zoomórficos, este también es un claro ejemplo de la búsqueda de imitación de la naturaleza. Estos robots imitan la morfología humana. Como pasa también con los zoomórficos, los humanoides, no son eficientes respecto a la movilidad aunque actualmente podemos encontrar diversos robots con capacidades de aprendizaje lo que les permite mejorar su autonomía y control.
- **Robots manipuladores móviles:** otro tipo de robots, que no están clasificados como móviles o terrestres son los clasificados como **brazos manipuladores**, estos robots están fijos en un punto y se encargan de realizar tareas repetitivas y precisas. Un robot manipulador móvil es un robot con una base móvil que dispone de un brazo manipulador en su torso.
- **Robots transformables:** estos robots tienen la capacidad de cambiar su apariencia y desplazarse de distinta forma según la necesidad. Un claro ejemplo seria un robot anfibio capaz de ir por agua y tierra o incluso aire.
- **Robots marinos o submarinos:** estos robots tienen la capacidad de moverse en un medio acuático. Dado que el medio es muy distinto al utilizado por la mayoría de los robots, es necesario adaptar o utilizar otro tipo de sensores. Los denominados **Autonomous Underwater Vehicle (AUV)** son robots no tripulados con navegación autónoma, y los **Remotely Operated Vehicle (ROV)** están tripulados desde un barco y suelen tener dependencia energética. No todos los robots marinos trabajan a grandes profundidades, existen robots **planeadores** o **gliders** que planean entre la superficie marina y debajo del mar para aumentar su eficiencia energética.
- **Robots aéreos:** al igual que los robots marinos, también existen robots que se mueven en un entorno aéreo. Los denominados **Unmanned Aerial Vehicle (UAV)** son robots que pueden realizar vuelos automáticos sin necesidad de pilotaje, aunque también, como pasaba en los marinos, existe otro tipo llamado **Remotely Piloted Vehicle (RPV)** pilotado a distancia por un controlador. En la seccion UAV (véase § 3.1.3) se puede encontrar más información de dichos robots móviles.

En los dos últimos tipos de robots, se han utilizado términos como *unmanned* o *autonomous* y *remotely operated* o *remotely piloted*, estos términos se utilizan para definir el grado de autonomía que posee el robot.

Definimos cómo **autonomía de un dispositivo** la capacidad que tiene de funcionar gracias a una fuente de alimentación, pero este término, en relación a los robots, cambia un poco. Entendemos cómo la **autonomía de un robot** a la capacidad que tiene de auto-gobernarse, esto implica también el grado de capacidad que tiene el robot de enfrentarse a situaciones en un entorno cambiante.

Se definen tres tipos de grados de autonomía para un robot:

- **Teleoperados:** son los anteriormente llamados *Remotely Piloted* o *Remotely Operated*. Son robots controlados mediante un cable físico o un sistema inalámbrico por un piloto o un operario. Son robots que tienen un grado de autonomía mínimo, casi nulo, que no son capaces de realizar acciones por sí mismos. Los controles utilizados para manejar dichos robots suelen ir desde el clásico joystick de ejes hasta controladores hápticos donde el operador recibe información del entorno.
- **Semiautomáticos:** si anteriormente decíamos que los robots teleoperados no eran capaces de realizar acciones sin intervención de un operador, los robots semiautomáticos, aunque controlados a distancia, si son capaces de corregir la acción del operador. Estos robots poseen un grado de autonomía parcial, pueden ir desde robots que solo corrigen la acción del operario en relación a sus sensores, como los sistemas de conducción asistida avanzada **Advanced Driver-Assistance Systems (ADAS)** de los vehículos cotidianos, hasta robots casi autónomos, que solo requieren de la intervención de un operario en ciertos momentos.
- **Automáticos:** los robots automáticos son los que poseen mayor grado de autonomía. Su grado de autonomía depende del conocimiento del entorno que tenga el robot dado que estos lo único que hacen es entender las variables de su entorno y tomar decisiones en relación a ellas y llevar a cabo acciones autónomas, en ningún momento son controlados por operarios.

Como dato final, otro tipo de clasificación de los robots es según su utilidad y origen, clasificándose como:

- Industriales de manipulación.
- Servicio
- Investigación
- Militares
- Médicos
- Nano robots
- Educacionales
- Imprimibles
- Juguetes robóticos

A lo largo de todo el mundo existen empresas dedicadas a los robots terrestres, en el cuadro 3.1 se puede ver una muestra de dichas empresas relacionadas con la inspección, exploración, rescate y defensa..

Empresa	Pais	Descripción
Aquiles Robotics Systems	España	Robots de seguridad Aquiles I y Aquiles Protector. Dispone de brazo que le permite realizar operaciones sobre objetos.
Automax Robots	Japón	Robots manipuladores para trabajos peligrosos (Cables de alta tensión, sustancias peligrosas).
Bluebotics	Suiza	Plataforma móvil Shrimp III para terrenos abruptos (unión de ruedas y patas).
CRASAR	EEUU	Centro de la U. del Sur de Florida dedicado a la robotica para búsqueda y rescate.
Cybernetix	Francia	Robots para inspección y desmantelamiento en edificios (centrales nucleares) e inspección en exteriores.
Hydro-Quebec Research Institute	Canadá	Robots LineScout y LineROVer para inspección y trabajos en líneas de alta tensión.
iRobot	EEUU	Diversos robots terrestres de inspección y militares, como Packbot, SUGV, 110 Fisrtlook y el 710 Warrior.
Proytecsa	España	Robots serie AUNAV con orugas y dos brazos para aplicaciones de seguridad como la desactivación de explosivos.
QuinetiQ	EEUU	Robots TALON y Dragon Runner.
ReconRobotics	EEUU	Robots de la serie Recon Scout y robots Throwbot para exploración e inspección orientados principalmente a defensa y misiones militares.
Robotnik Automation SLL	España	Robots RESCUER, GUARDIAN y Summit.
Robowatch Technologies	Alemania	Robots autónomos o teleoperados como CHRYSOR, MOSRO, OFRO y ASSENDRO.
Tmsuk	Japón	T-52 Enryu, robot de rescate teleoperado.
Yujin Robot	Corea	Robots móvil con orugas ROBHAZ para aplicaciones civiles y militares.

Cuadro 3.1: Empresas de robótica relacionadas con la inspección, exploración, rescate y defensa. `http://www.roboticadeservicios.com/robots_exploradores.html`

3.1.3 UAV

3.1.3.1. Introducción

Unmanned Aerial Vehicle, vehículo aéreo no tripulado o comúnmente dron, un término que actualmente está en auge y que es bastante conocido y, aunque fue acuñado en los años 90, tiene más de un siglo de historia.

Como ya se ha mencionado con anterioridad, está catalogado como robot terrestre, pero en este apartado se entrara en más detalle.

Podemos dividir el nombre en dos partes, vehículo aéreo, esto incluye todo dispositivo capaz de mantenerse flotando en el aire mediante métodos propios; y no tripulado, tiene que tener la capacidad de ser pilotado remotamente o de poder volar autónomamente, no necesariamente tiene que disponer de un sistema muy complejo.

Origen del vehículo aéreo no tripulado. Con el término dron explicado, nos vienen a la cabeza complejos drones capaces de realizar misiones de espionaje, el conocido como *Predator* [2], pero nos sorprendería saber que, técnicamente, sus inicios se remontan al siglo XIX, donde se empezaron a desarrollar las bases de lo que actualmente son los aviones tripulados[3] , gente como George Cayley[4], John Stringfellow[5], Félix du Temple[6], William Samuel Henson[7] y otros pioneros de la aviación, crearon inicialmente, y para su propia seguridad, lo que se podría definir como vehículos aéreos no tripulados, precursores de sus primeros modelos tripulados.

A lo largo del planeta fueron surgiendo inventores que siguieron la misma progresión lógica, planeadores no tripulados, tripulados, aviones no tripulados y tripulados como podemos ver en el cuadro 3.2.

País	Planeador no tripulado	Planeador tripulado	Avión no tripulado	Avión tripulado
Inglaterra	Caeyly, 1809	Caeyly, 1849	Cody, 1907	Cody, 1908
Francia		Ferber, 1901	Du Temple, 1857	Santos-Dumont, 1906
Alemania		Lilienthal, 1891		
Japón		LePrieur/Aibara, 1909	Ninomiya, 1891	Nagahara, 1911
Rusia				Rossinsky, 1910
Estados Unidos		Chanute, 1896	Langley, 1896	Hnos. Wright, 1903

Cuadro 3.2: Primeros vuelos sostenidos conocidos en diversas naciones

Desarrollo militar y Nikolas Tesla. ¿Pero cómo se pasa de aviones de tela y madera a

[2] https://es.wikipedia.org/wiki/General_Atomics_MQ\discretionary{-}{}{}1_Predator

[3] http://drones.uv.es/origen-y-desarrollo-de-los-drones/

[4] https://es.wikipedia.org/wiki/George_Cayley

[5] https://fr.wikipedia.org/wiki/John_Stringfellow

[6] https://es.wikipedia.org/wiki/Félix_du_Temple

[7] https://en.wikipedia.org/wiki/William_Samuel_Henson

lo que hoy conocemos como drones? Esta pregunta tiene muy fácil respuesta: Desarrollo militar y un gran inventor, Nikolas Tesla[8].

La primera aeronave no tripulada utilizada para la guerra data de Julio de 1849[9] , cuando el ejército austriaco lanzo unos 200 globos aerostáticos, cargados con bombas, sobre la ciudad italiana de Venecia. Posteriormente, unos veinte años después, durante la Guerra Civil de EE.UU., el ejercito Confederado y el de la Unión utilizaron globos para misiones de reconocimiento. Samuel P. Langley[10] desarrollo varios aviones no tripulados en 1896 que fueron capaces de volar sobre el río Potomac y en 1898, en la Guerra Hispano-Americana, se tiene constancia de que, el ejército de los EE.UU., utilizó una cometa, equipada con una cámara de fotos, para tomar lo que serian las primeras **fotografías de reconocimiento aéreo**.

La evolución del dron a nivel militar pasó por varias fases:

- En la Primera Guerra Mundial se utilizaron como método para obtención de **fotografías aéreas en zonas de guerra**, vigilancia y espionaje.
- Posteriormente se les utilizo como **blanco de prácticas** para las fuerzas militares. Fue la terminología anglosajona la que acuñó el nombre por el cual se conocen comúnmente hoy: *Drones*.
- Después de la Primera Guerra Mundial y durante la Segunda Guerra Mundial, el dron se enfocó más hacia una **bomba volante de gran alcance**.
- Durante el periodo de la Guerra Fría se utilizó como sistema de espionaje y captura de datos de inteligencia.
- En la época reciente, el dron, se utiliza como un **arma de vigilancia**, dado que puede realizar misiones de reconocimiento y acabar con enemigos si se requiere. El *Predator* o *depredador* es el principal punto de referencia de los drones, de uso militar, actuales.

Toda esta evolución no se habría dado de no ser por el gran inventor Nikolas Tesla. Fue en 1898, en un estanque del Madison Square Garden, cuando Nikolas Tesla presento su *Telautomaton* (véase Figura 3.2): un barco, que poseía la capacidad de moverse en varias direcciones gracias a un **control remoto** que utilizaba distintas frecuencias de radio para identificar las acciones.

Curiosamente, Tesla, fue reacio a la invención del avión y lo catalogó de fracaso ante el uso de los dirigibles.

La capacidad de manejar un dispositivo a distancia llamó la atención de Archibald M. Low[11], que fue llamado el padre de los sistemas de radiocontrol debido a su trabajo pionero

[8] https://es.wikipedia.org/wiki/Nikola_Tesla

[9] http://eldrone.es/historia-de-los-drones/

[10] https://es.wikipedia.org/wiki/Samuel_Pierpont_Langley

[11] https://en.wikipedia.org/wiki/Archibald_Low

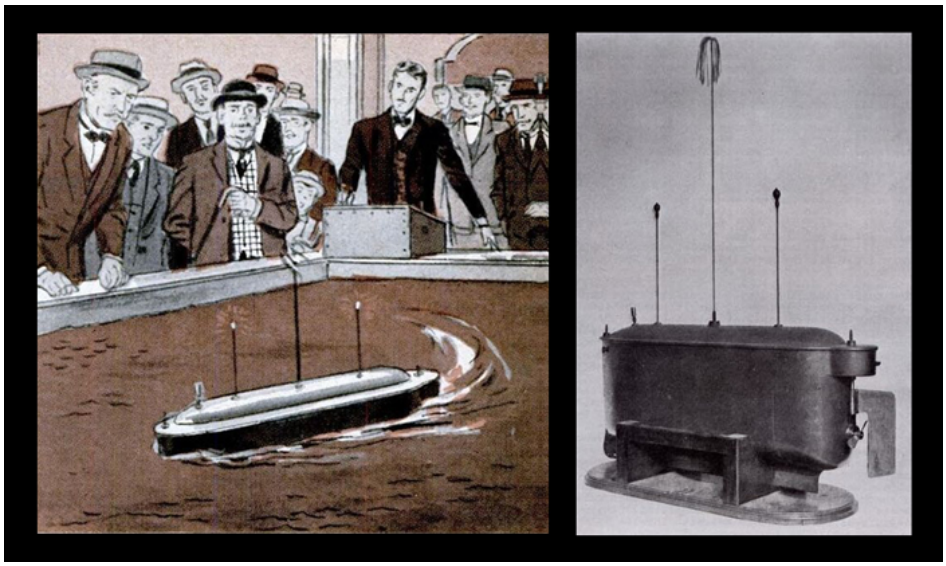

Figura 3.2: Presentación del Telautomaton de Tesla. https://sites.google.com/site/vidaehistoriadenikolatesla/teleautomata

en el sistema de guiado de cohetes, aviones y torpedos. Low fue el supervisor de un proyecto que involucraba aviones dirigidos por control remoto y explosivos.

Por otro lado, otro inventor, Elmer Ambrose Sperry[12] desarrolló el primer **giroscopio**, que aunque algo tosco y grande, fue un gran avance para el mundo de la aviación. Sperry también desarrollo una serie de aeronaves no tripuladas capaces de lanzar bombas, que según el New York Times de 1926 eran guiadas con gran precisión, dicho proyecto fue finalizado al terminar la guerra en 1918. Continuando con las bombas dirigidas, Sperry fabrico la *Hewitt Sperry*, capaz de volar 50 millas con una carga de 300 libras, dicho *avión* tenia incorporado una versión mejorada del giroscopio inventado por Sperry. Como se ha comentado anteriormente, este periodo de entreguerras se centró más en la creación de bombas dirigidas que en aviones propiamente dicho, aunque cabe destacar el proyecto **Aerial Target**[13], creado por los británicos, que demostró que se podía utilizar un sistema de control remoto para dirigir una aeronave de forma eficiente.

Entreguerras y principios de la Segunda Guerra Mundial. En los años 20[14] se desarrollaron buques capaces de ser controlados remotamente, también se desarrollo el **Long-range gun with Lynx engine (LARNYX)**[15], que realizo su primer vuelo en 1927. Ya en los años 30 se incremento el interés militar en el desarrollo de vehículos no tripulados.

Al final de la Segunda Guerra Mundial Gran Bretaña dejo de lado los misiles crucero y se centro en el desarrollo de blancos aéreos con control remoto y desarrollo el *Queen bee*

[12] https://es.wikipedia.org/wiki/Elmer_Ambrose_Sperry
[13] http://flyingmachines.ru/Site2/Crafts/Craft29148.htm
[14] https://sites.google.com/site/uavuni/1910-s
[15] https://en.wikipedia.org/wiki/RAE_Larynx

que fue utilizado para el entrenamiento de las fuerzas de artillería. Mientras tanto el ejército de EE.UU. desarrollo el RP4, creado por Radioplane Company[16], también utilizado para el entrenamiento de las fuerzas armadas.

Postguerra. Durante la década de los 50, en la postguerra, la compañía Radioplane, desarrollo los *Falconer* o *Shelduck* que se siguieron fabricando hasta 1980 y que recibieron la designación Basic Training Target (BTT) y que adoptaron sistemas de radio-control cada vez más eficientes. También en EE.UU. se desarrollaron los *Firebee* (véase Figura 3.3) creados por la compañía aeronáutica Ryan.

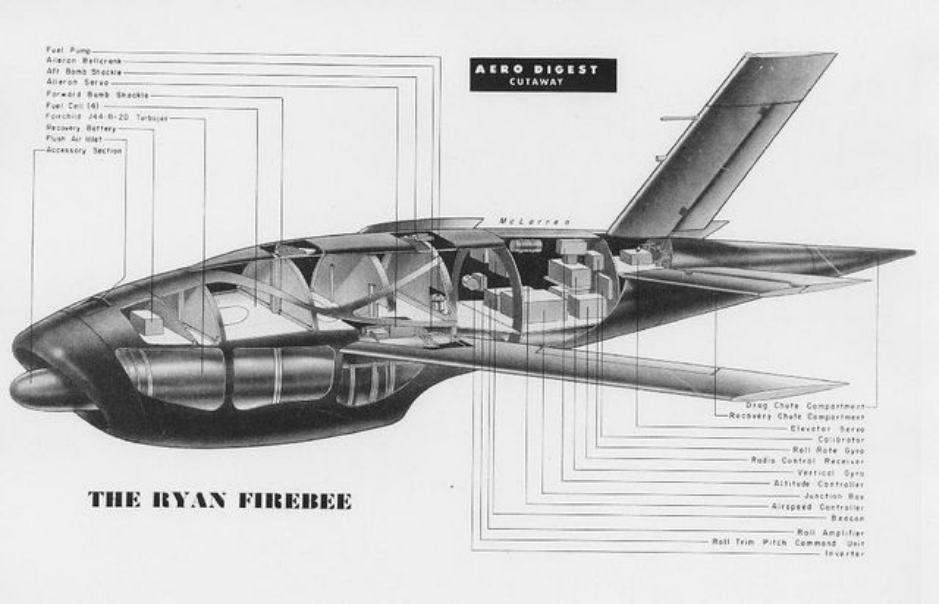

Figura 3.3: El Ryan Firebee. http://eldrone.es/historia-de-los-drones/

En esa fecha también se desarrollaron los sistemas *Croswbows*, sistemas enfocados a interferir en los radares enemigos y crear falsos positivos Estos sistemas, aunque también fueron a radio-control, fueron muy innovadores dado que incluyeron un sistema de video para un control más eficiente del piloto.

La Guerra de Vietnam y la Guerra Fría. La Guerra de Vietnam impulsó el desarrollo de lo que hoy se conoce como los drones actuales. La necesidad de conocer el terreno obligo a los EE.UU a desarrollar el mayor sistema de vigilancia con vehículos no tripulados. Los *Lightning Bugs* se utilizaron sobre Vietnam del Norte a partir de 1968. Entre 1964 y 1975, unos 1.000 drones realizaron más de 34.000 misiones de vigilancia, costando más de 250 millones de dólares al año. Fue en 1972 cuando se les incorporó la tecnología **Long Range Navigation (LORAN)** que aportó un sistema de ayuda a la navegación mejorando drásticamente la capacidad de pilotaje de dichas aeronaves. El sistema era similar a un Global Positioning System (GPS)[17] actual.

[16] https://en.wikipedia.org/wiki/Radioplane_Company

[17] https://es.wikipedia.org/wiki/Sistema_de_posicionamiento_global#Historia

Durante la Guerra Fría los Firebee creados en los 50 fueron modificados para llevar cámaras y realizar misiones de reconocimiento. Dichas aeronaves operaban a grandes altitudes pilotadas desde tierra o a baja altura pilotadas desde aeronaves tripuladas. En 1962, la empresa Ryan, creadora de los Firebees obtuvo fondos de las Fuerzas Aéreas para desarrollar el proyecto *Gran Safari*, siendo catalogado como el primer avión no tripulado de vigilancia. Los Firebees sufrieron diversos cambios pasando por los modelos *Ryan 147*, *AQM-34* y los ya conocidos *Lightning Bugs* o *Luciernagas*. Estos drones, lanzados desde aviones Hercules, volaban en rutas pre-programadas o eran controlados por operadores de radio a bordo, una vez finalizada la misión, desplegaban sus paracaídas para ser recogidos por helicópteros.

El helicóptero Drone Anti-Submarine Helicopter (DASH) fue el **primer UAV de los EE.UU.** creado para la guerra. Fue un diseño desde cero y no basado en los drones utilizados como blanco, fue diseñado para llevar diversas cargas explosivas y no incorporaba un autopiloto ni sensores, por lo que, como tal, fue un retraso dado que solo podía ser radio-controlado, sin embargo fue el primero en utilizar como base una nave de ala giratoria.

Década de los 70. En esta época, las Fuerzas Aéreas de EE.UU se centraron en aumentar el alcance y los métodos de vigilancia de los vehículos no teledirigidos, desarrollaron un programa que fue llevado a cabo por las compañías Boing y Ryan en el cual se crearon aviones capaces de volar de forma remota unas 24 horas. La versión Mk2 del *Chukar* incorporo, en los años 70, a su sistema, un autopiloto avanzado para realizar operaciones a grandes distancias o **Beyon Line of Sight (BLOS)**.

Años 1980 - 1990. En los 80 se creó la serie Canadair CL, su desarrollo comenzó en los años 60 y durante 20 años su diseño fue mejorando hasta entrar en servicio en los 90.

El modelo original de CL-89, era un dron orientado a la vigilancia, disponía de dos cámaras, una cámara normal y otra infrarroja, que se utilizaban de día y de noche dependiendo de las condiciones de luz. Este modelo era lanzado con un cohete desde una rampa y se recuperaba una vez finalizada la misión gracias a un sistema de paracaídas y airbags que frenaban la caída.

El desarrollo en computación y sistemas de control electrónico que se llevo a cabo durante esta época fue el que empezó a definir a los drones actuales.

En 1993 se terminaron de desplegar los satélites que formaban parte del novedoso **GPS**. Este sistema permitió a los operadores de drones pilotar los aviones de forma más eficiente ya que les liberaba de las restricciones de alcance y de los sistemas de navegación inexactos. A este nuevo sistema, se le sumaron los **sistemas digitales de control de vuelo** o Digital Flight Control System (DFCS), que dotaron de mayor alcance y precisión a los drones.

Un año más tarde, en 1994, empezó el desarrollo del *Predator* creado por las Fuerzas Aereas de EE.UU. con la colaboración de General Atomics Aeronautical Systems[18]. Fue en

[18] https://es.wikipedia.org/wiki/General_Atomics

1995 cuando se voló por primera vez con éxito. Está clasificado como un sistema de tipo UAV de techo medio y largo alcance.

Así, después de unos 200 años de evolución (véase Figura 3.4), es como se llega al dron actual.

Figura 3.4: Diversos tipos de UAVs a lo largo de los años. De izquierda a derecha y de arriba abajo: avión automático Hewitt-Sperry, Aquila RPV-drone, DASH , Ryan AQM-34L Firebee drone *Tom Cat*, MQ 9 Reaper Drone. `http://eldrone.es/historia-de-los-drones/`

Época actual: UAV Civil. Apartándonos de la evolución ligada al desarrollo militar, en la época actual, los drones han ido evolucionando hacia drones cada vez más pequeños, más inteligentes y, en escala, a drones con mayor rango y autonomía.

Su utilidad, desligada al uso militar, se ha centrado sobre todo en tareas de inspección de zonas remotas, vigilancia, topografía y uso recreativo. Cabe destacar, que aunque existen drones de ala fija, el dron mas común de uso civil y profesional fuera del uso militar, es un dron basado en hélice giratoria, ya sea de tres, cuatro u ocho motores.

Existen distintos modelos de uso civil y de coste reducido: DJI Phantom de DJI[19], SOLO de 3DR[20] y la serie Parrot[21] (véase Figura 3.5). Estos drones permiten realizar las tareas mencionadas anteriormente de forma fácil y sencilla gracias a sus sistemas de pilotaje inteligente. Cabe destacar que la empresa 3DR ha sido la impulsora del controlador *PixHawk*, basado en el diseño PX4 open-source, tanto el harware como el software del controlador es de código abierto bajo licencia BSD.

Figura 3.5: Parrot Airborne Cargo Travis. `https://www.parrot.com/es/minidrones/parrot-airborne-cargo-travis`

Como mención especial, en el año 2010, una empresa española, Flightech Systems[22], obtuvo el primer Certificado de Aeronavegabilidad Experimental de Europa para su UAV, el FT-ALTEA. Este UAV tiene 6 metros de envergadura, está equipado con cámaras térmicas y de alta definición, además de varios sensores, dispone también de un sistema de navegación autónomo que le permite ser pilotado durante el día y la noche, además de la capacidad de despegue y aterrizaje automático.

3.1.3.2. Clasificación

Hasta ahora se han definido todos los aviones no tripulados o como UAV o como Drones, incluso hemos llamado así a los primeros misiles balísticos, pero fue en los años 90 cuando el término UAV se hizo de uso común para definir a estas naves y reemplazo al termino RPV, término usado durante la Guerra de Vietnam para definir a los *Lightning Bugs*. Fue en la publicación del Ministerio de Defensa de EE.UU. *Joint Publication 1-02, Department of Defense Dictionary* [oS15], donde se definió por primera vez el término UAV:

[19] `http://www.dji.com/es/phantom`
[20] `https://3dr.com`
[21] `https://www.parrot.com`
[22] `http://flightechspanish.weebly.com`

«Un vehículo aéreo motorizado que no lleva a bordo a un operador humano, utiliza las fuerzas aerodinámicas para generar la sustentación, puede volar autónomamente o ser tripulado de forma remota, que puede ser fungible o recuperable, y que puede transportar una carga letal o no. No se consideran UAV a los misiles balísticos o semibalísticos, misiles crucero y proyectiles de artillería.»

Anteriormente, en este documento, se ha definido a los misiles balísticos como UAV,pero según la definicion del Ministerio de Defensa de EE.UU. la definición es errónea. También, y según esta definición, quedan excluidos los globos o dirigibles.

Pero, cabe mencionar, que los términos UAVs y RPV, introducidos recientemente en este documento, son solo dos de entre todos los términos que existen para definir las naves no tripuladas (véase Figura 3.6). Por ejemplo, podemos añadir el término Unmanned Aerial System (UAS) a la colección, que vendría a ser el conjunto de sistemas que rodea un UAV.

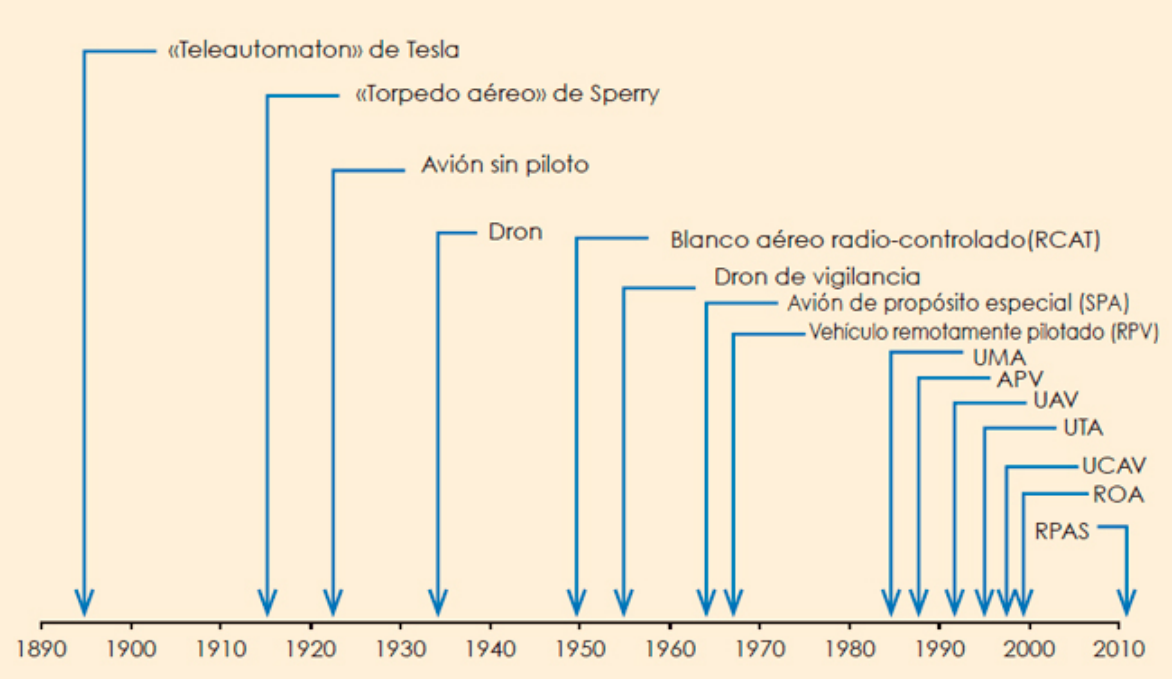

Figura 3.6: Cronología de los nombres aplicados a las aeronaves robóticas. `http://drones.uv.es/origen-y-desarrollo-de-los-drones/`

En 2011, la Organización de Aviación Civil Internacional (OACI)[23], al cual está suscrito España desde el Convenio de Chicago de 1944 en su circular 328, reconoció, a las aeronaves no tripuladas como aeronaves, y de todas sus definiciones escogió a los Remotely-Piloted Aircraft (RPA) como las únicas aptas para la aviación civil, excluyendo, por ejemplo, a las autónomas.

Un RPA es toda aquella aeronave no tripulada, pero con un piloto al mando. El término RPA introduce además el término **Remotely-Piloted Aircraft System (RPAS)**, que engloba

[23] `https://es.wikipedia.org/wiki/Organización_de_Aviación_Civil_Internacional`

todos los sistemas o elementos requeridos por el RPA para su vuelo, por ejemplo la estación de pilotaje remota asociada o **Remote Pilot Station (RPS)**.

Otros acrónimos son:

- Unmanned Aircraft (UMA)
- Automatically Piloted Vehicle (APV)
- Unmanned Tactical Aircraft (UTA)
- Unmanned Combat Air Vehicle (UCAV)
- Remotely Operated Aircraft (ROA)
- Micro Unmanned Aerial Vehicle (MUAV)

Dentro de los UAVs o Vehículo aereo no tripulado (VANT), se les clasifica en seis tipos[24]:

- **Blanco:** se utilizan para entrenamiento militar.
- **Reconocimiento:** capturan imágenes de un área definida para su posterior uso. Dentro de esta definición entran los RPAS o MUAV. Dado que la mayoría de los UAVs realizan tareas de reconocimiento, se ha tomado este término como el genérico para definir dicha aplicación.
- **Combate UCAV:** Son los UCAV, utilizados en combate y en misiones de alto riesgo.
- **Logística:** están diseñados para llevar cargas. En 2016 la empresa Amazon empezó a utilizar UAV para entregar paquetes[25] (véase Figura 3.7).
- **Investigación y desarrollo:** utilizados, como su nombre dice, para investigación y desarrollo de nuevas tecnologías.
- **UAV comerciales y civiles:** creados para uso civil. Tienen un amplio uso, aunque por lo general se utilizan para realizar imágenes o vídeo.

Esta ha sido la clasificación dependiendo de su uso pero también pueden ser clasificados dependiendo de su capacidad de ascensión y rango:

- **Handheld:** unos 600 metros de altitud y 2 km de alcance.
- **Close:** unos 1500 metros y hasta 10 km de alcance.
- **NATO:** unos 3000 metros de altitud y hasta 50 km de alcance.
- **Tactical:** unos 5500 metros de altitud y hasta 160 km de alcance.
- **Medium Altitude, Long Endurance (MALE):** hasta 9000 metros de altitud y un alcance de unos 200 km.

[24] https://es.wikipedia.org/wiki/Vehículo_aéreo_no_tripulado

[25] http://www.elconfidencial.com/tecnologia/2016-12-14/amazon-drones-entregas-envios_1303960/, http://omicrono.elespanol.com/2016/12/envios-por-drone-de-amazon/

Figura 3.7: Amazon Prime Air Drone. http://omicrono.elespanol.com/2016/12/envios-por-drone-de-amazon/

- **High Altitude, Long Endurance (HALE):** hasta 9000 metros de altitud y alcance indeterminado.
- **HYPERSONIC:** alta velocidad. Supersónico (Mach 1-5) o hipersónico (Mach 5+): unos 15000 metros de altitud o altitud suborbital y un alcance de 200 km.
- **ORBITAL:** en órbitas bajas terrestres (Mach 25+).
- **CIS lunar:** viaja entre la Luna y la Tierra.

3.1.3.3. Dominios de aplicación

Ya hemos visto que los UAVs se han utilizado a lo largo de su historia para entrenamiento militar, toma de fotografías, espionaje, vigilancia e, incluso, como armas, todas estas, aplicaciones orientadas a uso militar, pero fue, con su uso civil, cuando se le empezó a sacar verdadera utilidad.

En la actualidad, los UAVs civiles, realizan diversas tareas y aunque algunas tienen su origen militar, muchas otras no.

Relacionadas con su uso de vigilancia podemos encontrar diversas aplicaciones:

- **Captura de imágenes o vídeo en eventos:** gracias a la capacidad que tienen de grabación en alta calidad algunos UAVs y su capacidad de volar a baja altura, son perfectos para realizar grabaciones o fotografías donde los brazos de grúa no llegan o donde los helicópteros no entran.
- **Búsqueda de personas:** las capacidades de estos dispositivos de transmitir vídeo y su gran manejabilidad los hace perfectos para la búsqueda de personas en bosques o montañas.

- **Control fiscal:** en España ya se están utilizando para realizar fotografías de terrenos donde el Ministerio de Hacienda cree que se están realizando obras[26].
- **Control fronterizo:** la Guardia Civil de España los está utilizando para vigilar los ingresos marítimos en el estrecho.
- **Control de incendios forestales:** la capacidad que tienen estos dispositivos de permanecer en vuelo casi constante a un coste mínimo los hace idóneos para vigilar los bosques desde el cielo.
- **Investigaciones:** arqueológicas, geológicas, zoológicas, hidrológicas. Toda investigación que necesite visualizar el terreno desde un punto elevado se puede apoyar en el uso de UAV.
- **Cartografía** la capacidad de los UAVs, de ala fija o de ala rotatoria, de tomar fotos a gran altura, los hace perfectos para la realización de ortofotomapas y de modelos de elevaciones del terreno de alta resolución.

Existen otras utilidades desligadas de la vigilancia, que han surgido de la necesidades del uso civil:

- **Envíos:** Amazon ya ha realizado su primera entrega de un paquete mediante un UAV (véase Figura 3.7). En Australia otra empresa tambien ha realizado un trabajo similar[27]. En Estados Unidos se ha permitido entregar medicinas mediante UAVs, de forma legal, en el estado de Virginia, [28]. La empresa Airbus ha iniciado un proyecto para la creación de un dron que transporte mercancías y personas dentro de las ciudades[29] (véase Figura 3.8).
- **Usarlos como satélites:** se han creado proyectos para dispersar contenido mediante drones por todo el planeta, una de sus utilidades, propuestas por la empresa Facebook, es llevar internet a zonas remotas[30] (véase Figura 3.9).
- **Agricultura:** gestión de cultivos. Existen proyectos para la vigilancia o gestión de extensas aéreas de cultivo[31].
- **Monitorización de instalaciones:** gracias a los UAVs, los operarios de compañías

[26] http://www.elmundo.es/economia/2016/07/26/579621d6e5fdea243f8b4620.html

[27] https://www.businessinsider.com.au/australian-startup-flirtey-trialled-drone-delivery-in-new-zealand-and-it-was-4-times-quicker-than-driving-2015-7

[28] http://www.nbcnews.com/tech/tech-news/first-faa-approved-drone-delivery-drops-medicine-virginia--n393986

[29] http://www.xdrones.es/movilidad-urbana-del-futuro-drones/

[30] http://www.infobae.com/tecno/2016/07/22/asi-es-aquila-el-drone-de-facebook-que-llevara-internet-a-todo-el-mundo/
https://www.xataka.com/drones/asi-fue-el-primer-vuelo-largo-de-quila-el-dron-para-llevar-internet-de-facebook

[31] http://www.innovagri.es/investigacion-desarrollo-inovacion/uso-de-drones-para-una-agricultura-de-detalle.html

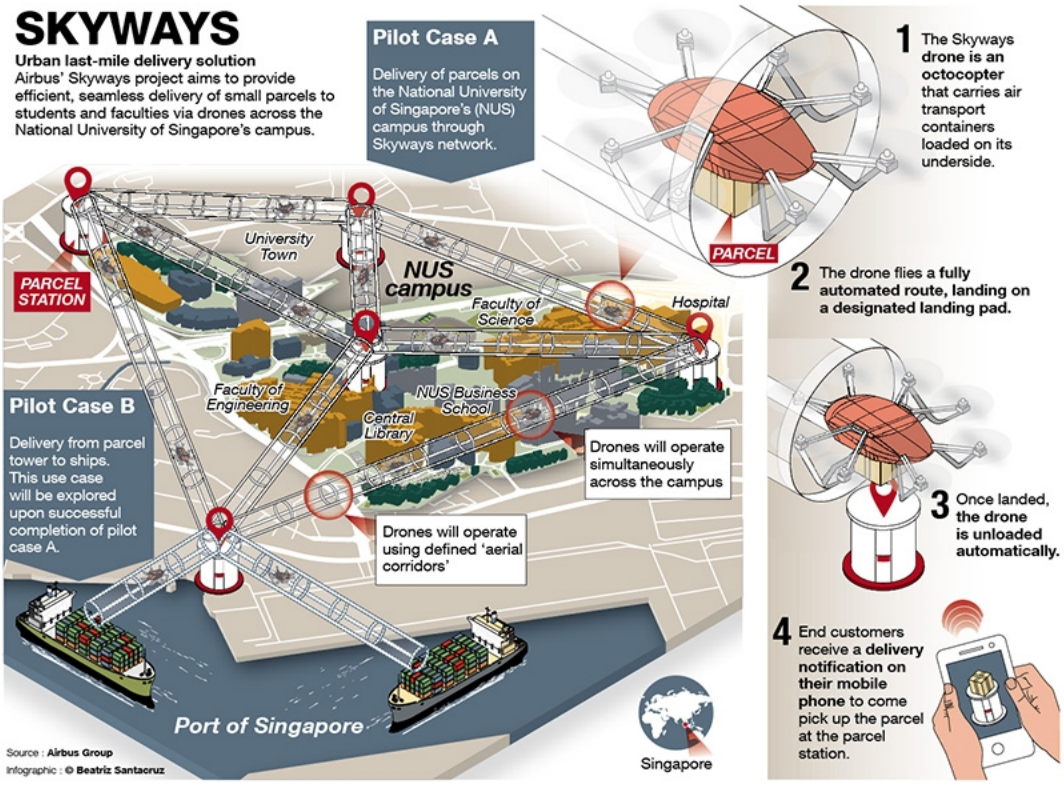

Figura 3.8: Drones de Airbus. `http://www.xdrones.es/movilidad-urbana-del-futuro-drones/`

eléctricas ya no tienen que subir a las torres de alta tensión o a los molinos de viento[32].

- **Uso de entretenimiento:** La búsqueda de UAVs cada vez más pequeños, inteligentes y baratos ha llevado estos dispositivos al ámbito del entretenimiento.

En resumen, la gran capacidad que tienen estos dispositivos para acceder a sitios remotos, su gran versatilidad al poder portar distintos dispositivos controlados remotamente y la capacidad de dotarlos de inteligencia, los hace perfectos para cubrir un amplio espectro de necesidades de uso civil. El único límite está en la legislación de cada país.

3.1.3.4. Entorno de desarrollo

Como hemos visto, la mayoría de las aplicaciones relacionadas con los UAVs están centradas en la vigilancia, en estas aplicaciones, la inteligencia del UAV, se centra en la manejabilidad y estabilidad de la aeronave, pero no por ello significa que no se les pueda dotar de una mayor inteligencia.

Actualmente existen distintas formas de pilotar automáticamente un UAV.

MAVLink y el uso de scripts. Micro Air Vehicle Link (MAVLINK)[33] es un protocolo de comunicaciones para MUAV. Fue lanzado en 2009 por Lorenz Meier bajo licencia Lesser

[32] `http://noticiasdelaciencia.com/not/16200/drones-para-inspeccionar-lineas-electricas`
[33] `https://en.wikipedia.org/wiki/MAVLink`

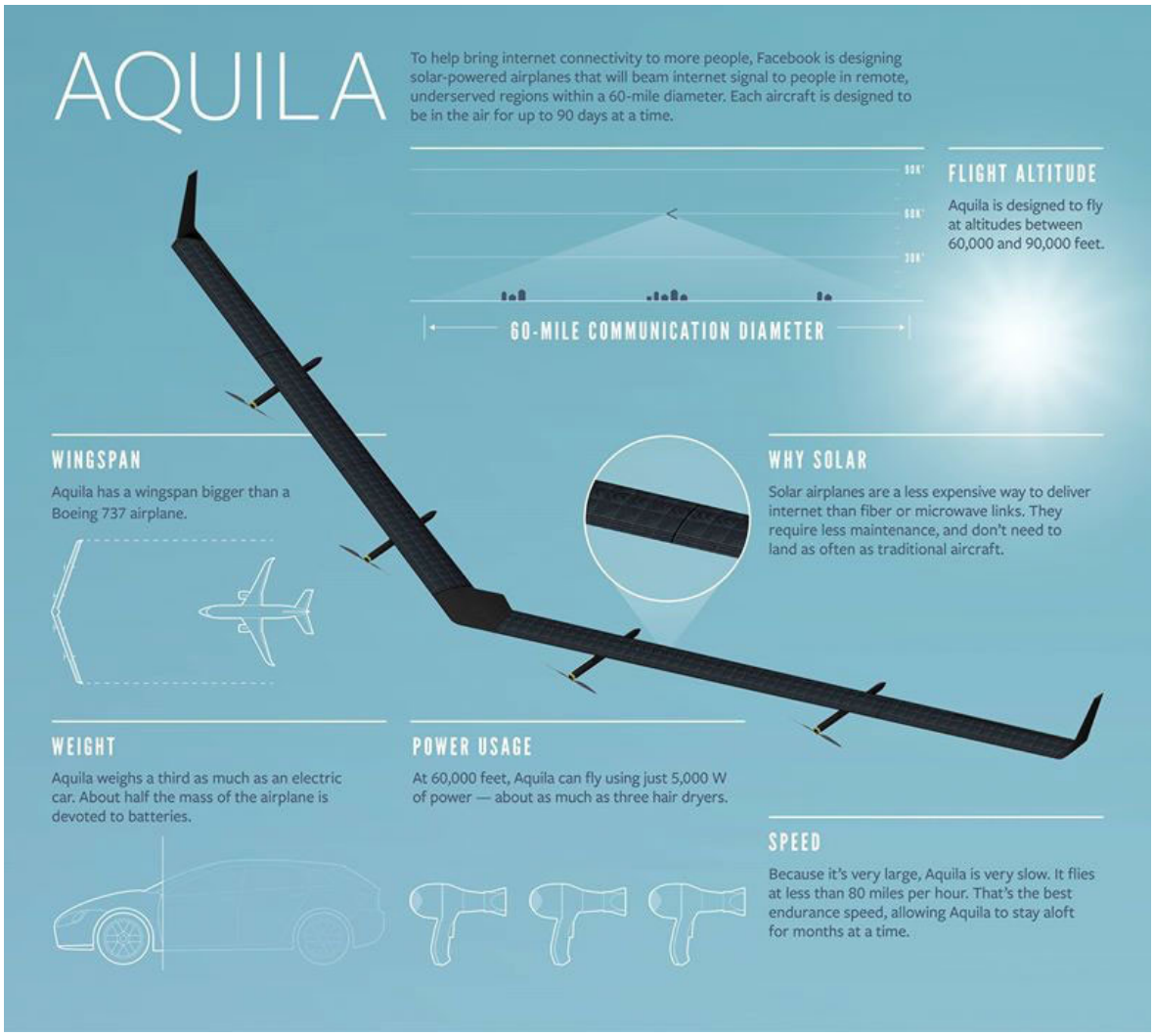

Figura 3.9: Aquila, el dron de Facebook. `https://www.xataka.com/drones/asi-fue-el-primer-vuelo-largo-de-aquila-el-dron-para-llevar-internet-de-facebook`

General Public License (LGPL).

Este protocolo se utiliza para las comunicaciones entre una estación de tierra o GCS y el UAV y para la inter-comunicación de los subsistemas del vehículo. Gracias a este protocolo podemos obtener la telemetría del UAV (orientación, localización GPS, velocidad, estado, etc...) y controlarlo remotamente.

Todos los mensajes de MAVLink están definidos con la misma estructura, que aunque de tamaño variable, está definida según la el cuadro 3.3.

Cada paquete enviado es una acción realizada por el UAV.

En el mercado existen distintos UAVs que utilizan controladores que, internamente, usan MAVLink para su sistema de comunicaciones. Estos controladores son el cerebro del UAV y aparte de las comunicaciones también gestionan el autopiloto.

Nombre de campo	Índice	Propósito
Start-of-frame	0	Inicio del mensaje (v1.0: 0xFE).
Payload-length	1	Tamaño de la carga (n).
Packet sequence	2	Contador de secuencia. Otorga capacidad al sistema para la detección de pérdida de paquetes.
System ID	3	Identificador del sistema de envío. Se permiten diferentes sistemas en la misma red.
Component ID	4	Identificador del componente de envío.
Message ID	5	Identificador del mensaje. El significado de la carga depende del identificador del mensaje.
Payload	6 to (n+6)	Carga.
CRC	(n+7) to (n+8)	Check-sum.

Cuadro 3.3: Estructura de los mensajes de MAVLink

Uno de esos controladores es el modelo **Pixhawk**[34] que está basado en **Ardupilot**. *Pixhawk* esta creado por la empresa 3DR Robotics que ofrece un kit de desarrollo llamado **DroneKit**[35] para UAV basados en *Ardupilot*, que se puede utilizar tanto en lenguajes Android, iOS, *Python* o incluso en la nube. Gracias a este kit, la programación del UAV se vuelve mucho más fácil, evita al programador tener que definir todos los tipos de mensajes que se pueden enviar por MAVLINK y nos abstrae de su uso, con el kit de desarrollo es tan fácil como ejecutar el código (véase Listado 3.1) y obtener la mayoría de los datos de telemetría del UAV.

Esta capacidad de enviar y recibir paquetes mediante MAVLINK, ya sea mediante *DroneKit* o sin una biblioteca externa, lo hace idóneo para lenguajes de script.

Otra utilidad que nos otorga MAVLINK es la capacidad de realizar misiones, entendiendo por misiones a un conjunto de comandos que pueden ser ejecutados en orden y de forma repetida, y que en lenguaje común seria como decirle al UAV :

- Despegar a 10 metros de altura.
- Vuela 10 metros al norte.
- Vuela 10 metros al sur.
- Aterriza.

SITL Simulator y MAVProxy. Software In The Loop (SITL) Simulator[36] es un simulador que permite controlar distintos tipos de UAV, ya sean de ala fija o de ala rotatoria, además de robots terrestres denominados rovers, y que permite ejecutar *Ardupilot* directamente en tu Personal Computer (PC) sin necesidad de Hardware (HW) especial, esto es debido a que *Ardupilot* es portable y funciona en numerosas plataformas.

[34] `https://pixhawk.org`
[35] `http://dronekit.io`
[36] `http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.htm`

```
# vehicle is an instance of the Vehicle class
print "Autopilot Firmware version: %s" % vehicle.version
print "Autopilot capabilities (supports ftp): %s" % vehicle.capabilities.ftp
print "Global Location: %s" % vehicle.location.global_frame
print "Global Location (relative altitude): %s" % vehicle.location.global_relative_frame
print "Local Location: %s" % vehicle.location.local_frame    #NED
print "Attitude: %s" % vehicle.attitude
print "Velocity: %s" % vehicle.velocity
print "GPS: %s" % vehicle.gps_0
print "Groundspeed: %s" % vehicle.groundspeed
print "Airspeed: %s" % vehicle.airspeed
print "Gimbal status: %s" % vehicle.gimbal
print "Battery: %s" % vehicle.battery
print "EKF OK?: %s" % vehicle.ekf_ok
print "Last Heartbeat: %s" % vehicle.last_heartbeat
print "Rangefinder: %s" % vehicle.rangefinder
print "Rangefinder distance: %s" % vehicle.rangefinder.distance
print "Rangefinder voltage: %s" % vehicle.rangefinder.voltage
print "Heading: %s" % vehicle.heading
print "Is Armable?: %s" % vehicle.is_armable
print "System status: %s" % vehicle.system_status.state
print "Mode: %s" % vehicle.mode.name    # settable
print "Armed: %s" % vehicle.armed    # settable
```

Listado 3.1: Obtención de los datos del UAV

Los datos del sensor son simulados a partir de un simulador de vuelo incorporado en SITL. *Ardupilot* tiene integrado una gran gama de simuladores de vehículos, además SITL permite utilizar simuladores externos.

SITL define una serie de puertos que pueden ser usados por aplicaciones externas. Como podemos ver en la Figura 3.10 el GCS se conecta mediante protocolo TCP al puerto 5760 y **MAVProxy** al puerto 5763. El puerto UDP 14550 puede ser utilizado para conectar otro tipo de GCS

SITL puede utilizar **MAVProxy**[37] como GCS. *MAVProxy* es una estación de control de tierra totalmente funcional que soporta el protocolo MAVLINK. *MAVProxy* está programada en *Python* y puede ser ejecutada (véase Figura 3.11) en cualquier sistema operativo Portable Operating System Interface UNIX (POSIX), como por ejemplo Linux, Windows y OS X.

Parrot SDK. Otro kit de desarrollo es el Parrot Software Development Kit (SDK)[38], creado por la empresa Parrot para sus UAVs. Este kit está basado en lenguaje C++ o XCode y como el *DroneKit* permite, al programador, abstraerse de toda la complejidad de los UAVs y realizar una programación fácil y sencilla. Actualmente está en su versión número 3. Además, es compatible con las misiones de vuelo de MAVLINK.

[37] http://ardupilot.github.io/MAVProxy/html/index.html

[38] http://developer.parrot.com/

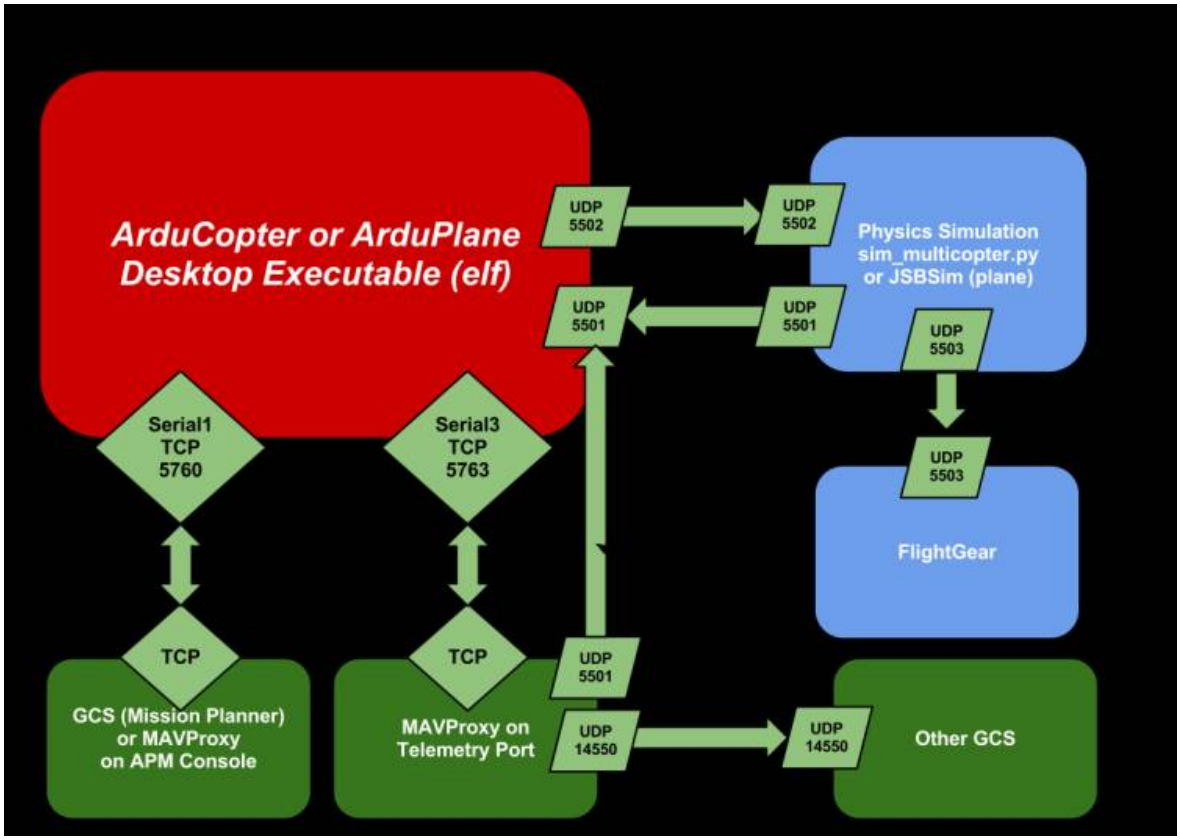

Figura 3.10: Ejecución de MAVProxy `http://ardupilot.org/dev/docs/copter-sitl-mavproxy-tutorial.html`

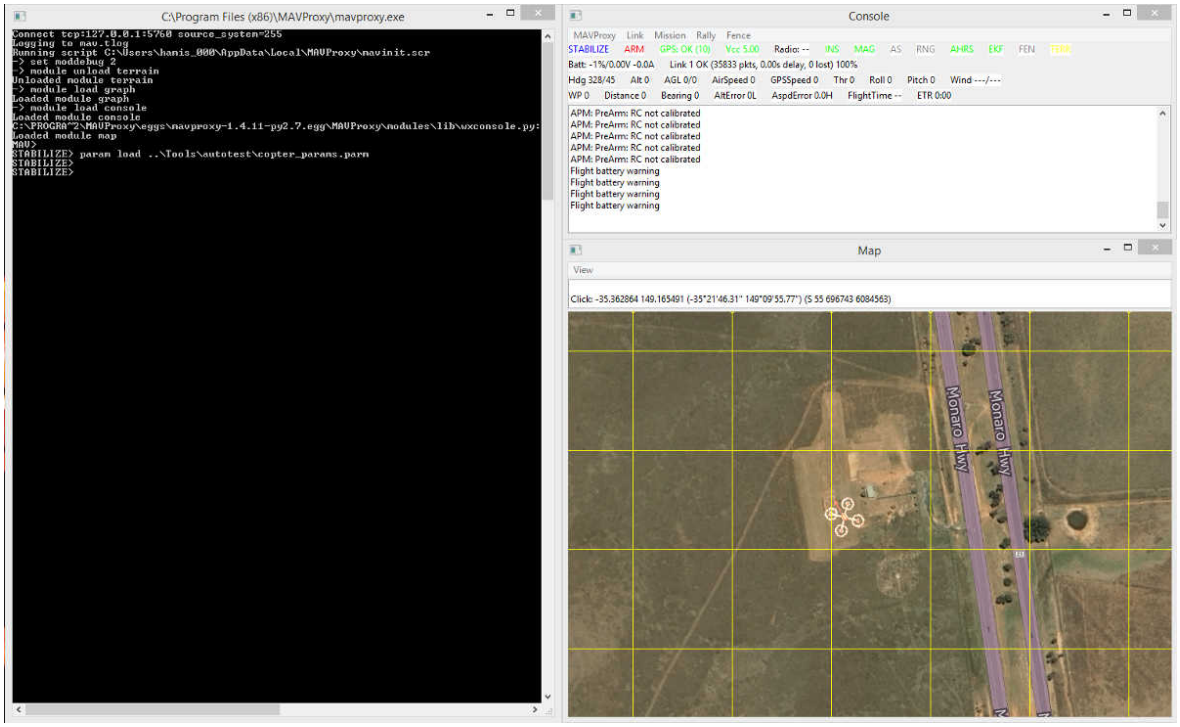

Figura 3.11: Estructura de SITL. `http://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html`

El SDK, dado que los UAVs de la empresa Parrot utilizan Bluetooth o Wifi para su conexión con la estación de tierra, permite buscar en distintas redes a los dispositivos, una vez localizados puede obtener o enviar la información de los UAVs como si de un socket de red se tratase. Este es uno de los grandes beneficios respecto al uso de MAVLink para la conexión remota.

Dado que el SDK utiliza un lenguaje orientado a objetos, de alto nivel y que permite la conectividad con otros lenguajes, se puede otorgar al UAV de mayor inteligencia, por ejemplo, se puede realizar una aplicación desarrollada en lenguaje *Python* y que se comunique con el controlador del UAV programado en C++, o programar una aplicación web que se comunique con el controlador programado en C++.

Sphinx: El simulador de Parrot. *Sphinx*[39] es una herramienta de simulación gratuita que cubre las necesidades de los desarrolladores de software para UAV de la empresa *Parrot*. Al igual que el simulador utilizado con *Ardupilot* está basado en una simulación SITL. *Sphinx* permite configurar el entorno y el UAV mediante archivos *.worl* y *.drone* respectivamente. Está basado en el simulador de código abierto *Gazebo*[40] que otorga un entorno 3D para la simulación (véase Figura 3.13).

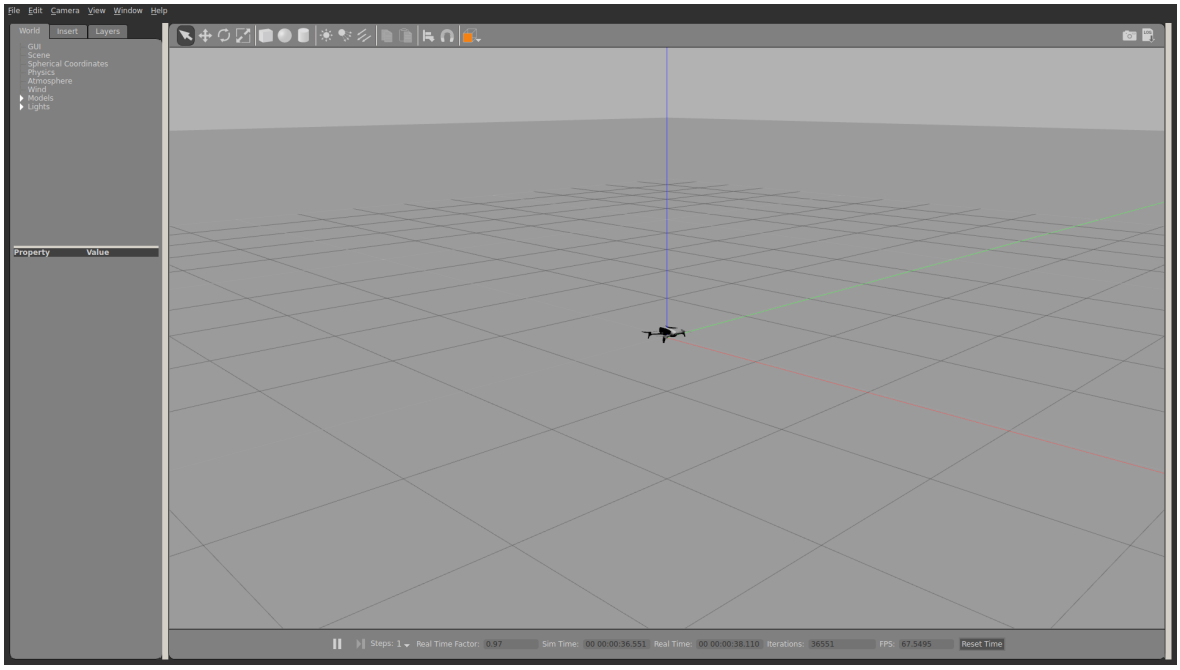

Figura 3.12: Entorno 3D de Sphinx. `https://developer.parrot.com/docs/sphinx/firststep.html`

[39] `https://developer.parrot.com/docs/sphinx/index.html`

[40] *«Gazebo es un simulador de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma personalizada, crear mundos virtuales usando sencillas herramientas CAD e importar modelos ya creados. »* `https://moodle2015-16.ua.es/moodle/mod/book/view.php?id=82546&chapterid=2401`, `http://www.gazebosim.org/`

Permite simular los siguientes UAVs de la marca *Parrot*[41]:

- Parrot Bebop
- Parrot Bebop2
- Parrot Disco
- Parrot Airborne
- Parrot Mambo
- Parrot Swing

Además, para el análisis del vuelo, otorga un panel, vía web, en el cual se pueden configurar las graficas que se deseen mostrar en tiempo real[42](véase Figura 3.13).

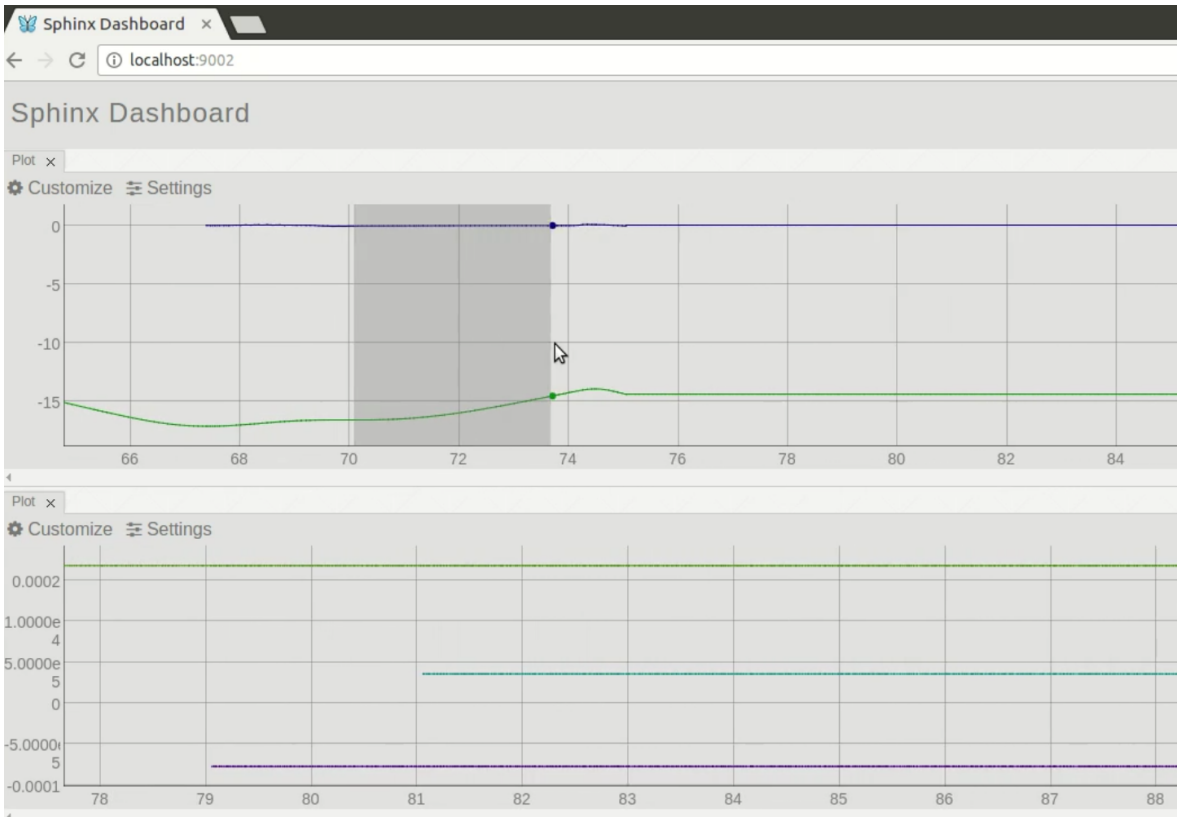

Figura 3.13: Tablero de mandos de Sphinx. `https://developer.parrot.com/docs/sphinx/visualization.html`

DroneKit vs Parrot SDK. Ambos kits de desarrollo permiten al programador abstraerse de las complejidades del UAV, pero dado que el SDK de Parrot utiliza lenguaje C++ lo hace más útil para un desarrollo más complejo.

También y aunque no está relacionado con la calidad de los entornos de desarrollo, cabe destacar que, los UAVs basados en *Ardupilot* que ofrecen diversas empresas, suelen ser UAVs

[41] `http://forum.developer.parrot.com/t/introducing-parrot-simulator-sphinx/5800`
[42] `https://developer.parrot.com/docs/sphinx/firststep.html`

de un tamaño considerable si los comparamos con la gama baja de Parrot, lo que para la investigación y el desarrollo de pruebas de concepto es un inconveniente dado que a mayor tamaño mayor debe ser la zona de pruebas

3.1.4 Sistema multi-agente

El núcleo de un sistema multi-agente, es el **agente**: un agente es una unidad *inteligente*, equivalente a un proceso de un sistema operativo, que realiza la tarea para la que se ha diseñado sin necesidad de que se le diga explícitamente que hacer en cada momento, denotando, así, independencia. Un agente además dispone de varias características: **autonomía**, **visión local** y **descentralización**. Un sistema multi-agente consiste en un número no determinado de agentes que interactúan entre ellos, normalmente mediante mensajes enviados utilizando mecanismos de comunicación, estos agentes realizan tareas conjuntas y necesitan ser capaces de cooperar, coordinarse y negociar entre ellos tal y como nosotros cooperamos, nos coordinamos y negociamos en el día a día.

Aunque se ha mencionado que los agentes son *inteligentes* dicha inteligencia puede ser individual, donde el agente denota inteligencia por sí solo, o conjunta, donde es el sistema multi-agente el que es inteligente a partir de las acciones que realizan sus agentes.

A lo largo de la historia de la computación hasta la fecha se han mantenido las mismas tendencias [Woo01a]:

- Ubicuidad.
- Interconexión.
- Inteligencia.
- Delegación.
- Orientación a las personas.

Por **ubicuidad** entendemos la capacidad que tiene la computación de estar presente en diversos dispositivos y esto ha sido posible gracias a la reducción de coste de computación y al abaratamiento de dichos dispositivos.

Aunque los dispositivos antiguos se veían como entidades aisladas que solo se comunicaban con su operador, la evolución ha llevado a los dispositivos por el camino de la **conectividad**. Actualmente, y gracias a la evolución de Internet, se busca diseñar los sistemas de computación como sistemas distribuidos donde cada dispositivo sea un proceso de interacción.

A lo largo de estos años, el entendimiento de cómo se realizan las tareas nos ha llevado a poder delegar tareas más complejas a los sistemas de computación, pudiéndose decir que cada vez son más **inteligentes**.

Actualmente se delegan multitud de tareas a los sistemas computacionales que anteriormente eran impensables. **Delegar** dichas tareas implica dar el control al sistema para que tome decisiones sin supervisión. Por ejemplo, en la historia de los UAVs, se ha visto que es posible delegar ciertas decisiones y tareas relacionadas con el control de vuelo, de un avión, a un sistema de computación.

La última tendencia es la **orientación a las personas**. Cada vez vemos más tendencia a crear lenguajes de alto nivel que nos abstraen de las complejidades de la computación y hace que programar se asemeje más a como vemos el mundo. Esta tendencia se ve reflejada en el cambio que se ha realizado en como interactuamos con los computadores, desde los primeros ordenadores programables mediante interruptores, hasta los *nuevos* ordenadores con sistemas operativos gráficos pasando por los ordenadores manejados mediante líneas de comandos. Esta evolución, nos ha llevado, cada vez más, a un manejo a mas alto nivel que no requiere un gran entendimiento del funcionamiento interno de los sistemas de computación.

Estas tendencias han llevado a la creación de un nuevo campo en la ciencia de la computación: Los **sistemas multi-agente**.

Los sistemas multi-agente introducen dos nuevas problemáticas: Como diseñamos agentes que sean independientes y capaces de realizar las tareas para las que han sido diseñados, y como diseñamos como interactúan dichos agentes entre ellos de modo que completen sus áreas aunque exista conflictos entre ellos.

Estos problemas nos hacen enfocar la creación de los sistemas desde dos puntos de vista. El primer problema está relacionado con el **diseño del agente** y el segundo con el **diseño de las sociedades**.

Para desarrollar estos sistemas multi-agente se han creado **metodologías de software orientadas a agentes** o Agent Oriented Software Engineering (AOSE)[43] y notaciones. Dichas metodologías y notaciones especifican artefactos de desarrollo orientados a la creación de sistemas basados en agentes. En el cuadro 3.4 se puede ver unos ejemplos de metodologías y notaciones orientadas a agentes.

Russel y Norving definieron cinco tipos de grupos de agentes basándose en su grado de inteligencia y capacidad [RN03]:

- **Simple reflex agents:** son agentes simples, que basan su actividad en condiciones. Si la condición se cumple el agente realiza una acción. Estos agentes solo pueden ser diseñados si tienen constancia de todo el entorno (véase Figura 3.14).
- **model-based reflex agents:** similares a los anteriores. Los agentes basados en modelos, conocen la totalidad del entorno gracias a un modelo interno que lo define.

[43] https://es.wikipedia.org/wiki/Sistema_multiagente

Nombre	Creador	Descripción
Vocales (Voyelles)	Yves Demazeau	Una de las primeras propuestas en el área, y considera la concepción de sistemas multiagentes desde varios puntos de vista, correspondientes a las vocales: Agente, Entorno, Interacciones, y Organización.
GAIA	Michael Wooldridge y Nick Jennings	Propone cómo realizar un análisis basado en roles del sistema multi-agente.
MASE	Scott A. Deloach	Propone agentes como extensiones de objetos y proporciona la herramienta AgentTool para análisis, diseño e implementación.
AgentUML	James Odell	Propone una notación, extendiendo UML, para especificar protocolos de comunicación entre agentes.
MADKiT	Jacques Ferber	Esta basada en el paradigma Agente-Role-Organización de la metodología Aalaadin.
ADELFE	Grupo IRIT de la Universidad de Toulouse	Trata especialmente los temas de cooperación entre agentes.
INGENIAS	Grupo GRASIA de la UCM	Extiende la metodología MESSAGE y proporciona un conjunto de herramientas para modelar y generar código de sistemas multiagente.
Mas-CommonKADS	Carlos Iglesias	Extiende la metodología CommonKADS, para sistemas expertos, a agentes, utilizando estructuración orientada a objetos y lenguajes de especificación de protocolos como SDL.
SemanticAgent	Grupo LIRIS de la Universidad de Lyon	Basada en el SWRL.
NetLogo	Desarrollado por el CCL	Es una plataforma orientada a los sistemas de multi-agente, para modelar complejidad.
CORMAS (COmmon Resources Multi-Agent System)		Es un framework de desarrollo de sistemas multi-agente, de código abierto y basado en el lenguaje de programación orientada a objetos SmallTalk. Trata los temas de investigación científica para el desarrollo agrícola y para la negociación entre los actores.

Cuadro 3.4: Metodologías y notaciones de ingeniería de software orientada a agentes

- **goal-based agents:** se basan en los agentes de modelos y añaden el concepto de **goal** o **meta**. Internamente tienen definidas estas metas y las utilizan para elegir que acción realizar.
- **utility-based agents:** están basados en los *goal-based agents*, pero no diferencian entre los estados que están dentro de las metas y los que no. Estos agentes añaden una función de **utilidad** a sus metas y definen cual es el textbfgrado de aceptación que están dispuestos a aceptar (véase Figura 3.15) .
- **learning agents:** como su nombre indica, estos agentes, pueden realizar tareas en entornos desconocidos e ir aprendiendo . Reciben un *feedback* que les informa sobre como lo están haciendo y determinan que modificar para mejorar.

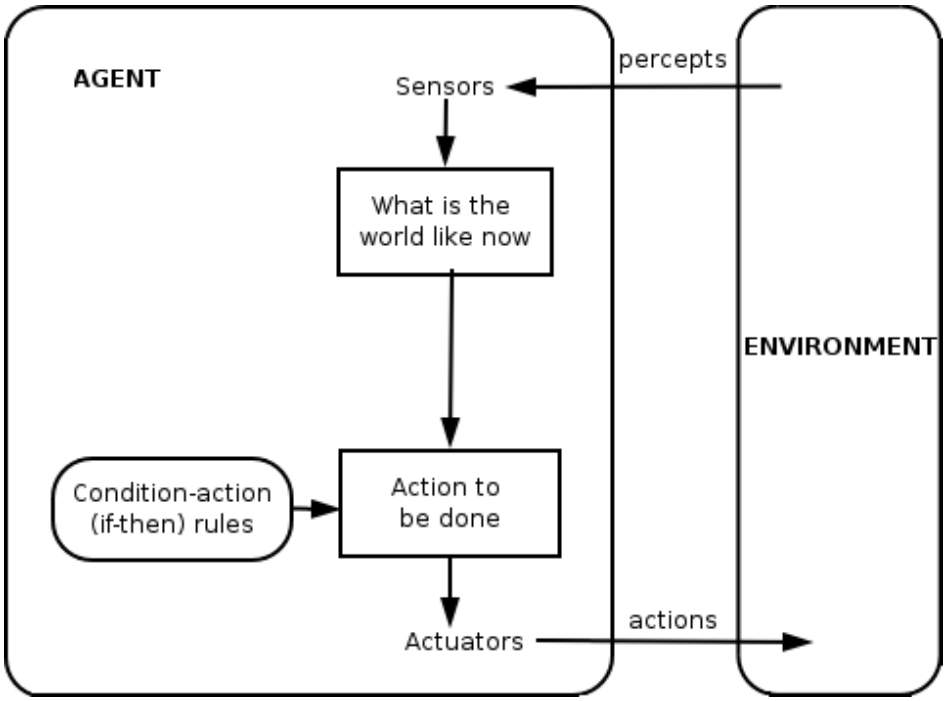

Figura 3.14: Simple reflex agent. https://en.wikipedia.org/wiki/Intelligent_agent

La Foundation for Intelligent Physical Agents (FIPA)[44], creada en 1996, fue creada con la intención de definir un conjunto completo de normas para la implementación de sistemas multi-agentes. A lo largo de los años ha contado con miembros distinguidos como Hewlett-Packard, IBM, BT, Sun Microsystems, Fujitsu y muchos más. Aunque, algunas plataformas adoptaron sus estándares, se disolvió en 2005 y se creó el comité de estándares **IEEE**.

Los estándares FIPA más ampliamente adoptados son las especificaciones **Agent Management** y **Agent Communication Language (FIPA-ACL)**.

Sistemas que usan estándares FIPA:

- Jade.
- Java Intelligent Agent Compontentware (JIAC).
- The SPADE Multiagent and Organizations Platform (Picon).
- JACK Intelligent Agents (Java).
- April Agent Platform (AAP) and Language (April).
- Zeus Agent Building Toolkit.
- The Fipa-OS agent platform.

[44] https://es.wikipedia.org/wiki/Foundation_for_Intelligent_Physical_Agents, http://www.fipa.org/

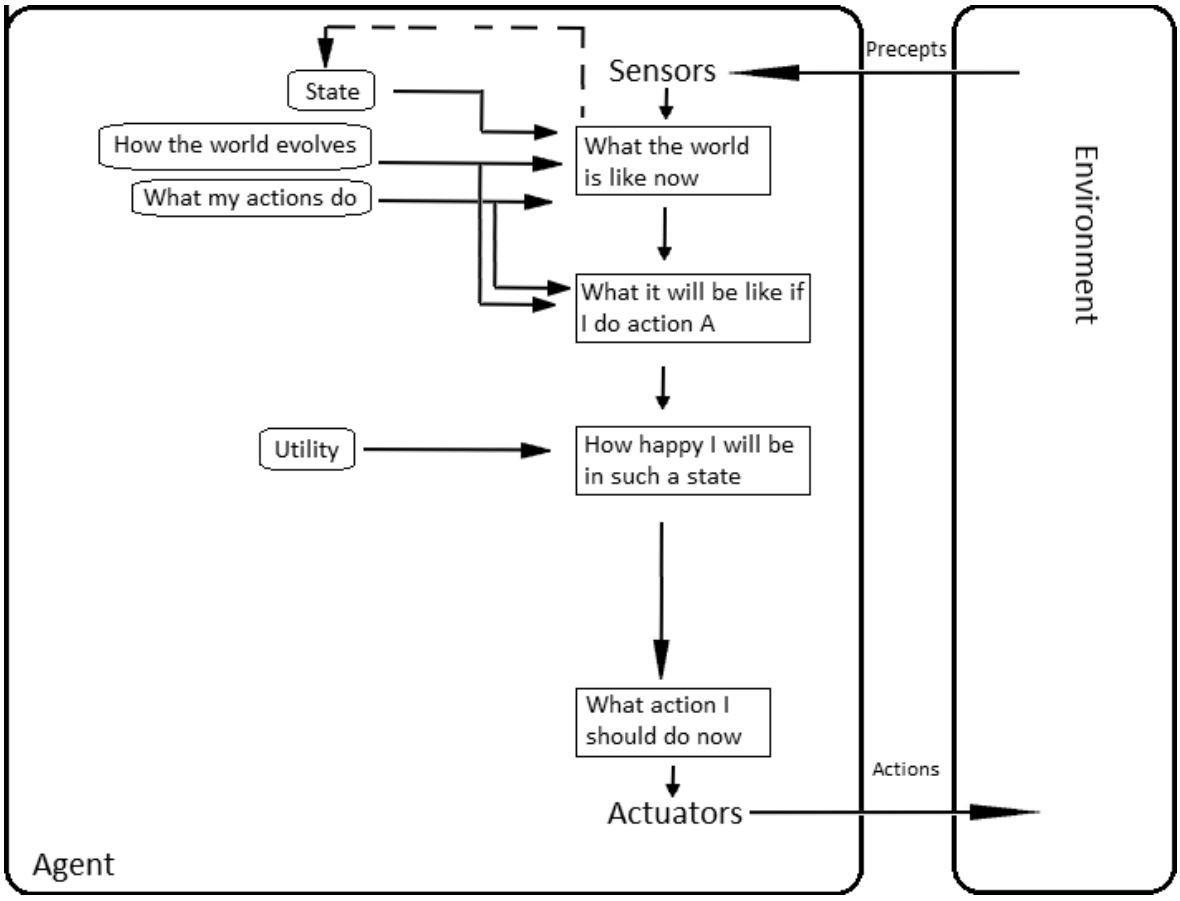

Figura 3.15: Model-based, utility-based agent. https://en.wikipedia.org/wiki/Intelligent_agent

3.2 Programación concurrente

3.2.1 Introducción

Vivimos en un mundo distribuido. Cada uno de nosotros realiza tareas de forma independiente, simultanea, compartiendo recursos y haciendo colas, por lo que lógicamente la programación concurrente es algo necesario para describir dichas tareas que no se atañen a un paradigma secuencial.

Asombraría saber que, los primeros desarrollos de computación concurrente datan de finales del siglo XIX[45], cuando el telégrafo y los ferrocarriles desarrollaron soluciones para su problema de **recursos compartidos**. A nivel académico, el estudio de los algoritmos concurrentes empezó en los años 60, cuando **Dijkstra**[46] publicó en la revista *Communications of the ACM.* su articulo *Solution of a problem in concurrent programming control* [Dij65b] en el año 1965, estableciendo y resolviendo la **exclusión mutua**.

Entre las definiciones que podemos encontrar en la Real Academia Española (RAE), sobre concurrencia, es la que la define como *«coincidencia, concurso simultáneo de varias*

[45] https://en.wikipedia.org/wiki/Concurrent_computing#History
[46] https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

circunstancias»[47] la que nos interesa.

En ciencias de la computación, la concurrencia, o **computación concurrente**, es la capacidad que tienen los sistemas de cómputo de realizar tareas, ya sean un conjunto de procesos o hilos de ejecución creados por un único programa, simultáneamente. Los cálculos computacionales, lo que comúnmente se conoce como operaciones, pueden ser realizados por un **único procesador** que alterna entre operaciones de distintas procesos o en **múltiples procesadores** que pueden estar o no separados físicamente. Dado que la Central Processor Unit (CPU) solo puede tener un proceso activo a la vez, el paralelismo estrictamente dicho, es **simulado**, y por eso este tipo de computación sea defina estrictamente como concurrente y no como simultanea.

La **programación concurrente** engloba las técnicas utilizadas para expresar el paralelismo entre tareas y solventar los problemas de comunicación y sincronización entre procesos y la gestión del acceso a recursos compartidos. Uno de los principales problemas de la programación concurrente es asegurar un orden de los procesos críticos y que el orden de ejecución de los procesos no críticos no altere el resultado. Dicha programación es muy similar a la **programación paralela**, solo que como ya se ha mencionado, esta, se centra más en la interacción entre tareas.

Dicho esto, se podría llegar a la conclusión de que este tipo de programación tiene una complejidad superior a la programación serial, por la necesidad que tiene de estudiar las comunicaciones y sincronizaciones entre procesos y las técnicas de recursos compartidos pero una vez entendidas las bases de la programación concurrente nos damos cuenta de las ventajas que ofrece [GS97].

En los años 50, se desarrollaron las primeras máquinas con varios procesadores. Estas máquinas disponían de un procesador de propósito general y varios procesadores destinados a los procesos de entrada y salida. Aunque técnicamente no se trataba de ejecución concurrente sí que fue un avance para la computación concurrente dado que permitía a la máquina realizar operaciones mientras se estaban utilizando las entradas y salidas.

A principios de los años 60, aparecieron los primeros multiprocesadores reales, en los que el programador de tareas del sistema operativo repartía, gracias a una **cola de trabajos**, los programas entre los distintos procesadores.

Antes de este avance, los sistemas operativos eran **sistemas monoprogramdos** que se bloqueaban al realizar operaciones de entrada y salida, dado que estas eran más lentas, dejando ocioso al procesador. Las colas de trabajo permitió a los sistemas operativos transformarse en **sistemas multiprogramados** permitiendo al sistema operativo desplazar un proceso bloqueante y realizar un *cambio de contexto* para así mantener al procesador ocupado con otro proceso de la cola. Este mecanismo supuso un mejor reparto de carga de trabajo y por

[47] `http://dle.rae.es/srv/fetch?id=AAkKXT5`

consiguiente un mayor rendimiento del sistema gracias a un mejor aprovechamiento de la CPU.

Fue ya en 1972, con la aparición del lenguaje de programacion Concurrent Pascal[48], desarrollado por Per Brinch Hanse[49], cuando se abrieron las puertas a otros lenguajes de alto nivel que incorporaban concurrencia.

Desde ese momento y hasta estos días, la programación concurrente ha ido ganando interés y actualmente se utiliza en diversos sistemas, desde un simple chat hasta complejos motores de física.

3.2.2 Conceptos básicos

En la programación concurrente se tienen que dejar muy claros los términos para no llevar a confusión, por ejemplo, no es lo mismo un proceso, o tarea, que un programa.

Una **tarea** o **proceso** es, a grandes rasgos, un programa en ejecución. Explicado con mayor detalle, un proceso es una abstracción del Sistema Operativo (SO), una abstracción lógica, que no física, que permite al SO ejecutar varios fragmentos de un programa simultáneamente. El SO dota al proceso de cierta independencia, evita que tenga interferencias y le asegura cierta seguridad en su ejecución. Un proceso se ejecuta en un fragmento de memoria a la que solo él tiene acceso, por lo que para comunicar dos procesos se tienen que utilizar mecanismos de comunicación otorgados por el entorno. A nivel de computo, los procesos son pesados, la creación, comunicación y cancelación de procesos requiere una elevado procesamiento, siendo el SO el encargado de realizar dichas acciones.

Entendemos un **programa concurrente** como un conjunto de procesos. Cada proceso es considerado como un programa secuencial, con un único flujo de control. Por lo tanto, se deduce que, para que un programa concurrente se ejecute de forma correcta es necesario definir que procesos lo componen, que información intercambian y como se sincronizan.

En la figura 3.16 se puede ver los estados por los cual pasa un proceso una vez se ha creado al ejecutarse un programa.

Un **proceso padre**, o proceso inicial de un programa, puede crear otros procesos, denominados procesos hijos. Padre e hijos pueden ejecutarse concurrentemente o el padre puede esperar a que los procesos hijos terminen su ejecución.

Otro concepto, creado relativamente hace poco, es el de **hilo** o **thread**. Existen SO **monohilo**, con un solo hilo de ejecución, y SO **multi-hilo**, la diferencia reside en que los SO multi-hilo disponen de una pila de proceso para cada hilo[50]. Además, si antes decíamos que los procesos son denominados como pesados, debido a su elevado procesamiento, los hilos son definidos como procesos ligeros ya que solo disponen de los recursos necesarios para

[48] https://en.wikipedia.org/wiki/Concurrent_Pascal

[49] https://en.wikipedia.org/wiki/Per_Brinch_Hansen

[50] http://systope.blogspot.com.es/2012/05/procesos-e-hilos.html

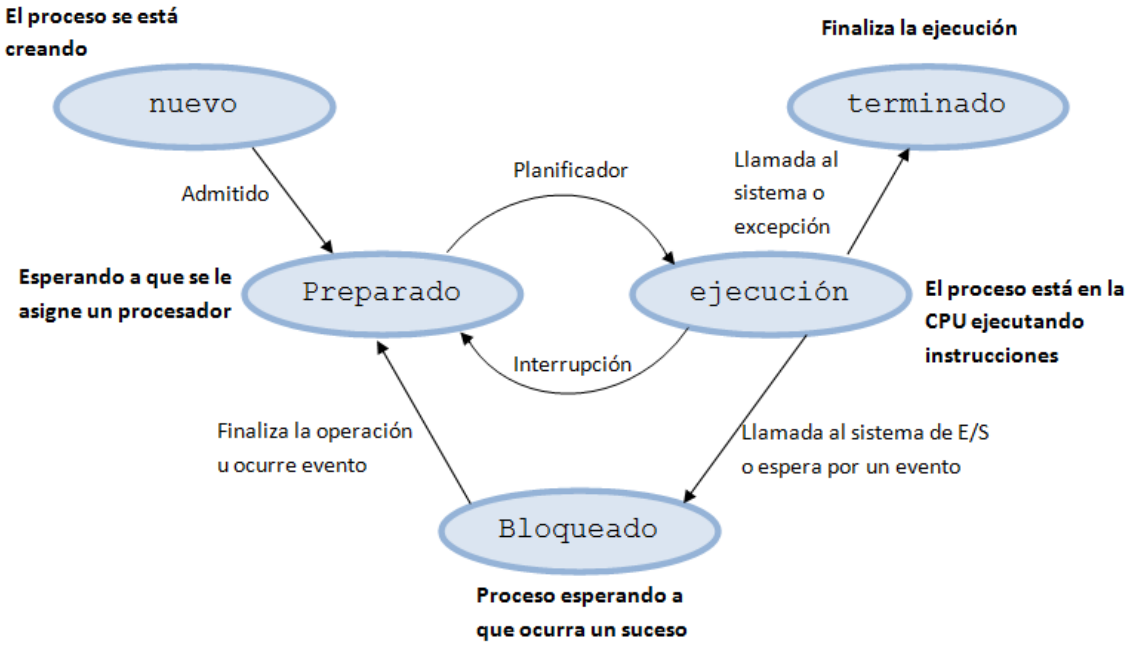

Figura 3.16: Diagrama de estados de un proceso

mantener su independencia de ejecución. Un proceso pesado es un proceso con un solo hilo. A nivel mas detallado, un hilo consiste en:

- Contador de programa.
- Juego de registros
- Espacio en pila.

Los hilos creados por la misma aplicación comparten su sección de código, los datos y los recursos del SO por lo que pueden comunicarse de forma no segura mediante su memoria local dado que comparten el mismo espacio de memoria de direcciones y la misma estructura de datos, o de forma segura mediante los mecanismos que ofrece el SO, al igual que los procesos. Lo que no comparten entre si los hilos es el registro de la CPU, el contador de programa, la pila de variables locales y su estado.

Entre los estados por los que puede pasar un hilo encontramos: Ejecución, preparado y bloqueado. Aunque el acceso a memoria diferencia a los procesos de los hilos, estos tienen grandes similitudes. Como los procesos, los hilos pueden estar en varios estados, ya sea en listo, bloqueado, ejecución o terminado; también, al igual que los procesos, los hilos comparten la CPU y solo uno puede estar en ejecución en un instante dado; cada hilo tiene su propia pila y contador de programa y se ejecuta de forma secuencial; además, cada hilo,

al igual que los procesos, puede tener hilos hijos ejecutándose.

Son llamados **procesos ligeros** por su agilidad o rapidez para ser creados, terminados y cambiados de contexto en el gestor de colas del SO, además, su memoria compartida hace mas rápida la comunicación entre hilos ya que no necesitan invocar el núcleo del SO para comunicarse.

Cabe mencionar que cada SO implementa los hilos de forma diferente, cabe así destacar Win32[51] y Portable Operating System Interface UNIX (POSIX)[52] (pthreads)[53]

POSIX define la llamada *fork* para la creación de procesos y la llamada *pthread_create* para la creación de hilos y aunque en el siguiente apartado se entrara en más detalles sobre dichas funciones,en el cuadro 3.5 podemos observar los resultados (medidos en segundos) de ejecutar la llamada *fork* y *pthread_create* 50.000 veces. Como podemos observar, se demuestra que la creación de hilos es más eficiente que la creación de procesos.

Platform	**fork()**			**pthread_create()**		
	real	**user**	**sys**	**real**	**user**	**sys**
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6 4	7.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Cuadro 3.5: Tiempo de creación de 50.000 procesos e hilos. `https://computing.llnl.gov/tutorials/pthreads`

Otra demostración de la eficacia de los hilos, viene reflejada en el tiempo de acceso a la memoria, en el cuadro 3.6 puede observarse la velocidad de transferencia entre dos procesos utilizando Interfaz de Paso de Mensajes (MPI)[54] y dos hilos. Cabe mencionar que MPI implementa la comunicación a partir de una memoria compartida entre los dos procesos, por lo que siempre requiere una operación de copia.

A nivel de comunicación también aparecen una serie de conceptos a mencionar. Como hemos visto, existen mecanismos de comunicación que nos otorga el SO, pero dado que

[51] `https://es.wikipedia.org/wiki/API_de_Windows`

[52] `https://es.wikipedia.org/wiki/POSIX`

[53] `https://en.wikipedia.org/wiki/POSIX_Threads`

[54] `https://es.wikipedia.org/wiki/Interfaz_de_Paso_de_Mensajes`

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium	2 1.8	6.4

Cuadro 3.6: Velocidad de lectura entre dos procesos y dos hilos. `https://computing.llnl.gov/tutorials/pthreads`

el SO puede desalojar un proceso sin necesidad de informar al programa, pueden darse el caso de que dos o más procesos o hilos estén accediendo a un mismo recurso de forma concurrente.

Podría darse el caso de que al acceder a los recursos compartidos la memoria fuera modificada y se trabajara con datos erróneos, pero existen una serie de mecanismos que permiten bloquear el acceso a un recurso compartido, aunque estos accesos bloqueantes pueden causar otros problemas. Un conjunto de procesos puede quedarse eternamente bloqueados esperando el acceso a un recurso. Este estado es denominado como **deadlock** o **interbloqueo**[55].

El termino de **inanición**[56] define el estado en el que queda un proceso al que siempre se le deniega el acceso a un recurso. No es sinónimo de deadlock, aunque un interbloqueo genera inanición, pero cabe la posibilidad de que la inanición termine por sí sola, mientras que un interbloqueo no. Edsger Dijkstra definió un claro ejemplo de inanición en su cena de los filosos[57], donde estos compartían los cubiertos para cenar y entraban en inanición cuando todos cogían un cubierto, quedando así, «eternamente», esperando el acceso a su segundo cubierto.

Las soluciones para evitar estos interbloqueos también pueden acarrear otros problemas, como el denominado *livelock* que define el estado en el que quedan dos o más procesos al cambiar de estado continuamente en respuesta a cambios de otros procesos [Sch89]

Existen distintas técnicas o mecanismos para evitar estos problemas o causarlos. A bajo nivel disponemos de los **semáforos**, que, como los semáforos que se usan para los pasos de peatones, bloquean el acceso a zonas de memoria compartida y hacen que los procesos esperen en cola. A un nivel un poco mas superior, existen los mecanismos de **envió o paso de mensajes** que permite enviar mensajes a destinatarios concretos, estos mecanismos son

[55] `https://es.wikipedia.org/wiki/Bloqueo_mutuo`

[56] `https://es.wikipedia.org/wiki/Inanición_(informática)`

[57] `https://es.wikipedia.org/wiki/Problema_de_la_cena_de_los_filósofos`

idóneos para sincronización de procesos. También existen mecanismos a un nivel más alto, como los **monitores**, que proveen a los datos de cierta lógica que asegura el control del acceso. Otro mecanismo de sincronización es el rendezvous[58] donde dos procesos se sincronizan para leer y escribir sobre el mismo recurso. En el siguiente apartado se detallan más dichas tecnicas.

Para desarrollar todos estos mecanismos y escribir un programa concurrente es posible utilizar un lenguaje como **ADA**[59], siendo este un lenguaje que provee estructuras embebidas para programacion concurrente y que aporta herramientas orientadas al diseño de software de seguridad critica. Otro lenguaje de programacion es **Occam**, este lenguaje permite utilizar los hilos de una forma más amigable ademas de otorgar soporte para ambientes multiprocesado. Pero tambien es posible utilizar estos mecanismos con un lenguaje secuencial, como C, utilizando los mecanismos que otorga, por ejemplo, un SO multiprocesador como UNIX.

Aunque la programación concurrente tiene más términos y conceptos, estos son los más comunes y utilizados.

3.2.3 Mecanismos de sincronización

Los **semáforos**[60] fueron creados por Edsger Dijkstra en 1965 [Dij65a] y se usaron por primera vez en el sistema operativo THEOS[61].

Es una variable especial o tipo abstracto de datos usado para controlar el acceso a recursos compartidos en sistemas concurrentes.

Los semáforos inicializa un contador de acceso según el valor entero recibido en el constructor.

Fueron definidos por Dijkstra con dos operaciones, **P** o comúnmente llamada *wait* y **V** o *signal*. Dichos nombres tienen origen en el idioma holandes, *Verhogen* significa incrementar y *Proberen* probar, aunque la palabra que uso Dijkstra fue *prolaag* que viene de *probeer te verlagen* que significa *intentar decrementar*.

Dichas operaciones incrementan o decrementan el contador de acceso permitiendo al hilo o proceso entrar en la zona sensible. Se define como semáforo binario aquel que ha sido inicializado su contador a uno y se utiliza para acceder a un solo recurso. Los semáforos aseguran que la acción será realizada de forma atómica, dado que el acceso a la zona sensible está restringido al hilo o proceso que adquiere el semáforo. El SO permitirá el acceso a la zona sensible mientras el contador de acceso se mantenga por encima de cero, una vez alcanzado el valor de cero, el SO pondrá a los hilos en espera hasta que se libere un recurso.

Las utilidades de los semáforos pueden ser varias:

[58] https://es.wikipedia.org/wiki/Primitiva_de_sincronización_rendezvous
[59] http://www.adaic.org/
[60] https://es.wikipedia.org/wiki/Semáforo_(informática)
[61] https://es.wikipedia.org/wiki/THEOS

- Implementar cierres de exclusión mutua o locks.
- Barreras
- Limitar el número de hilos que pueden acceder a un recurso.
- Sincronización. Inicializar un recurso mediante un semáforo inicializado a cero.

El paso de mensajes esta en un nivel intermedio, son un paradigma de la programación siendo uno de los conceptos claves en los modelos de programación concurrente, distribuida y OO.

Un **mensaje** es un fragmento de información que un emisor envía a un receptor. El **paso de mensajes** puede ser síncrono o asíncrono. El modelo de **mensajes síncronos** es más sencillo de programar, dado que solo requiere que un hilo o proceso envié el mensaje y que otro hilo reciba el mensaje. Este sistema es bloqueante. En MPI existen las funciones *s_-send* para enviar el mensaje y *receive* para recibirlo. Aunque es un sistema más sencillo, tiene ciertas carencias, como el tiempo de espera entre que el emisor envía el mensaje y el receptor empieza su lectura, problema que podría añadir latencia a la aplicación, también el sistema requiere que emisor y receptor se reúnan en un punto pudiendo dejar a una de las dos partes bloqueadas en caso de fallo.

En los **mensajes asíncronos** no se requiere esa citación por parte de los interesados, las llamadas no bloquean los hilos por lo que pueden seguir ejecutándose mientras envían y leen. Este sistema añade un problema de seguridad en los mensajes dado que es posible que el contenido del mensaje quede obsoleto si el emisor sigue haciendo operaciones con las variables, para esto existen los envíos asíncronos seguros e inseguros. Los métodos seguros pasan por hacer una copia de los datos temporal en un lugar seguro e inalterable para su posterior envió. La función en MPI es *send*. El envió inseguro se realiza con la función *i_-send* y permite al hilo continuar la ejecución sin esperar a que se haya enviado el mensaje. En este último caso, es responsabilidad del programador controlar que no se modifiquen los datos. MPI añade las funciones *wait_send* y *wait_recv* que bloquean al emisor hasta que los datos han sido enviados o recibidos, también existen las funciones *test_send* y *test_recv* que recibe un valor booleano con el estado del envió o recepción.

Los **monitores**[62] fueron definidos por Charles Antony Richard Hoare en 1974 [Hoa74].

Los **monitores** son un mecanismo de **alto nivel**, pensados para usarse en entornos multi-hilo o muti-proceso. Están compuestos por cuatro componentes: inicialización, como su nombre indica, es el código que se ejecuta cuando el monitor se crea; datos privados, las llamadas privadas del monitor; métodos del monitor, los métodos públicos del monitor; y cola de entrada, mantiene un buffer con los hilos que han llamado a algún método pero no han recibido respuesta.

[62] `https://es.wikipedia.org/wiki/Monitor_(concurrencia)`

Internamente, los monitores, utilizan mecanismos de sincronización y de bloqueo, por eso se habla de que son mecanismos de alto nivel, dado que, una vez creados, abstraen al desarrollador de la funcionalidad interna del monitor. Los monitores tienen dos funciones internas para operar con las variables de condición:

- ***cond_wait(c)*:** el hilo o proceso que ejecuta la llamada se queda en espera hasta que se realice la condicion c. El monitor adquire el bloqueo y da el control al siguiente hilo.
- ***cond_signal(c)*:** ejecuta un hilo que este en espera mediante *cond_wait*. El monitor solo activa un hilo. Existen dos tipos de monitores: Hoare y Mesa.

Los monitores de tipo **Hoare** el hilo que ejecuta el *cond_signal(c)* cede el monitor al hilo que está esperando (*cond_wait(c)*). El monitor adquiere el cerrojo y se lo pasa al hilo durmiente. El control volverá al hilo inicial una vez el hilo despertado realice su tarea y deje el monitor libre. Es posible que las condiciones cambien antes de que el hilo durmiente sea despertado.

En el modelo **Mesa** es bastante más lioso, dado que no cede el monitor y controla que el acceso sea siempre con la condición correcta. Estos modelos son menos propensos a errores dado que un hilo podría ejecutar la llamada *cond_signal(c)* sin afectar al hilo en espera. Este tipo de modelos también incluye la llamada *cond_broadcast* que pone en estado listo los hilos que estaban esperando con *cond_wait(c)*.

3.2.4 Programación multi-hilo

Ya se ha hablado del concepto de hilo, pero en este apartado nos centraremos más en cómo aprovechar dichos hilos para realizar una programación eficiente y de alto rendimiento a partir de los múltiples hilos.

Actualmente, la mayoría de las CPU, tiene capacidad de multi-hilo o multithreading y disponen de soporte HW para ejecutar dichos hilos de forma eficiente.

Aunque en el inicio se le dio más importancia al paralelismo a nivel de instrucción, la programación multi-hilo ha ido cogiendo fuerza a lo largo de los años y esta va muy ligada a la computación de alto rendimiento.

Esta relación es debido a que el tiempo de ejecución de un programa puede verse reducido drásticamente gracias a un buen diseño concurrente. Si se localizan las tareas que pueden realizarse de forma paralela y se separan en distintos hilos en vez de ejecutarse de forma secuencial, el tiempo de ejecución de un programa se verá reducido en relación al número de hilos que se solapen.

Como ya se ha explicado con anterioridad, los hilos son más eficientes que los procesos, comparten el espacio de memoria y el tiempo que necesitan para cambiar de estado es inferior. Pero esta memoria compartida también puede ser un punto negativo para la computación de alto rendimiento dado que los conflictos de lectura que se pueden generar merman

el rendimiento del programa.

Cada SO implementa dichos hilos de distinta forma, como ya hemos mencionado existen distintos estándares para los SO: Win32 y POSIX (pthreads), pero en este apartado vamos a entrar más en detalle.

Los SO, por lo general, implementan dos tipos de hilos, que se diferencian en la capacidad de control que tiene, el hilo, sobre sí mismo. Por un lado tenemos los **apropiativos**, en este tipo de hilos es el SO el que se encarga de decidir cuándo realizar un cambio de contexto. Y en el otro lado tenemos los **cooperativos**, estos hilos deben decidir cuándo abandonar el control, lo que puede acarrear problemas si el hilo se queda esperando un recurso.

En los SO basados en **Windows** disponemos de la Application Programming Interface (API), el número que le precede depende de la versión de Windows y su arquitectura, así un Windows XP de 32bits dispondrá de una API Win32 y un Windows 10 de 64bits dispondrá de una API Win64. Hay que mencionar que también existe una versión de 16bits, pero esta versión no da soporte a la programación multi-hilos. En Windows, cada hilo es propietario de un registro privado donde almacena su estado, y este es capaz de *resumir* su tarea, una vez el SO lo trae de vuelta a ejecución. Este registro es controlado internamente por el SO.

En **Windows** diferenciamos **dos tipos de hilos**: **Los hilos de interfaz de usuario** y **los hilos obreros**. Dado que Windows es un SO basado en ventanas, existe la necesidad de crear un hilo para controlarlas, así nacen los **hilos de UI**. Este tipo de hilo crea su propia ventana y tiene la capacidad de recibir mensajes de ventana. La versión de Win16, recordemos que no tenia soporte para multi-hilo, incluía un sistema de exclusión mutua (Mutex) para este tipo de ventanas, lo que hacía que varias ventanas de Windows95 no pudiesen acceder al núcleo de usuario ni al Graphics Device Interface (GDI)[63] de 16bits, lo que lo hacía más lento respecto a Windows más modernos.

Los **hilos obreros** no crean ventanas, por lo que, obviamente, no pueden recibir mensajes de ventana, por lo que quedan relegados a ser usados como *mano de obra* y realizar tareas en segundo plano.

A diferencia de otros SO, en Windows, tanto los hilos, como los procesos, son eliminados al eliminar el proceso o hilo que los creo.

Existen una serie de funciones que otorga el SO para la creación de procesos e hilos. En los sistemas basados en Windows, disponemos de las siguientes llamadas:

- ***System()*:** esta llamada es utilizada para ejecutar un comando.
- ***Exec()*:** cambia el proceso padre y lo sustituye por el nuevo proceso a ejecutar. Una vez finalizado no vuelve al programa inicial.
- ***Spawn()*:** llamada similar a exec.

[63] `https://msdn.microsoft.com/es-es/library/windows/desktop/dd145203.aspx`

- ***CreateProcess()*:** como dice su nombre, crea un proceso nuevo y ejecuta un comando concurrentemente al proceso padre. Ambos procesos pueden compartir memoria.
- ***CreateThread()*:** crea un hilo independiente y como tal, comparte el mismo espacio de direcciones del proceso padre, por lo que comparten la memoria y el código del programa.

La API de Windows no solo otorga mecanismos para crear procesos e hilos, también aporta métodos de comunicación entre ellos.

Existen tres métodos de comunicación:

- **Variables globales:** dado que los hilos comparten el mismo espacio de memoria es lógico, aunque no fiable, utilizar variables globales para compartir información. Decimos que no es un método fiable, dado que, puede darse el caso de que Windows quite el control a un proceso en medio de una actualización de datos, por lo que los otros procesos podrían obtener datos erróneos. Para este problema existen los mecanismos de bloqueo.
- **Mensajes de Windows:** anteriormente, al hablar de los hilos de interfaz de usuario, hemos hablado de mensaje de ventana, estos mensajes son unos mensajes específicos que otorga Windows para que las interfaces se comuniquen. Este método no puede ser usado si el receptor no es un hilo de interfaz de usuario.
- **Eventos:** estos eventos pueden ser utilizados para despertar hilos durmientes. Estos hilos durmientes no reciben porción de la CPU a no ser que reciban un evento que los despierte.

En los SO basados en **POSIX** podemos encontrar los POSIX threads o comúnmente llamados pthreads. POSIX thredas es una API definida en el estándar POSIX.1c [fITa][64] que pertenece a la biblioteca libpthread.a3 y otorga a los sistemas POSIX las funciones necesarias para el uso de los hilos. Dicha API esta implementada en la mayoría de los sistemas UNIX o similares, como FreeBSD, NetBSD, OpenBSD, Linux, Mac OS X, Android y Solaris, tambien existe una implementaciones para DR-DOS y Windows como parte del subsistema Windows Services for UNIX Windows Services for UNIX (SFU) que provee la implementación de cierto número de APIs POSIX, también se puede encontrar en paquetes externos como pthreads-w32[65] que implementa los hilos por encima de los Win32.

La biblioteca pthreads define una serie de tipos, funciones y constantes en lenguaje C, define con el prefijo *pthread_* todas las llamadas que otorga y dispone de alrededor de 100 funciones que se engloban en cuatro grupos:

[64] http://standards.ieee.org/findstds/standard/1003.1-2001.html
[65] http://www.sourceware.org/pthreads-win32/conformance.html

- **Manejo de hilos:** llamadas que manejan los hilos directamente, creación, unión, eliminación, etc...
- **Sincronización entre hilos:** funciones para la lectura y escritura de bloqueos y barreras.
- **Variables condicionales:** este grupo incluye funciones de creación, destrucción y espera además de señales.
- **Bloqueadores o mutex:** llamadas relacionadas con la sincronización. Este grupo otorga llamadas para crear, destruir, bloquear y desbloquear secciones críticas del código.

La API de POSIX semaphore, aunque trabaja con pthreads, no es parte del estándar de hilos. Esta API fue definida en POSIX.1b [fITb] Las llamadas a la API de POSIX semaphore tienen el prefijo *sem_* en vez del ya conocido, *pthread_*.

POSIX dispone de la llamada *exec()*, que al igual que los sistemas basados en Windows, sustituye al proceso padre con un proceso nuevo y también de una llamada *fork()* que crea un proceso hijo independiente del padre, similar a *createProcess()* de Win32. Como se puede intuir, la llamada *pthrad_create()* crea un hilo de ejecución, dicha función devuelve el identificador del hijo para su control.

Entre los sistemas de sincronización que nos otorga POSIX podemos encontrar el *pthread_join()*. Esta función, bloquea el hilo padre hasta que el hilo hijo ejecuta la llamada *pthread_exit()* (véase figura 3.17). Es posible pasar el estado de finalización al hilo padre a través de dicha llamada.

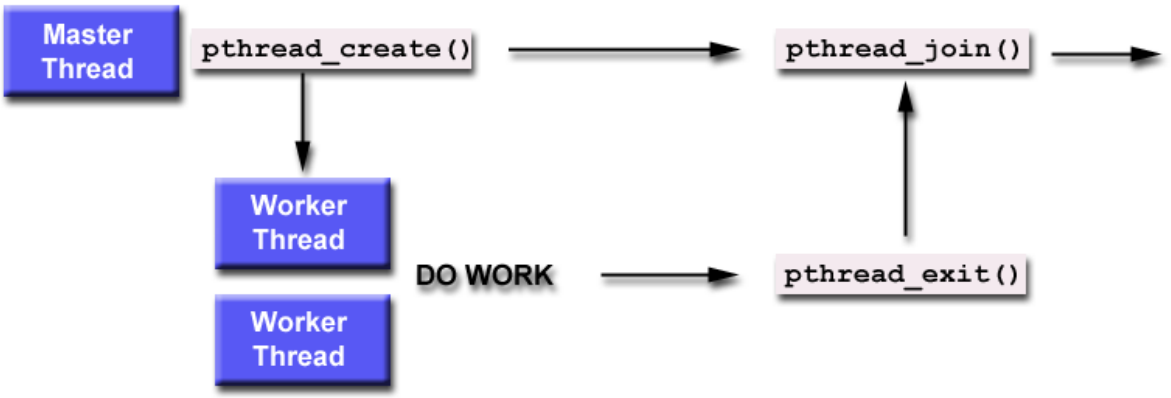

Figura 3.17: Diagrama de espera entre un hilo padre e hijo. https://computing.llnl.gov/tutorials/pthreads/

Es posible evitar que el proceso padre espere al hijo si el hijo se separa mediante la función *pthread_detach()*.

Pthread incluye diversas funciones para el control de los hilos, y aunque en este documento no vamos a mencionar todas, las aquí explicadas son las más importantes para el control de los hilos.

Cabe mencionar, que aunque estas dos APIs, Win32 y pthreads, han sido escritas en lenguaje C es posible, como veremos a continuación, utilizar dichas llamadas al sistema, con otros lenguajes.

3.2.4.1. Python

El lenguaje de programación **Python** fue creado por Guido van Rossum[66] en el Centrum Wiskunde & Informatica (CWI) de los Países Bajos a finales de los años 80[67].

Es un lenguaje de programación interpretado, su filosofía hace hincapié en una sintaxis que favorezca un código legible.

Python soporta Orientación a Objetos (OO), programación imperativa e incluso programación funcional, por lo que está clasificado como lenguaje multiparadigma. Es un lenguaje de tipado dinámico, interpretado y multiplataforma.

Incluye el módulo *thread* que otorga a *Python* primitivas de bajo nivel para trabajar con múltiples hilos, además también añade mecanismos para sincronización. El módulo *threading* añade una manera más fácil y sencilla de manejar los hilos a alto nivel. Este módulo está desarrollado por encima del módulo *thread*. Es un módulo opcional de *Python* y está soportado tanto en sistemas basados en Wndows, Linux, SGI IRIX, Solaris 2.x y en todos los sistemas que dispongan de una implementación de POSIX thread.

En *Python* 3 el módulo *thread* ha sido renombrado a *_thread*. Los hilos en *Python* están controlados por el Global Interpreter Lock (GIL)[68] por lo que, aunque exista más de un procesador, solo puede ejecutarse un hilo a la vez, por lo que, dependiendo de la situación, pueda interesar más utilizar procesos que hilos, dado que estos no sufren de dicha limitación[69].

El desalojo de estos hilos se realiza cada cierto número de instrucciones de bytecode. Este número está fijado, por defecto, a 10, aunque puede ser modificado a partir de la funcion *sys.setcheckinterval*. Dicho desalojo también se realiza cuando el hilo realiza una operación de E/S o se pone el hilo a dormir mediante *time.sleep*.

Una manera de optimizar el tiempo de ejecución y el número de cambios de contexto es lanzar el programa con el flag -O para optimizar el número de instrucciones a generar, a menos instrucciones, menos cambios de contexto.

Python permite la programación OO por lo que dentro el módulo *threading* podremos encontrar la clase *Thread*. Si nuestra intención es crear un hilo en *Python* deberemos crear una clase que herede de *Thread* y que, si necesitamos nuestro propio constructor, llame a *threading.Thread.__init__(self)* para inicializar el objeto de manera correcta, ademas dicha clase debe contener un método llamado *run* que será ejecutado una vez se haya creado el

[66] https://es.wikipedia.org/wiki/Guido_van_Rossum

[67] http://www.artima.com/intv/pythonP.html

[68] https://wiki.python.org/moin/GlobalInterpreterLock

[69] http://mundogeek.net/archivos/2008/04/18/threads-en-python/

objeto correspondiente e inicializado mediante la llamada *start()* (véase Listado 3.2).

```
1  import threading
2  class MyThread (threading.Thread):
3        def __init__(self, id):
4            threading.Thread.__init__(self)
5            self.threadID = id
6        def run(self):
7            print "Thread: ", self.threadID
8  print "Parent thread."
9  for i in range(0, 10):
10     t = MyThread(i)
11     t.start()
12     t.join()
```

Listado 3.2: *Threading example* en Python

Como hemos visto en el Listado 3.2, los hilos de *Python* permiten esperar la finalización de la ejecución de un hilo hijo mediante la llamada *join()*, esto en algunas plataformas es necesario, dado que es posible que una vez finalizado el proceso padre, los hilos hijos sean eliminados. Otro método, es crear una instancia de la clase *Thread* y pasarle como parámetro una función a ejecutar además de argumentos necesarios (véase Listado 3.3).

```
1  import threading
2  def imprime(num):
3      print "Thread: ", self.threadID
4  print "Parent thread."
5  for i in range(0, 10):
6      t = threading.Thread(target=imprime, args=(i, ))
7      t.start()
```

Listado 3.3: *Threading function example* en Python

Es posible identificar los hilos mediante el parámetro *name*, aunque por defecto, *Python*, asigna nombres a cada hilo, también puede recibir un parámetro de tipo booleano llamado *verbore* que indicara al módulo que imprima mensajes en modo depuración. El constructor también puede recibir un parámetro llamado *group*, que actualmente no tiene utilidad, pero en un futuro se utilizara para agrupar hilos.

Para una mejor depuración, es posible utilizar el módulo *logging* que incluye en sus mensajes el nombre del hilo en ejecución. El módulo incluye una serie de llamadas que otorgan información sobre los hilos. El método *isAlive* nos devuelve el estado del hilo. También, la función *threading.enumerate* nos devuelve una lista de los hilos en ejecución creados por nuestro programa.

Es posible crear nuestros hilos en modo **Daemon**[70] lo que permite al hilo principal no quedarse bloqueado(véase Listado 3.4).

```
1  import threading
2  import logging
3  import time
4  logging.basicConfig( level=logging.DEBUG,
5      format='[%(levelname)s] - %(threadName)-10s : %(message)s')
6  def daemon():
7      logging.debug('Start')
8      time.sleep(2)
9      logging.debug('Exit')
10 d = threading.Thread(target=daemon, name='Daemon')
11 d.setDaemon(True)
12 d.start()
```

Listado 3.4: *Threading daemon example* en Python

Otra clase incluida en el módulo *threading* es *Timer* que hereda de *Thread* y que lanza automáticamente el hilo una vez ha pasado el tiempo de espera definido en su constructor. Es posible cancelar esta ejecución con la llamada *cancel*.

El módulo *threading* también añade métodos de sincronización. Entre los métodos que implementa podemos encontrar locks, semáforos, condiciones y eventos.

La clase *Lock* incluida en el módulo *threading* permite crear un **candado** y utilizarlo para bloquear el acceso a zonas sensibles del código. Los *Lock* funcionan como puertas con un cerrojo y una sola llave, una vez la llave ha sido adquirida por un hilo, la puerta no puede ser abierta, por otro hilo, hasta que el hilo libere la llave. En *Python*, la llave se adquiere con la función *acquire* del cerrojo *Lock* y se libera con la función *release*. La función *acquire* puede ser bloqueante o no, dependiendo del parámetro booleano que se le pase, en caso de no querer bloquear el hilo, la llamada *acquire* devolverá *True* o *False* dependiendo de si se ha adquirido o no la llave (véase Listado 3.5).

Python también incluye otro tipo de candados llamados **semáforos** que utilizan la clase correspondiente *Semaphore*. Los semáforos, a diferencia de los cerrojos, pueden ser abiertos con varias llaves. La función *Semaphore* recibe el número de llaves que quiere poner a disposición de los hilos y, como en los cerrojos, las reparte a los hilos con la función *acquire* restando uno al número de llaves disponibles y son los hilos los que devuelven las llaves con la función *release* añadiendo uno al número de llaves disponibles. Es cuando el contador interno del semáforo llega a cero, cuando el hilo se bloquea a la espera de una llave al llamar a *acquire*.

[70] https://www.genbetadev.com/python/multiprocesamiento-en-python-threads-a-fondo-introduccion

```
lista = []
lock = threading.Lock()
def anyadir(obj):
    lock.acquire()
    lista.append(obj)
    lock.release()
def obtener():
    lock.acquire()
    obj = lista.pop()
    lock.release()
return obj
```

Listado 3.5: *Threading lock example* en Python

Es posible que, por un mal diseño de programación, se liberen más llaves de las que correspondan y el semáforo supere el contador inicial, esto puede ser evitado con el tipo de semáforo *BoundedSemaphore*.

Las **condiciones** están disponibles en la clase *Condition*, pueden recibir un *Lock* o crear uno automáticamente y son utilizadas para notificar a los hilos que ciertas acciones se han realizado. Supongamos que un número indefinido de comensales espera a que suena la campanilla que indica que la cena este servida y mientras tanto, la cocinera termina la cena, la sirve y notifica a los asistentes que la cena está servida mediante dicha campanilla. En este ejemplo, todos los comensales han adquirido el condicional (*Condition*) y han quedado a la espera (*wait*) de que suene la campanilla, posteriormente la cocinera lo notifica (*notifyall*) a los comensales utilizando la campanilla. Es posible notificar la condición a un solo hilo mediante la llamada *notify*, pero solo el primer hilo que responda a la llamada podrá acceder a la zona conflictiva (véase Listado 3.6).

```
lista = []
cond = threading.Condition()
def consumir():
    cond.acquire()
    cond.wait()
    obj = lista.pop()
    cond.release()
    return obj
def producir(obj):
    cond.acquire()
    lista.append(obj)
    cond.notify()
cond.release()
```

Listado 3.6: *Threading condition example* en Python

Las condiciones son la base de los **eventos**. Los eventos en *Python* o *Event* abstraen al programador del uso del *Lock* y de las funciones *acquire* o *release*. Como en el caso anterior, los comensales quedan a la espera (*wait*) hasta que la cocinera lanza el evento con la función

set.

Estos mecanismos permiten al programador controlar el acceso a zonas sensibles, pero si se necesita compartir información entre hilos es posible utilizar la clase *Queue* que nos otorga *Python*. *Queue* esta implementada como una **cola First In First Out (FIFO)** con soporte multi-hilo. Se trata de un canal en el cual los hilos pueden dejar mensajes mediante la función *put* y leerlos mediante *get*. No existe una forma directa de reconocer los mensajes a no ser que se lean con anterioridad. La lectura y escritura de los mensajes puede ser bloqueante, obligando al hilo a esperar hasta que su mensaje sea leído o hasta que reciba un mensaje. Como pasaba con los hilos, se puede realizar una espera sobre una cola con la función *join*, dicho hilo se quedara bloqueado hasta que otro hilo ejecute la llamada *task_done*.

Aunque no se han explicado al detalle todas las opciones, con estas funciones, un programador novel podría desarrollar una aplicación en *Python* capaz de sincronizar y manejar hilos concurrentes de forma eficiente y sin bloqueos.

3.3 Toma de decisiones

3.3.1 Introducción

Visto desde el punto psicológico, la toma de decisiones es: «uno de los procesos más difíciles a los que se enfrenta el ser humano»[71]. Cada individuo toma decisiones de forma diferente y basándose en su experiencia. El mundo de los sistemas multi-agentes no es diferente y más en un sistema donde los agentes no tienen por qué compartir sus metas. Aunque, hablamos de un **acuerdo de beneficio mutuo** cuando varios agentes pueden llegar a cumplir su meta, parcialmente o en su totalidad, de forma simultánea.

La habilidad de llegar a un acuerdo sin terceros es una capacidad que tienen que tener los agentes autónomos inteligentes. Sin esta capacidad, no serian capaces de trabajar en sociedad. La capacidad que tenga un agente para **negociar** y **argumentar** definirá su capacidad para llegar a un acuerdo con otro agente [Woo01b] [Wei06].

Los **mecanismos** o **protocolos** utilizados entre dos agentes en las negociaciones, fueron definidos, en lo que comúnmente se llama las **reglas del encuentro** o *rules of encounter*, por Rosenschein y Zlotkin en 1994 [RN03]. Siguiendo estas reglas, es posible diseñar protocolos que cubran todas las necesidades de la negociación.

El **diseño de mecánicas** o **mechanism design** se basa en diseñar protocolos orientados a gobernar las interacciones de un sistema multi-agente. El objetivo en el diseño de sistemas de comunicación «convencionales »es evitar *deadlocks*, *livelocks*, etc... [Hol91], pero en el diseño de protocolos, el objetivo es distinto. Sandholm[72] definió en 1999 las siguientes cualidades que tienen que tener un protocolo de comunicación:

[71] `http://www.cop.es/colegiados/m-00451/tomadeciones.htm`

[72] `http://www.cs.cmu.edu/~sandholm/`

- **Garantizar el éxito:** el protocolo debe garantizar que se llegara a un acuerdo final.
- **Maximizar el bienestar social:** Intuitivamente, el protocolo debe maximizar el bienestar social maximizando la suma de las utilidades de los participantes.
- **Eficiencia de pareto:** la negociación tendrá eficiencia de pareto si no existe otro resultado que haga que un agente tenga un resultado mejor sin hacer que otro agente obtenga un resultado peor. En las negociaciones que no tienen pareto, es posible que un agente obtenga el mejor resultado y los otros participantes resultados un poco peores que el mejor.
- **Racionalidad individual:** se dice que el protocolo tiene racionalidad individual si sigue el protocolo en favor del interés de los participantes.
- **Estabilidad:** el protocolo es estable si proporciona a todos los agentes un incentivo para comportarse de una manera particular
- **Simplicidad:** es aquel que hace la estrategia apropiada para el participante.
- **Distribucion:** un buen protocolo minimiza las comunicaciones entre agentes.

Otro tema relacionado es, dado un protocolo en particular, como se puede diseñar una **estrategia** en particular para ser utilizara por los agentes mientras negocian. Un agente se centrara en utilizar la estrategia que le otorgue mayor beneficio. La cuestión es, que puestos a crear agentes que sean capaces de negociar, nos interesa que su beneficio sea el mayor, no solo en teoría, sino también en la práctica.

3.3.2 Subastas

Con la aparición de Internet y la World Wide Web (WWW)[73] el mundo de las subastas ha ampliado su público. Grandes empresas, como *eBay*, han basado su negocio en las subastas on-line. En el mundo de los agentes, las subastas son un método de negociación. Los agentes, pueden tomar el rol de **subastador**, o **auctioneer**, que es quien tiene la intención de vender un **bien/producto/servicio**, y de **postor** o **bidder** que es quien compra un *servicio*. La meta de la subasta es que el *subastador* llegue a un acuerdo con uno de los *postores*. El *subastador* buscara optimizar al máximo la ganancia y el *postor* buscara minimizar el precio al que compra. La teoría de subastas analiza los protocolos y las *estrategias* de los agentes en las subastas [Wei06]. Existen distintos factores que pueden influir en que estrategia y protocolo utiliza el agente. Existen tres configuraciones dependiendo del valor que le otorga el agente al bien. Si el valor del precio es conocido por todos, se dice que es un una subasta de **valor publicó** o común para todos los agentes. En caso de que el *bien* tenga alguna característica que lo haga más valioso para un agente entonces hablamos de **valor privado**. Existe un tercer caso donde el valor del *bien* varía según el valor que el agente le otorga y el que le otorgan los demás agentes, este tipo se llama **valor correlacionado**. Otro punto importante es como

[73] https://es.wikipedia.org/wiki/World_Wide_Web

se selecciona el ganador y el pago a realizar. El primero es obvio, en este tipo de subastas, el subastador busca al mayor postor, pero el precio a pagar depende del tipo de subasta. Si el postor paga el precio pujado entonces se denomina como **primer-precio**. Existe otro tipo de subasta, las de **segundo-precio**, donde el postor paga el precio de la segunda mayor puja.

La privacidad de las pujas también es un factor a tener en cuenta. En el caso donde las pujas son públicas y todo el mundo puede conocerlas es decir, existe un *conocimiento común* son denominadas como **llanto abierto** o **open cry** y en el caso de ser privadas, donde solo el subastador y el postor las conocen, se denomina **oferta sellada** o **sealed bid**.

Por último, el mecanismo utilizado para realizar las pujas también influye en la decisión de los agentes. El numero de pujas puede variar según el tipo de subasta. El primer mecanismo se conoce como **one shot** y solo permite realizar una puja por postor. El segundo empieza con un valor mínimo, denominado **reservation**, y si los postores están interesados deben superar dicho valor. El valor asciende hasta que no quedan más postores interesados, quedando como ganador la ultima puja aceptada. Una variante de este mecanismo empieza por un valor elevado y va bajando el valor poco a poco hasta encontrar un postor.

Tipos de subasta:

- **Subasta inglesa:** la subasta inglesa es de primer-precio, llanto abierto y ascendente. Cada postor es libre de aumentar su puja. Cuando no quedan postores que quieran aumentar su puja entonces el ultimo postor, con la puja más alta, gana el *bien*. Una estrategia utilizada por los agentes debera tener en cuenta, el valor privado del agente, las estimaciones de otros postores y su historial de pujas. Por ejemplo, en subastas de valor privado, los agentes tienden a sumar una cantidad fija a la puja superior y paran cuando llegan a su límite.
- **Subasta primer-precio oferta-sellada:** en esta subasta cada postor puja una vez sin conocer la puja de los otros postores. No existe una estrategia ganadora en este tipo de pujas, pero por lo general, se suele pujar en relación al valor que le damos al *bien* y el conocimiento previo de otras pujas sobre *bienes* parecidos. La mejor estrategia de un agente es la que le otorgue un valor inferior, pero ganador, en relación a otras pujas. Según la distribución de probabilidad de los valores de los agentes es posible determinar el valor del equilibrio Nash[74].
- **Subasta holandesa:** tipo descendiente. El vendedor baja la puja hasta que uno de los postores adquiere el *bien* a dicho precio. A nivel de estrategia es similar a la subasta de *primer-precio oferta-sellada* dado que solo importa la puja ganadora.
- **Subasta Vickrey:** con *segundo-precio* y *oferta-sellada*. cada postor realiza una sola

[74] *«Un equilibrio de Nash es una situación en la cual todos los jugadores han puesto en práctica, y saben que lo han hecho, una estrategia que maximiza sus ganancias dadas las estrategias de los otros. Consecuentemente, ningún jugador tiene ningún incentivo para modificar individualmente su estrategia.»* `https://es.wikipedia.org/wiki/Equilibrio_de_Nash`

puja y paga el que realice la mayor puja pero a coste de la segunda puja. Una estrategia dominante en este tipo de pujas es pujar su valor real. Es posible que un agente puje **más** del valor que ha estimado y que eso le haga ganar, pero es posible que haya ganado a un precio superior al que estimo. Si un agente puja **menos** de su valor estimado tiene menor probabilidad de ganar, pero en caso de ganar, este siempre habrá pagado menos de su valor estimado.

3.3.3 Negociaciones

Aunque las subastas son herramientas muy útiles para asignar recursos, su uso queda muy reducido dado que solo se limitan a decidir donde localizar los *bienes*. Para un configuración más general donde los agentes necesitan alcanzar un acuerdo sobre asuntos de interés mutuo necesitaremos otro tipo de técnicas. Estas técnicas son denominadas **negociaciones**. En 1994, Rosenschein y Zloktin [RN03], distinguieron entre dos tipos de dominios de negociación: **dominio orientado a tareas** o **Task Oriented Domains (TOD)** y **dominios orientados al valor** o **Worth Oriented Domains (WOD)**[75]

Es necesario definir una serie de conceptos básicos relacionados con las negociaciones [Woo01b]:

- **Un conjunto de negociaciones:** representa el espacio de posibles **propuestas** que los agentes pueden hacer.
- **Un protocolo:** que define las posibles *propuestas* que los agentes pueden hacer, basadas en la historia de la negociación.
- **Una colección de estrategias:** una para cada agente, que determinan que *propuestas* harán los agente. La *estrategias* de un agente es privada, aunque la *estrategias*sea privada, cabe destacar que, la mayoría de las subastas suelen ser de tipo *open cry* lo que hace que las *propuestas* puedan ser vistas por todos los participantes.
- **Las reglas:** que determinan cuándo se ha llegado a un acuerdo y cuál es el acuerdo

Las negociaciones normalmente se producen durante una serie de turnos, con cada agente realizando *propuestas*. Las *propuestas* de dichos agentes están definidas por sus *estrategias* y será sacada del *conjunto de negociaciones* legales que han sido definidas en el *protocolo*. Si se llega a un acuerdo, como definen las reglas, la negociación termina cerrando el trato.

La complejidad de una negociación se incrementa según el número de *issues* o asuntos a tratar. Por ejemplo, una negociación con un asunto simple puede ser una donde las dos partes negocian sobre el precio de un *bien*. En este escenario, la preferencia de los agentes es simétrica dado que lo que para uno es lo mejor, para el otro es lo peor. Estos escenarios simétricos son simples de analizar dado que para que el vendedor llegue a un acuerdo solo

[75] *«In* TOD *a goal specifies a set of tasks that the agent is required to carry out. In contrast, in* WOD *the goal definition is subsumed by a worth function over all possible final states. Those states with the highest value of worth might be thought of as those that satisfy the full goal, while others, with lower worth values, only partially satisfy the goal.»* [ZR99]

tiene que rebajar su precio y para que el comprador llegue al precio solo tiene que subirlo. En una negociación con asuntos múltiples, no solo se negocia en relación a un solo atributo, si no sobre múltiples atributos que están relacionados entre sí. En este tipo de negociaciones es menos obvio obtener una concesión, no es tan simple como aumentar o disminuir los valores.

Los múltiples atributos conducen a un aumento de los posibles acuerdos. Por ejemplo, un escenario donde los agentes estén negociando sobre el valor de n variables *booleanas*. El trato consiste en asignar un valor entre verdadero o falso a dichas variables, entonces los posibles acuerdos del dominio serian 2^n. La mayoría de las negociaciones son más complejas que este ejemplo.

Otro factor que complica las negociaciones es el número de agentes involucrados en el proceso y como estos interactúan. Existen distintas posibilidades:

- **Negociación uno a uno:** cuando un agente negocia con solo un agente. El ejemplo simétrico anterior, donde dos agentes negociaban el precio de un *bien* es un claro ejemplo de negociación uno a uno.
- **Negociación muchos a uno:** en esta configuración, un solo agente negocia con un número indefinido de agentes. Las subastas son un ejemplo de este tipo de negociaciones.
- **Negociación muchos a muchos:** en este tipo de negociaciones, muchos agentes negocian entre sí, simultáneamente. Esto significa, que en un momento dado, en una negociación entre n agentes, existen $n(n-1)/2$ hilos de negociación.

Por estas razones, la mayoría de los intentos de automatizar los procesos de negociación se han centrado en configuraciones simples. Asuntos simples, simétricos y uno a uno.

3.3.3.1. Gestión de compromisos en múltiples negociaciones concurrentes

En el articulo realizado por T. D Nguyen y N.R. Jennings, *Managing commitments in multiple concurrent negotiations* [NJ05b] se describe un modelo de negocio donde un comprador o *buyer* intenta llegar a un acuerdo con múltiples proveedores o *seller*. Gracias a la negociación concurrente con estos *proveedores* y a los acuerdos **parciales** que el *comprador* realiza, el *comprador* puede llegar a un buen acuerdo de manera eficiente. Dado que el *comprador* necesita realizar acuerdos intermedios antes de llegar a un acuerdo final con un agente, es necesario diseñar un modelo flexible que pueda razonar cuando comprometerse con un *vendedor* o cuando anular un acuerdo intermedio.

A continuación se describirá, de manera más detallada, el modelo de negociación descrito en dicho artículo.

El agente que desea contratar el **servicio** es denominado **buyer** y los agentes que otorgan los servicios **sellers**.

El *buyer* dispone de un tiempo máximo $t_{b_{max}}$, una vez concluido dicho tiempo, la negociación termina. De forma similar, cada *seller* α también dispone de su propio tiempo máximo de negociación $t_{\alpha_{max}}$. Cada agente, dispone de un conjunto de estrategias **Strategies (S)** que puede adoptar. Dado que el encuentro está limitado por el tiempo, las estrategias son dependientes de dicho factor. Existen distintas estrategias, pero pueden agruparse en tres tipos: **conceder**, que disminuye el valor rápidamente hasta llegar a su valor de *reservation* o mínimo aceptable. **Linear** que disminuye proporcionalmente al tiempo que queda para finalizar la negocioacion. **Tough** que mantiene el valor inicial hasta momentos antes de finalizar la negociacion, donde disminuye drasticamente hasta su valor de *reservation* [PFJ98]. Cada hilo de negociacion sigue un protocolo de secuencia alterna, donde en cada paso el agente puede aceptar la propuesta del oponente, proponer una contra oferta o finalizar la negociacion. Estos datos, como las propias preferencias de cada agente, son privados.

El modelo del *buyer* está compuesto por tres componentes: Un **coordinador**, un conjunto de *hilos de negociación* y un **gerente de compromiso** o **commitment manager** (véase Figura 3.18). Los *hilos de negociación* tratan directamente con cada uno de los *sellers*, existe un *hilo* por *vendedor* y son los responsables de decidir si se envía una contra oferta. Los *coordinadores* dedican las estrategias de negociación para cada *hilo*. Despues de cada ronda, los *hilos* reportan su estado al *coordinador*. Si un *hilo* llega a un acuerdo con un *seller* en particular, termina la negociación y espera hasta que se llegue al tiempo límite $t_{b_{max}}$. El *coordinador* informa a los demás *hilos de negociación* con el nuevo valor *reservation* y cambia las estrategias si es necesario. El **commitment manager**, maneja cualquier problema relacionado con los compromisos y anulaciones. Interviene cuando el *hilo* necesita decidir cuándo aceptar o no una propuesta de un vendedor o cuando un *seller* decide cancelar un acuerdo. El resultado del **commitment manager**, para por el *coordinador* para comprobarlo con los otros *hilos* antes de volver al *hilo* que lo ha invocado.

El coordinador. El *coordinador* es el responsable de coordinar los *hilos de negocio* y elegir la *estrategia* de negocio apropiada. Antes de empezar la negociación el *coordinador* tiene en cuenta la información que tiene de los *sellers*. Se consideran dos tipos de *sellers*: **conceder**, que están dispuestos a ceder para llegar a un acuerdo, y **non-conceder**, que suelen utilizar estrategias *tough*. Los distintos tipos de *sellers* son denominados A_{types}: types={con, non}. Esta información representa la distribución de probabilidad de que un agente sea de un tipo o de otro, suelen estar basadas en experiencias pasadas, mediante un sistema de referencias o a partir de terceros de confianza. Si no existe información previa, se supone que todos siguen una distribución uniforme.

El *coordinador* tiene en cuenta otros dos datos referentes al *seller*: la Percentage of Success Matrix (PS) o **matriz de porcentaje de éxito**, que almacena el porcentaje de tener un acuerdo utilizando un tipo de estrategia con un *seller* en concreto, y la Pay Off Matrix (PO) o la **matriz de reembolso**, que almacena la media de las ganancias en situaciones simila-

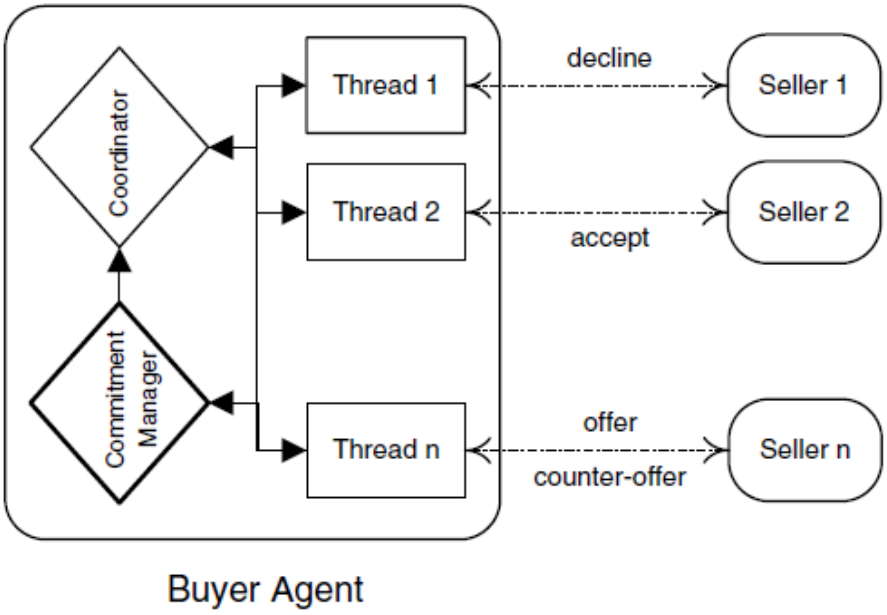

Figura 3.18: Arquitectura del sistema

res.

Con esta información el *coodinador* calcula la probabilidad de que un agente sea de un tipo. Basado en esto, el agente calcula la *utilidad* esperada de aplicar las distintas estrategias y selecciona la que mayor utilidad le otorga. La formula 3.1. calcula la Expected Utility (EU) de una estrategia $\lambda \epsilon S$, donde $P(a)$ es la probabilidad de que un *seller* sea de tipo a y PS y PO son los valores de las matrices correspondientes. Este proceso se repite hasta que el *coordinador* selecciona las estrategias para cada *seller*.

$$EU(\lambda) = \sum_{a \epsilon A_{types}} PS(\lambda, a) PO(\lambda, a) P(a) \tag{3.1}$$

Otra de las tareas del *coordinador* es clasificar los *sellers* durante la negociación. El *buyer* intenta clasificar al *seller* durante la negociación basándose en sus pujas. Asi, cuando se alcanza el momento $t : 2 < t \leq t_{b_{max}}$, denominado **el tiempo analisis**, el *coordinador* intenta determinar si el *seller* es de tipo *conceder* o *non-conceder*. En concreto, asumiendo que $U(\lambda, t')$ es el valor de *utility* de un *seller* λ en el momento $t' : (1 \leq t' \leq t)$ según las preferencias del *buyer*. Entonces, el *seller* λ es considerado un *conceder* si $\forall t' \epsilon \, |3, t| : \frac{U(\lambda,t')-U(\lambda,t'-1)}{U(\lambda,t'-1)-U(\lambda,t'-2)} > \theta$ donde θ es el valor umbral fijado en el comportamiento de tipo *conceder*. Si esta condición falla, el *seller* es considerado de tipo *non-conceder*.

Ahora, dado un conjunto de S y un conjunto de *sellers* clasificados A_s, el *coodinador* cambiara la estrategia de todos los *hilos de negociación* basándose en el tipo del agente con el que creen que están negociando. En concreto, para cada agente $\alpha \epsilon A_s$, el *coordinador*

selecciona una *estrategia* $\lambda \epsilon S$ que otorga el maximo de *utilidad esperada* y lo aplica al correspondiente *hilo* usando la formula 3.2.

$$P(j \epsilon A_{types}) = \begin{cases} 1 & \text{si } \lambda \text{ es de tipo } j \\ 0 & \text{si no} \end{cases} \tag{3.2}$$

Los hilos de negociación. Individualmente, cada *hilo de negociación* es el responsable de tratar con un *seller* en nombre del *buyer*. Cada *hilo* hereda las referencias del *buyer* y obtiene su estrategia del *coordinador*. Cada *hilo* está compuesto por tres subcomponentes (véase Figura 3.19): **comunicaciones**, **procesos** y **estrategias**. El subcomponente de *comunicaciones* es el responsable de las comunicaciones con el *coordinador* y el *commitment manager*. Cada ronda comprueba si tiene mensajes del *coordinador* y los delega al subcomponente de *procesos*. Después de cada ronda, reporta el estado del hilo al *coordinador*. Los *procesos* manejan los mensajes que les envía el subcomponente de *comunicaciones*. Pueden cambiar el valor de *reservation* o la *estrategia*. El subcomponente de *estrategia* es el responsable de realizar las ofertas o contra ofertas y tambien decide cuando aceptar, en cooperación con el *commitment manager*, la oferta realizada por el *seller*.

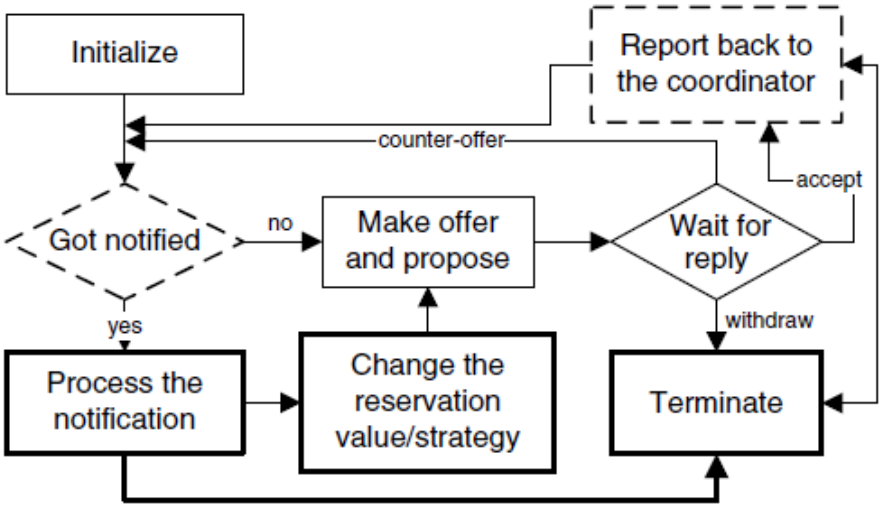

Figura 3.19: Hilo de negociación

El commitment manager. Cada vez que el *buyer* y un *seller* α deciden llegar a un acuerdo intermedio de valor de *utilidad* $U(\alpha, t)$ (según el *buyer*), el acuerdo se liga a los dos agentes. Si alguno de los dos decide cancelar el contrato, tendrá que pagar una **cuota de anulación** o **decommitment fee** (ρ). Dicha cuota se calcula en base a un porcentaje del valor de la *utilidad* del acuerdo. Para el cálculo de dicho cuota se utiliza la formula 3.3, donde t_α es el tiempo en el que se realizo el contrato, ρ_0 es la **penalización inicial**, cuando el tiempo es $t\alpha$,

y $\rho_{max} \geq \rho_0$ es la **penalización final**, donde el tiempo es $t_{b_{max}}$.

$$\rho(t) = U(\alpha, t) \times \left(\rho_0 + \frac{t - t_\alpha}{t_{b_{max} - t_\alpha}} \times (\rho_{max} - \rho_0) \right) \tag{3.3}$$

Considerando el siguiente ejemplo. El valor de $t_{b_{max}}$ de un *buyer* es 10, su *penalización inicial* ρ_0 es de un $5\,\%$ y la *penalizacion final* es del $10\,\%$. Un acuerdo con una *utilidad* de $0{,}58$ se realiza en $t = 6$. En $t = 9$ el *buyer* decide cancelar el contrato, según 3.3, tendrá que pagar:

$\rho(t) = 0{,}58 \times \left(0{,}05 + \frac{9-6}{10-6} \times (0{,}10 - 0{,}05)\right) = 0{,}58 \times 0{,}0875 = 0{,}05075$

Dado que el *buyer* no puede estar pagando una penalización continuamente, tiene que vigilar cuando aceptar los contratos. Para tener esto en cuenta, cuando se presenta un contrato $\phi(\alpha)$ que tiene un valor de *utilidad* $U(\alpha, t)$ de un *seller* α en un tiempo t, el *buyer* solo aceptara, y cancelara el contrato actual si existe, si se dan los siguientes casos:

- Si existe un contrato con otro agente α' en un tiempo $t_{\alpha'}$, la ganancia de *utilidad* tiene que ser superior a la utilidad del actual acuerdo más la penalización a pagar. Esto significa $U(\alpha, t) > U(\alpha', t_{\alpha'}) + \rho(t)$.
- El *grado de aceptación* (μ) del contrato $\phi(\alpha)$ debe superar un límite preestablecido (τ). Este límite define como se comporta el *buyer* si es **codicioso** o **greedy**, si tiende a aceptar cualquier acuerdo positivo, o **paciente** o **patient**, si solo acepta los que le dan una ganancia superior. μ es calculado comparando la *utilidad* del contrato $\phi(\alpha)$ con los valores de *utilidad* futuros de los siguientes contratos de los otros *sellers*. La formula 3.4 define $\rho(t)$ como la *cuota de anulación* que tiene que pagar el *buyer* en el momento t y $U_{exp}(\alpha, t)$ como la utilidad futura del *seller* α_i. El valor de $U_{exp}(\alpha, t)$ se calcula mediante 3.5, donde $d_U(t_1, t_2)$ es la distancia entre dos ofertas del mismo *seller* α_i en el tiempo t_1 y t_2 :$d_U(t_1, t_2) = U(\alpha_i, t_1) - U(\alpha_i, t_2)$

$$\mu(\phi(\alpha)) = \frac{U(\alpha, t) - \rho(t)}{max\left\{U_{exp}(\alpha_i, t) | \alpha_i \epsilon A_s \setminus \alpha\right\}} \tag{3.4}$$

$$U_{exp}(\alpha_i, t) = U(\alpha_i, t) + \frac{d_U(t, t-1))}{d_U(t-1, t-2)} \times |d_U(t, t-1)| \tag{3.5}$$

Hasta ahora, solo se han considerado escenarios donde el *buyer* solo podía tener un contrato activo a la vez. Sin embargo, es posible para el *buyer* mantener más de un acuerdo a la vez, y más tarde, seleccionar, entre los contratos temporalmente aceptados, el de mayor valor de *utilidad*. Este metodo evita el problema de aceptar un acuerdo, con solo un *seller*, y que dicho acuerdo sea cancelado cerca del tiempo límite o $t_{b_{max}}$, dejando al *buyer* con un tiempo mínimo para realizar otro acuerdo.

En algunos casos, es beneficioso para el *buyer* mantener acuerdos simultáneos con distintos *sellers*. Suponiendo que un *buyer* mantiene $\omega \geq 1$ contratos simultáneos y $\Omega = \{C_i | i \epsilon [1, \omega]\}$ el conjunto de contratos que el *buyter* ha aceptado siendo $|\Omega| \leq \omega$. $C_i = \{\phi', t'\}$ es un contrato que consiste en un acuerdo intermedio ϕ' realizado en un tiempo t'. Asumiendo que en el momento t, existe una oferta ϕ del *seller* k que tiene un *grado de aceptacion* $\mu(\phi(k)) > \tau$ (véase formula 3.4) y $U(\phi(k), t) > U(\phi', t_{k'}) + \rho(t) \forall C(\phi', t_{k'}) \epsilon \Omega$ entonces la oferta será aceptada por el *buyer*. Para que una oferta sea aceptada, se deben seguir los siguientes pasos:

- Si Ω no está lleno, es decir $|\Omega| < \omega$, $C(\phi, t)$ se añadira a Ω:$\Omega = \Omega \cap C(\phi, t)$.
- Si Ω está lleno, es decir $|\Omega| = \omega$, entonces se selecciona el $C_i = \{\phi', t'\}$ que tenga el menor valor de *utilidad* $U(\phi', t_{k'})$ y se elimina del conjunto Ω. Entonces, $C(\phi, t)$ se añadira a Ω:$\Omega = \Omega \cap C(\phi, t)$.

Ahora, si un *seller* k tiene un contrato $C(\phi(k), t_k)$ y decide cancelarlo, el contrato sera eliminado de Ω:$\Omega = \Omega \setminus C(\phi(k), t_k)$. Al final del proceso de negociacion, el *buyer*, si existe más de un contrato, seleccionara entre los contratos del conjunto Ω al que mayor *utilidad* aporte y cancelara los restantes.

Capítulo 4

Método de trabajo

En esta sección se detallará comá se ha realizado el trabajo, de forma que el lector llegue a entender como ha sido estructurado, planificado y controlado el desarrollado del proyecto. Se razonará tanto la selección de la metodología escogida para el desarrollo del proyecto y como las herramientas y tecnologías elegidas para llevar a cabo este TFG

4.1 Metodología

Según la RAE, una **metodología** es, *«Conjunto de métodos que se siguen en una investigación científica o en una exposición doctrinal»*[1]. Más en concreto, una **metodología de desarrollo de software** en *ingenieria de software* es *«un marco de trabajo usado para estructurar, planificar y controlar el proceso de desarrollo en sistemas de informacion»*[2]. A lo largo de los años han ido apareciendo distintas metodologías y enfoques. Estas *metodologías de desarrollo de software* están basadas en: una filosofía de desarrollo de programas de computacion y en herramientas, modelos y métodos que asisten al proceso de desarrollo de software.

La mayoría de las metodologías han desarrollado su propio enfoque y otras lo han adaptado de uno ya existente. Entre los enfoques más utilizados podemos encontrar:

- **Modelo en cascada:** es un proceso dividido en fases secuenciales, muy estricto, donde se hace hincapié en la planificación y las fechas. Su ejecución secuencial, fase a fase, hace imposible añadir requisitos una vez se han realizado las fases de análisis de necesidades y diseño.
- **Prototipado:** está basado en la construcción de prototipos que contienen la funcionalidad básica de la aplicación. Estos prototipos permiten al cliente evaluar la aplicación antes de que esté terminado el producto y así poder realizar cambios si fuesen necesarios. Se realiza un proceso iterativo, donde se va incrementando la funcionalidad del prototipo en cada entrega.
- **Incremental:** está basado en un proceso lineal e iterativo. El esquema *incremental* realiza todas las fases del desarrollo pero solo definiendo pequeñas partes de los requisitos

[1] http://dle.rae.es/srv/search?m=30&w=metodología

[2] https://es.wikipedia.org/wiki/Metodología_de_desarrollo_de_software

hasta que llega al producto final.

- **Espiral:** también está basado en un proceso lineal e iterativo. Las fases se conforman en forma de espiral y en cada bucle de la iteración se realizan un conjunto de tareas. Cada iteración por la espiral pasa por los cuatro cuadrantes de un circulo: *determinar objetivos*, *evaluar alternativas*, *desarrollar y verificar*, y *planear la próxima iteración* (véase Figura 4.1).
- **Rapid Application Development (RAD):** se centra en un desarrollo iterativo unido a la creación de prototipos y al uso de utilidades Computer Aided Software Engineering (CASE). Esta centrado en la usabilidad, utilidad y rapidez de ejecución.

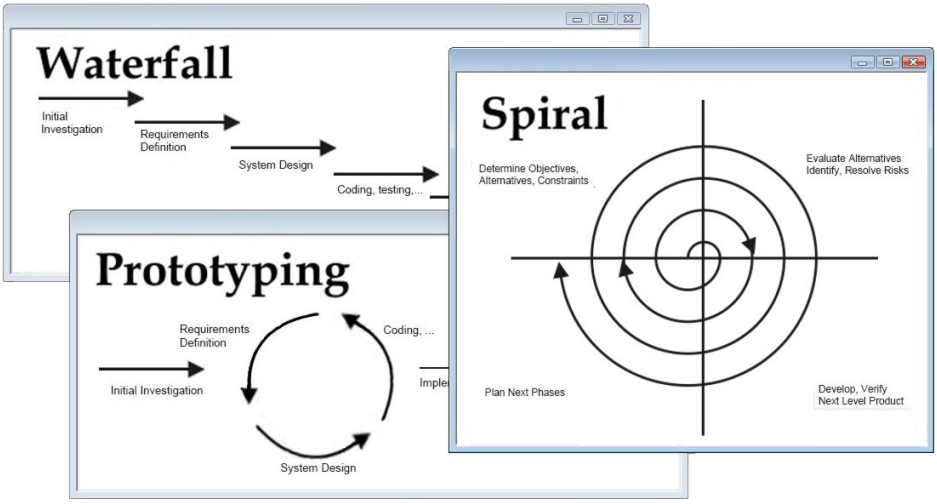

Figura 4.1: Tres patrones básicos en las metodologías de desarrollo de software. `https://es.wikipedia.org/wiki/Metodología_de_desarrollo_de_software`

Para el desarrollo de este TFG se ha elegido un esquema de tipo incremental, que permita incluir requisitos de forma progresiva y adaptar el diseño a las necesidades del proyecto. Cabe destacar que para este proyecto se busca agilidad a la hora de poder desarrollar el software correspondiente, sin necesidad de emplear demasiado tiempo en fases de análisis de requisitos y diseño.

Entre todas las opciones se ha escogido utilizar eXtreme Programming (XP) para el desarrollo de este proyecto, pero antes de continuar introduciremos un poco esta metodología.

4.1.1 ¿Qué son las metodologías ágiles?

En los años 90 se desarrollaron multitud de metodologías, entre ellas apareció el concepto de **desarrollo ágil** como contramedida a los métodos más tradicionales, estructurados y estrictos ya existentes.

Las **metodologías ágiles** como su nombre indica, son un conjunto de buenas prácticas que se llevan a cabo para realizar un proceso. Se basan, como ya se ha mencionado, en el

desarrollo iterativo e incremental, gracias a que son procesos iterativos, los requisitos y las soluciones, pueden modificarse según la necesidad del proyecto.

Una iteración del ciclo supone un incremento del valor del sistema, cada incremento añade funcionalidad por lo que desde el primer momento ya se dispone de funcionalidad. Este ciclo recorre una serie de fases: planificación, análisis de requisitos, diseño, codificación, pruebas y documentación. Aunque existe una fase de documentación es necesario explicar que los métodos ágiles enfatizan la comunicación cara a cara en vez de la documentación.

El **manifiesto agil**[3] enfatiza en:

- **Individuos e interacciones** sobre procesos y herramientas
- **Software funcionando** sobre documentación extensiva
- **Colaboración con el cliente** sobre negociación contractual
- **Respuesta ante el cambio** sobre seguir un plan

Dicho manifiesto incluye **doce principios** a tener en cuenta[4]:

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.

[3] `http://agilemanifesto.org/iso/es/manifesto.html`
[4] `http://agilemanifesto.org/iso/es/principles.html`

- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

Entre todas las metodologías agiles que aparecieron en torno a los años 90, cabe destacar: Scrum[5], Crystal Clear, eXtreme Programming (XP)[6], desarrollo de software adaptativo, feature driven development y Dynamic Systems Development Method (DSDM)[7]

4.1.2 Programación extrema

La **programación extrema** o eXtreme Programming (XP) es una metodología ágil, iterativa e incremental, formulada por Kent Beck, autor del libro *Extreme Programming Explained: Embrace Change* [Bec99].

Según H. Kniberg, en su libro *Scrum y XP desde las trincheras: Cómo hacemos Scrum*, el XP *«requiere que se complete algún tipo de producto potencialmente liberable al final de cada iteración. Estas iteraciones están diseñadas para ser cortas y de duración fija. Este enfoque en entregar código funcional cada poco tiempo significa que el desarrollador no tiene tiempo para teorías. No persiguen dibujar el modelo UML perfecto en una herramienta CASE, escribir el documento de requisitos perfecto o escribir código que se adapte a todos los cambios futuros imaginables. En vez de eso, el desarrollador se enfoca en que las cosas se hagan. Se acepta que puede equivocarse por el camino, pero también es consciente de que la mejor manera de encontrar dichos errores es dejar de pensar en el software a un nivel teórico de análisis y diseño y sumergirse en él, ensuciarse las manos y comenzar a construir el producto»* [Kni07]

De esta descripción se deduce que XP busca aumentar la productividad alejándose del papeleo que imponen otras metodologías y centrándose en crear un producto, que poco a poco irá evolucionando y mejorando con el tiempo.

Como en otras metodologías, como por ejemplo *Scrum*, existen distintos roles que tienen que cubrirse por miembros del equipo. La *programación extrema* define los siguientes roles:

- **Programador:** o desarrollador, escribe el código y las pruebas unitarias. Es el núcleo del equipo.
- **Cliente:** Define los requisitos del sistema definiendo las **historias de usuarios** y las pruebas funcionales que validan su implementación. Es quien elige las *historias de usuario* para la siguiente *iteración*.

[5] urlhttps://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software)
[6] urlhttps://es.wikipedia.org/wiki/Programación_extrema
[7] urlhttps://es.wikipedia.org/wiki/Método_de_desarrollo_de_sistemas_dinámicos

- **Tester:** es el encargado de todo el tema de pruebas, de diseñarlas, escribirlas, desplegarlas y de informar a los miembros del equipo con sus resultados.
- **Tracker:** es el encargado del seguimiento. Comprueba el grado de acierto entre las estimaciones y el tiempo real e informa al equipo para adaptar las futuras estimaciones.
- **Entrenador(coach):** es el guía del equipo. Responsable del proceso global y de llevar al equipo por el camino correcto.
- **Consultor:** es el único miembro externo del equipo, dado que en cada proyecto puede variar dado que sus conocimientos tienen que ser específicos al problema a solucionar.
- **Gestor(Big boss):** es el dueño de la empresa, suele ser el encargado de coordinar a los clientes y programadores.

La XP tiene una serie de características que definen a la metodología:

- **Desarrollo iterativo e incremental.**
- **Pruebas unitarias continuas:** estas pruebas son frecuentemente repetidas y automatizadas, incluyendo pruebas de regresión[8].
- **Programación en parejas:** la programación en parejas debe realizarse por dos *programadores* en un mismo puesto, debatiendo sobre la forma de codificar e implementar la funcionalidad que se les ha pedido.
- **Integración del equipo de programación con el cliente:** es una característica deseable para el mejor funcionamiento del proyecto.
- **Corrección de todos los errores:** hacer entregas frecuentes y corregir los errores antes de añadir nueva funcionalidad.
- **Refactorización del código:** se busca mejorar el código, sin cambiar la funcionalidad, para aumentar su legibilidad y mantenibilidad.
- **Propiedad del código compartida:** gracias a esta caracteristica es posible que todos los *programadores* del equipo conozcan y puedan modificar todo el sistema. Los *test de regresión* controlan que no se modifique ninguna funcionalidad pasada.
- **Simplicidad en el código:** parafraseando al refranero español *«Lo bueno, si breve, dos veces bueno»*. La *programación extrema* anima a realizar poca funcionalidad y cambiarla si es necesario en contraposición de añadir mucha funcionalidad que tal vez no se vaya a utilizar.

Otro punto de la *programacion extrema* son los cuatro valores en los que se basa: **simplicidad**, **comunicación**, **retroalimentación** y **coraje**. Un quiton valor, **respeto** fue añadido en la segunda edición de *Extreme Programming Explained* [BA05].

[8] `https://es.wikipedia.org/wiki/Pruebas_de_regresión`

La simplicidad es la base del XP. Simplificando el diseño se agiliza el desarrollo y se facilita el mantenimiento. Gracias a la refactorización del código es posible mantener dicha simplicidad. Otra de las medidas tomadas por el XP es mantener la documentación simplificada, basándose en comentar el código desarrollado lo mínimo posible y que este sea entendible por sí solo.

La **comunicación** se puede realizar de distintas formas y maneras. El código, para los desarrolladores, es una forma de comunicación, por lo tanto, y siguiendo también el valor de la *simplicidad*, realizar un código entendible, autodocumentado y con los comentarios necesarios mejorara la comunicación entre los desarrolladores al realizar programación en pareja.

Otro ejemplo de comunicación reside en las pruebas unitarias, estas, describen el funcionamiento y el diseño de las clases y métodos al realizar pequeños ejemplos del funcionamiento del sistema.

Por último, pero no menos importante, la comunicación con el cliente es fluida ya que este forma parte del equipo de desarrollo. Esto permite al cliente especificar que características son las más importantes y solucionar problemas sin demora.

Este tipo de comunicación con el cliente nos lleva a una **retroalimentación** muy eficiente. La opinión del cliente es conocida en tiempo real. Cuando se realizan ciclos cortos donde se entrega nueva funcionalidad se aumenta la posibilidad de detectar partes que no cumplen los requisitos.

El valor de **coraje o valentía** es un concepto muy genérico, pero se basa en que los miembros del equipo tienen que tener la suficiente determinación de realizar lo necesario para que el proyecto siga su curso. Por ejemplo, si es necesario eliminar parte de un código, solo se tiene que tomar el tiempo necesario para valorar los pros y contras, pero no demorarse.

El **respeto** es un valor que, a grandes rasgos, busca que los miembros del equipo siempre luchen por mejorar el producto. Para esto, las características y valores de la metodología guían hacia un sentimiento de unidad y de propiedad del proyecto que lleva hacia una mayor productividad. Un claro ejemplo de cómo la metodología lleva a un respeto entre los miembros del equipo es que las medidas de seguridad basadas en los test hacen que, de forma indirecta, los *programadores* respeten el código de sus compañeros.

Según P. Letelier y E. Sánchez. en su libro *Metodologías ágiles para el desarrollo de software: eXtreme Programming (XP)* [yES09] el **ciclo de vida** de XP sigue seis fases (véase Figura 4.2):

- **Exploración:** el o los *clientes*, en este caso David Vallejo, como director del proyecto, definen, a grandes ragos, las *historias de usuario*. Entre dichas *historias de usuario* se seleccionan las que tienen que ser implementadas en la próxima entrega. Mientras, los demás miembros del equipo, en este caso el alumno, se familiariza con las tecnologías

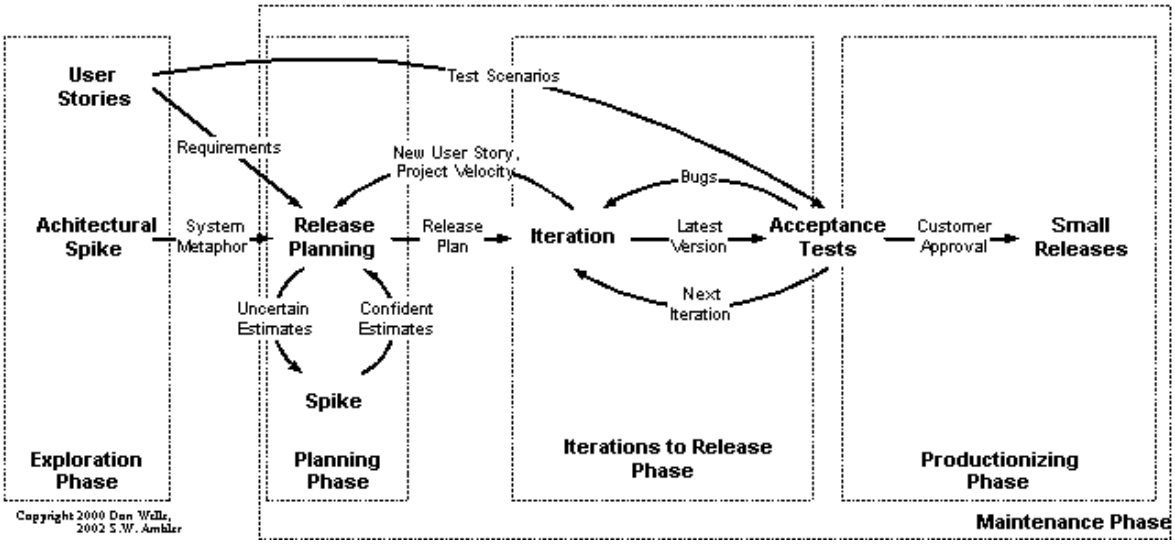

Figura 4.2: El ciclo de vida de la programación extrema. `https://eternalsunshineoftheismind.wordpress.com/2013/03/04/extreme-programming-xp-method-of-developing-i-s/`

y conceptos que se utilizaran para desarrollar el *TFG*.

- **Planificación de la entrega:** Se define un cronograma conjunto con el cliente donde se definen los hitos a alcanzar a partir de las *historias de usuario* y de la estimación realizada por los *programadores*
- **Iteraciones:** en cada iteración se busca aumentar la funcionalidad del proyecto. Cabe destacar que, por lo general, en la primera iteración se busca definir una estructura base que se utilizara como cimientos para el proyecto. Dependiendo de la funcionalidad implementada por las *iteraciones* es posible que se pase a *producción* el producto.
- **Producción:** es la fase donde se realizan las pruebas. También, es la fase donde se decide si incluir nueva funcionalidad a la versión actual.
- **Mantenimiento:** mientras se sigue incorporando nuevas funcionalidades es necesario realizar un mantenimiento del producto que está en fase de producción.
- **Muerte del proyecto:** una vez realizadas todas las *historias de usuario* y cuando el *cliente* ya no requiere añadir ninguna nueva, el producto pasa a la última fase, donde se realiza una documentación final del sistema y la arquitectura cierra a nuevos cambios.

  Las ventajas asociadas al uso de XP en este proyecto vienen derivadas de sus características y valores. Se obtendrá un producto con mayor rapidez. Serán aceptados cambios de requisitos de forma rápida e eficiente. La integración continua del producto lleva a la obtención de funcionalidades con mayor rapidez por lo que el *cliente* obtendrá un producto con mayor rapidez.

4.2 Herramientas

Los componentes HW y Software (SW) son un punto importante de este proyecto, por eso, y para poner al lector en contexto, los definiremos en este apartado.

4.2.1 Hardware

En este TFG se han utilizado los siguientes componentes HW:

- **Computador Toshiba.**
- **Conector Bluetooth® 4.0 Bluetooth Low Energy (BLE).**
- **Parrot Airborne Cargo Mars**[9] de *Parrot* (véase Figura 4.3):
  - **Estabilidad direccional:**
    - **Motorización:** 4 motores.
    - **Giroscopio:** 3 ejes.
    - **Acelerómetro:** 3 ejes $\pm$ 50 mg de precisión.
    - **Barómetro:** Sí.
  - **Memoria:**
    - **Almacenamiento:** Memoria Flash 1 GB.
  - **Cámara de fotos:**
    - **Cámara:** Minicámara integrada.
    - **Captura de imagen:** VGA (480 $\times$ 640) 300 000 píxeles.
  - **Velocidad:**
    - **Velocidad:** Hasta 18 km/h.
  - **Conectividad:**
    - **Conexión:** Vía Bluetooth® Smart / Bluetooth® 4.0 BLE.
  - **Alcance del Bluetooth:**
    - **Distancia máxima:** Alcance de 20 m.
  - **Batería:**
    - **Batería:** Polímeros de litio de 550 mAh.
    - **Autonomía:** 9 min (7 min con la carena).
    - **Tiempo de carga:** 25 minutos.
  - **Sistema operativo:**
    - **Sistema operativo:** Linux.
    - **SDK:** Disponibles para desarrolladores.
  - **Medidas y peso:**
    - **Medidas con carena:** 180 $\times$ 185 $\times$ 40 mm.

[9] https://www.parrot.com/es/minidrones/parrot-airborne-cargo-mars

- **Medidas sin carena:** 150 × 150 × 40 mm.
- **Peso sin carena:** 54 g.
- **Peso con carena:** 63 g.

Figura 4.3: Parrot Airborne Cargo Mars

4.3 Software

Para el desarrollo del TFG se han utilizado las siguientes herramientas SW:

4.3.1 Sistema operativo GNU/Linux Ubuntu 14.04.4 LTS

Es propiedad del empresario sudafricano Mark Shuttleworth[10]. **Ubuntu**es un SO basado en GNU\Linux, esta creado a partir del código base del proyecto *Debian*[11] y está distribuido como *software libre*. Su primer lanzamiento fue el 20 de Octubre del 2004.

Dispone de otras versiones orientadas a otros dispositivos: *Ubuntu Server*, orientada a servidores, *Ubuntu Desktop Remix*, orientada a empresas, *Ubuntu TV*, creada para los televisores y *Ubuntu Tablet* junto con *Ubuntu Phone*, creados para dispositivos móviles.

Cada seis meses sale una nueva versión de Ubuntu y recibe soporte por parte de Canonical durante nueve meses. Las versiones Long Term Support (LTS) se liberan cada dos años y reciben soporte durante cinco años. En la fecha de este documento, la última versión estable es la 17.04 denominada Zesty Zapus que fue lanzada el 20 de Octubre del 2016.

[10] https://es.wikipedia.org/wiki/Mark_Shuttleworth

[11] https://es.wikipedia.org/wiki/Debian

4.3.2 LaTeX, la clase esi-tfg y Texmaker

Para la maquetación y composición de este TFG se ha utilizado el lenguaje LaTeX[12] que es un conjunto de macros creadas por Leslie Lamport[13] en 1984 para el lenguaje de composición tipográfica desarrollado por Donald Knuth[14] TeX[15].

Aunque esta forma de maquetación puede ser un poco liosa, dado que no es tan visual como otros editores del tipo What You See Is What You Get (WYSIWYG), que quiere decir *«lo que ves es lo que obtienes»*, permite al escritor abstraerse de la complejidad de la maquetación y centrarse lo que se quiere transmitir. LaTeX interpreta el diseño del documento mediante *etiquetas* o *comandos*. Se puede definir inicialmente una estructura del documento, con sus capítulos, secciones y párrafos y ponerse a escribir sin volver a pensar en si todo quedara bien colocado.

Como estructura inicial y base de esta documentación se ha utilizado la clase **esi-tfg**[16] que otorga el grupo *ARCO*[17] de la *Escuela Superior de Informática de Ciudad Real*. Gracias a esta clase ha sido posible maquetar este TFG de forma rápida, fácil y sencilla.

El editor de texto utilizado para desarrollar el código LaTeX ha sido **Texmaker** es un editor multiplataforma liberado bajo licencia General Public License (GPL). Permite escribir documentos con LaTeX en una sola aplicación. Incluye además corrector ortográfico, autocompletado de comandos, plegado de código y un visor de pdf. Para que *Texmaker* funcione se tiene que tener instalado alguna distribución de TeX como: TeX Live, MiKTeX o proTeXt.

4.3.3 PyCharm Community Edition

Como Integrated Development Environment (IDE)[18] para el desarrollo del codigo del proyecto se ha elegido **PyCharm**[19] que está orientado al desarrollo en lenguaje *Python*. Es desarrollado por la empresa *JetBrains* y otorga al programador herramientas para el análisis del código, depuración gráfica, test unitarios y versión de controles, además de dar soporte al desarrollo web en *Django*.

Es una herramienta multiplataforma que es distribuida, en su versión *Community Edition*, bajo licencia *Apache License*. También dispone de una versión *Professional Edition* bajo licencia propietaria, dicha versión incorpora herramientas extra.

[12] https://www.latex-project.org/
[13] https://es.wikipedia.org/wiki/Leslie_Lamport
[14] https://es.wikipedia.org/wiki/Donald_Knuth
[15] http://www.cervantex.es/queestex
[16] https://bitbucket.org/arco_group/esi-tfg
[17] http://arco.esi.uclm.es/
[18] https://en.wikipedia.org/wiki/Integrated_development_environment
[19] https://www.jetbrains.com/pycharm/

4.3.4 Git

Para mantener un control sobre el código desarrollado y llevar un *control de versiones* se ha utilizado **Git**. *Git* fue diseñado por Linus Torvalds[20], baso su diseño en *BitKeeper* y en *Monotone*.

Entre las características más importantes de *Git* podemos encontrar:

- Multitud de **herramientas** y apoyo al **desarrollo no lineal**. *Git* incluye herramientas especificas para navegar y visualizar un historial de desarrollo no lineal
- **Gestión distribuida.** al igual que otras herramientas de *control de versiones* como: Mercurial, BitKeeper, Darcs, Bazaar y Monotone, Git genera una copia local del historial en cada nodo del *repositorio*. Los cambios se propagan entre los nodos del *repositorio*.
- Es posible publicar el almacén de información mediante **Hypertext Transfer Protocol (HTTP)**,**File Transfer Protocol (FTP)**, **rsync** y otros metodos.
- Es posible utilizar otras herramientas de *control de versiones* distintas a *Git* como **SVN** gracias a la aplicación *git-svn*.
- Gran **velocidad y eficiencia** en la gestión de grandes proyecto.

Como almacén del repositorio se ha utilizado el servicio gratuito de **Bitbucket**[21] que es un servicio. escrito en *Python*, de alojamiento basado en web y compatible con el sistema de control de versiones de *Git*.

4.3.5 Protégé

Para el diseño de la **ontología**[22] se ha utilizado la herramienta **Protégé**[23] (véase Figura 4.4). Es un editor de código libre para la creación de *ontologías* y sistemas de gestión de conocimiento mediante una interfaz grafica. Esta desarrollado por la *Universidad de Stanford*[24] aunque inicialmente fue creado en colaboración con la *Universidad de Manchester*[25], fue liberado bajo licencia *BSD 2-clause license*. Incluye clasificadores para validar la consistencia de los modelos.

[20] https://es.wikipedia.org/wiki/Linus_Torvalds

[21] https://bitbucket.org/product

[22] *«En ciencias de la computación y ciencias de la comunicación, una* ontología *es una definición formal de tipos, propiedades, y relaciones entre entidades que realmente o fundamentalmente existen para un dominio de discusión en particular. Es una aplicación práctica de la ontología filosófica, con una taxonomía.»* https://es.wikipedia.org/wiki/Ontología_(informática)

[23] http://protege.stanford.edu/

[24] https://en.wikipedia.org/wiki/Stanford_University

[25] https://en.wikipedia.org/wiki/University_of_Manchester

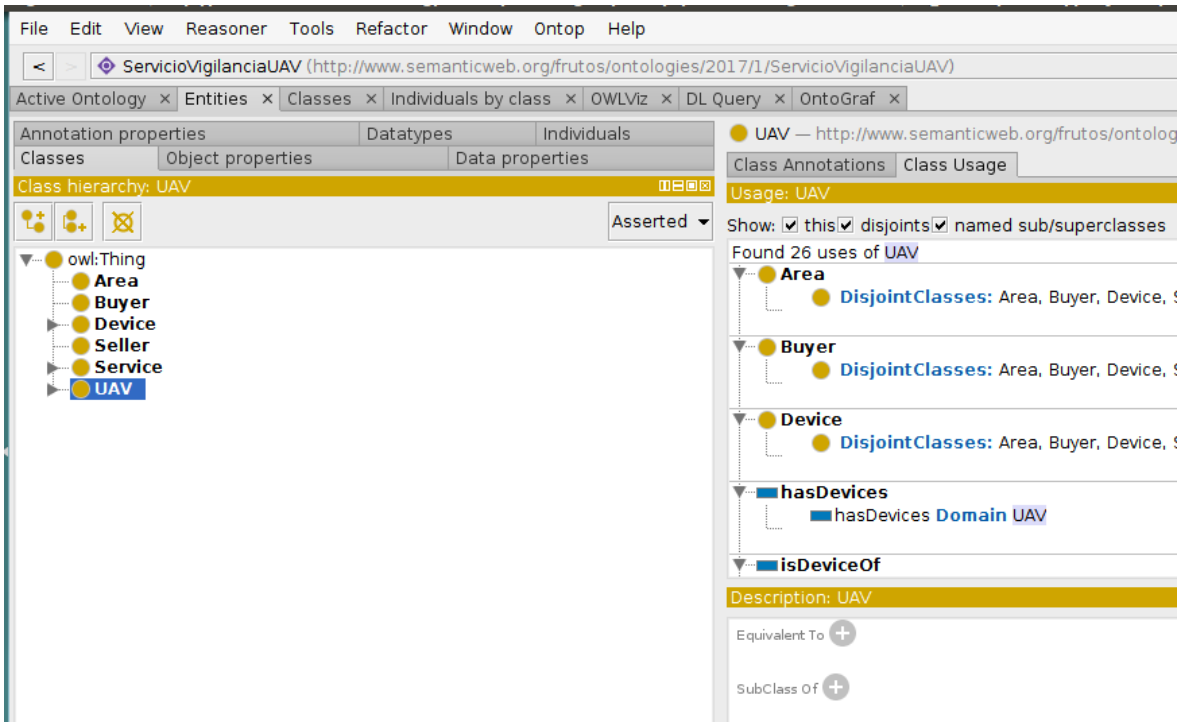

Figura 4.4: Protégé IDE

4.4 Lenguajes

4.4.1 Python 3.4

Para el desarrollo del código de este TFG se ha utilizado el lenguaje **Python** (véase § 3.2.4.1). Aunque el SDK de Parrot está orientado a lenguaje C++ se ha elegido el lenguaje *Python* dado que permite implementar código de forma rápida y sencilla, además de poder utilizar código escrito en C++ junto con el código de *Python*[26] o utilizar herramientas como *SWIG*[27]. Se ha elegido la versión 3.4 dado que la biblioteca de *Python* de **Owlready** que se utiliza en el TFG solo es compatible con las versiones de *Python* superiores a la 3.0.

4.4.2 Owlready

Owlready[28] es un módulo para programación orientada a *ontologías* basadas en Ontology Web Language (OWL)[29] y escrito en *Python*. Permite al desarrollador cargar ontologías en formato OWL *Extensible Markup Language (*XML*)* y comprobar la consistencia de dichos modelos. Se utilizara para unificar los *agentes* desarrollados en *Python* y otorgarles una es-

[26] `https://docs.python.org/3/extending/extending.html`

[27] `http://www.swig.org/exec.html`

[28] `https://pypi.python.org/pypi/Owlready`

[29] *«El Lenguaje de Ontologías Web (*OWL*) está diseñado para ser usado en aplicaciones que necesitan procesar el contenido de la información en lugar de únicamente representar información para los humanos. OWL facilita un mejor mecanismo de interpretabilidad de contenido Web que los mecanismos admitidos por XML, RDF, y esquema RDF (RDF-S) proporcionando vocabulario adicional junto con una semántica formal. OWL tiene tres sublenguajes, con un nivel de expresividad creciente: OWL Lite, OWL DL, y OWL Full. »* `https://www.w3.org/2007/09/OWL-Overview-es.html`

tructura descrita en una *ontología*. Permite añadir a las clases descritas en la ontología una serie de métodos definidos en *Python*.

4.4.3 SDK Parrot

Inicialmente, para este proyecto se pensó utilizar para la conexión con los UAV de *Parrot* su propio SDK. Con la idea de escribir un pequeño fragmento de código, en C++, que generaría una biblioteca externa, que permitirá, al *agente*, en este caso los *sellers* definidos en § 3.3.3.1, controlar el UAV desde su código escrito en *Python*, pero dado que la versión actual del SDK no permite, buscar dispositivos mediante *bluetooth* en lenguaje C++, se ha descartado su uso.

4.4.4 Nodejs

Entre las opciones existentes para conectar los UAVs de *Parrot* mediante *bluetooth* existen varias basadas en **Node.js**[30].

Node.js es un entorno multiplataforma de código abierto, para la capa de servidor, aunque no está limitado solamente a dicha capa. Node.js utiliza código **Javascript** para sus desarrollos y está basado en el lenguaje de programación *ECMAScript* asíncrono, con una entrada y salida basada en eventos y es muy similar en su propósito a *Python*.

Permite realizar de forma rápida y sencilla servidores API Representational State Transfer (REST)[31] permitiendo crear un servidor que controle al UAV mediante llamadas a la API REST.

Internamente, dicha API REST utilizada **node-rolling-spider**[32] un módulo creado para *nodejs* que permite manejar el UAV *Cargo* mediante funciones de *javascript*.

Node-rolling-spider utiliza **Noble**[33] como módulo para el uso de dispositivos BLE.

chapterArquitectura

En este capítulo, se analizara en detalle la arquitectura diseñada que servirá como base del sistema descrito en este TFG. Se definirán los módulos y agentes utilizados en la arquitectura y se detallaran las comunicaciones que realizan entre sí.

4.5 Visión general de la arquitectura

Uno de los principales problemas de este proyecto es la adaptabilidad al entorno, ya que es necesario que el proyecto no se quede enfocado a un solo caso de uso y que pueda adaptarse de forma fácil y sencilla a los distintos servicios que puede llevar a cabo un UAV actualmente y en un futuro.

[30] https://nodejs.org/es/

[31] https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional

[32] https://github.com/voodootikigod/node-rolling-spider

[33] https://github.com/sandeepmistry/noble

Para solucionar esta adaptabilidad de forma sencilla se ha optado por realizar una arquitectura modular dado que se busca una arquitectura que pueda solucionar todos las necesidades del proyecto y que tenga una gran versatilidad y capacidad de adaptación y crecimiento.

Estos elementos separados engloban distintas funcionalidades del sistema y si fuese necesario, podrían ser sustituidos por otros módulos que realizasen la misma tarea y que usasen la misma interfaz. La arquitectura modular también permite agregar módulos al diseño de forma sencilla, utilizando las salidas de otros módulos y comunicándose con ellos. Se ha diseñado una arquitectura basada en **módulos** (véase Figura 4.5). Esta arquitectura, como su nombre indica, está formada por módulos que deben seguir los conceptos de acoplamiento y cohesión y, a ser posible, ser independientes.

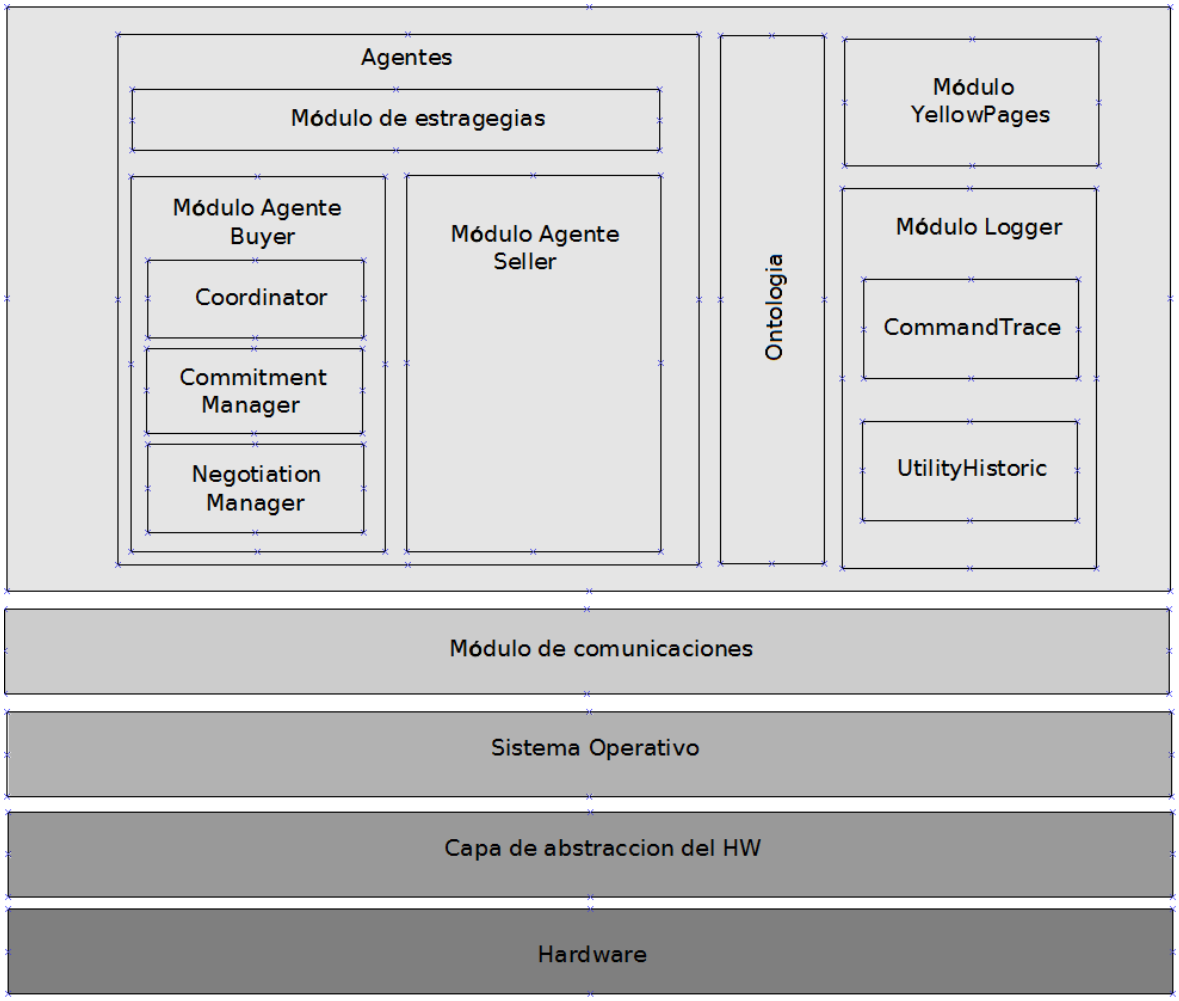

Figura 4.5: Diseño de Arquitectura del proyecto

Según Bertrand Meyer: *«El acto de particionar un programa en componentes individuales para reducir su complejidad en algún grado... A pesar de particionar un programa es útil por esta razón, una justificación más poderosa para particionar un programa es que crea una serie de límites bien definidos y documentados en el programa. Estos límites, o interfaces, son muy valiosos en la comprensión del programa»* [Mye78]

Entre las cualidades de la arquitectura modular destaca la capacidad que tiene para definir los límites de un módulo, definiendo sus entradas y salidas a través de interfaces o sistemas de comunicación haciéndola una arquitectura muy apta para este proyecto dado que puede gestionar la complejidad actual y futura mediante la implementación de módulos que satisfagan las necesidades del cliente.

Entre los módulos que se han creado tenemos:

- **Modulo Comunicaciones:** encargado de gestionar las comunicaciones entre los agentes. El diseño de este módulo es similar al diseño de un **patrón fachada**[34] y sirve de enlace de comunicaciones para todos los agentes y módulos. Para la realización de este proyecto se ha optado por canales de tipo *Queues* de *Python* pero podrían ser sustituidos por llamadas HTTP a una API REST u otro cualquier método de comunicaciones, como por ejemplo: los sockets de *ZeroC Ice*[35].
- **módulo YellowPages:** registra a los *sellers* y los lista a petición de los *buyers*. Este módulo puede ser suplantado por cualquier módulo que realice el servicio de un directorio de servicios. Es ejecutado en un hilo aparte y realiza una escucha activa de los canales de entrada definidos. Define dos canales de entrada utilizados para la comunicación con el exterior.
- **módulo Logger:** implementa toda la funcionalidad relacionada con el análisis forense de los datos. Es el encargado de tratar los datos generados durante la subasta y generar documentos útiles para el análisis. Cada sub-módulo define su propio canal de entrada para recibir datos del exterior.
- **Ontología:** es un módulo que aporta una estructura a los agentes. Dicha estructura está basada en una ontología que describe un servicio orientado a vigilancia mediante UAV.
- **Agentes:** engloba todos los agentes descritos en el apartado § 3.3.3.1. Estos agentes son los actores del sistema y son los encargados de realizar la subasta. Se han definido los siguientes sub-módulos:
  - **módulo Agente Buyer:** define al agente *buyer*. Este agente realiza el rol de *buyer*, como su nombre indica, e internamente puede ejecutar una serie de hilos que le dan ayudan durante el proceso de la subasta.
  - **módulo Agente Seller:** define al agente *seller*. Este agente realiza el rol de *seller*, como su nombre indica. Dicho agente espera una llamada externa para iniciar el proceso de subasta. También es el encargado de activar el UAV en caso de ganar la subasta.

[34] *«Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. This can be used to simplify a number of complicated object interactions into a single interface.»* `http://wiki.c2.com/?FacadePattern`

[35] `https://zeroc.com/`

- **módulo de estrategias:** en este módulo se encuentran los distintos tipos de estrategia que pueden ser utilizadas por los agentes. Define que actitud tendrán los agentes frente a las ofertas y contraofertas recibidas.

Como se puede observarse en el diagrama de flujo de la Figura 4.6, que representa una negociación entre un agente *buyer* y un agente *seller*, existe una comunicación constante entre los distintos agentes y sus módulos. Esta comunicación entre agentes es realizada gracias a la interfaz que otorga el módulo de comunicaciones.

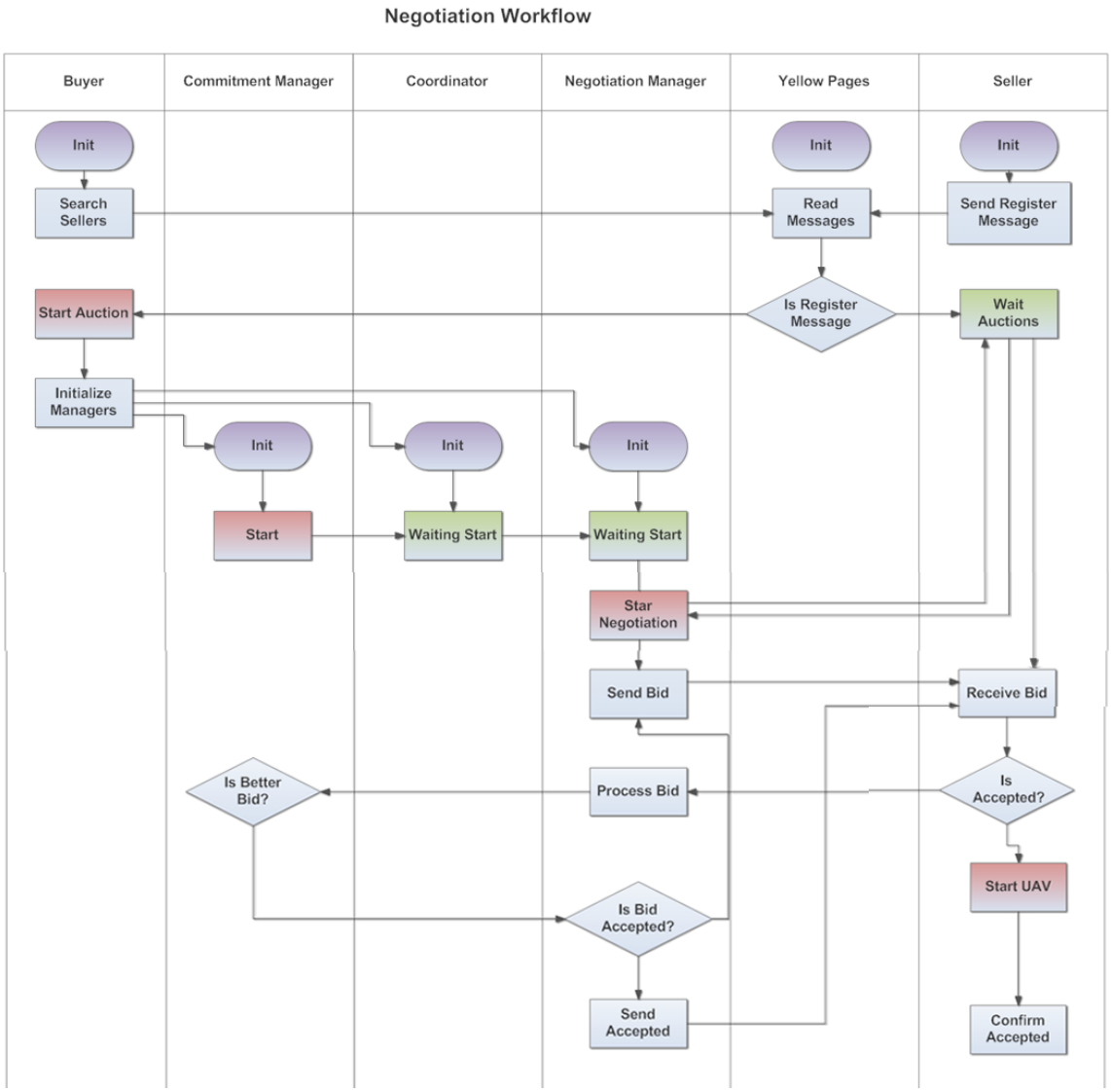

Figura 4.6: Negociación realizada entre un *buyer* y un *seller*

Entrando en más detalle, en el flujo de datos podemos observar como el agente *seller* se registra en el módulo *Yellow pages* (véase Figura 4.7).

Posteriormente, independientemente del tiempo, el agente *buyer* pide la lista de *sellers* que otorguen el servicio deseado e inicializa el proceso de puja con ellos. En este proce-

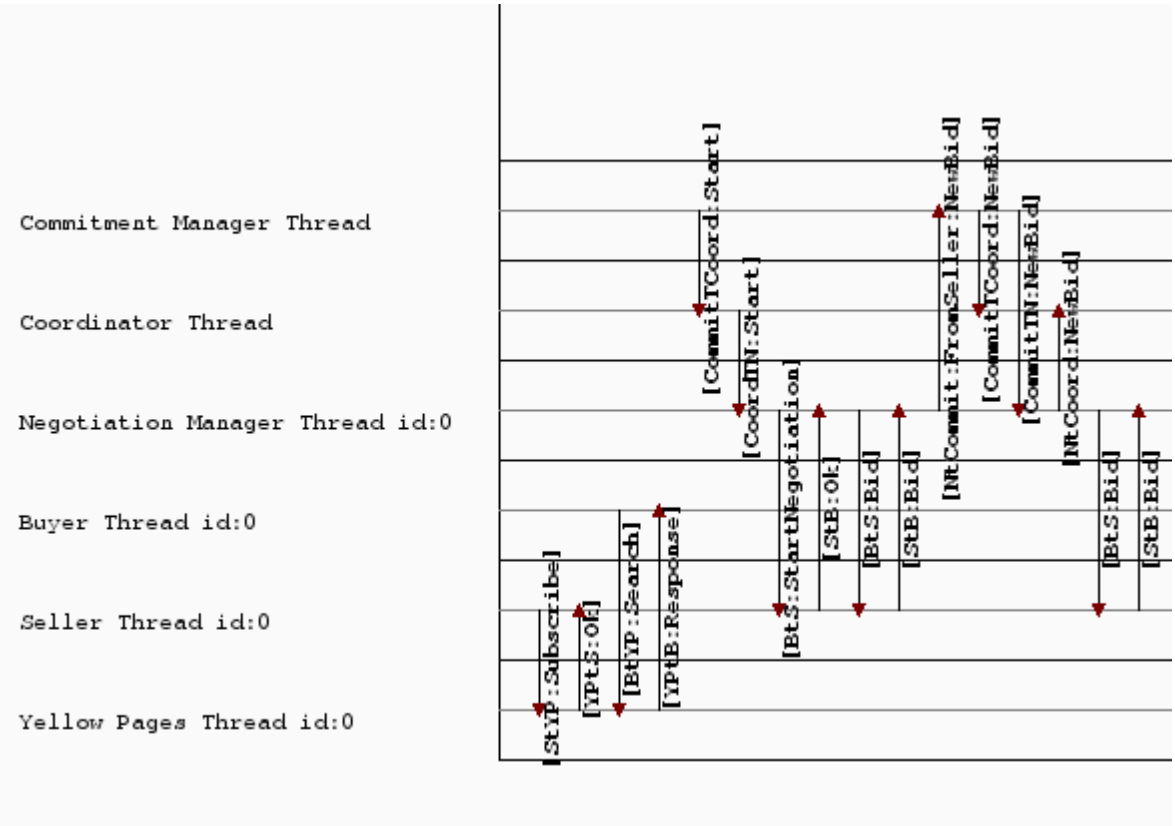

Figura 4.7: Mensajes enviados durante el registro del *seller* en Yellow pages

so, el agente *buyer* crea los *hilos* necesarios para el *Commitment Manager*, *Coordinator* y tantos *Negotiation Manager* como *sellers* otorguen el servicio, en este caso solo uno. El *Negotiation Manager* inicia un proceso de pujas con el *seller* en el cual intercambian ofertas y contraofertas hasta que el *Commitment manager* decide aceptar la puja y avisar al *Negotiation Manager* (véase Figura 4.8) (cabe destacar que el *seller* no acepta las pujas, solo las iguala si lo cree necesario).

Cuando esto sucede, se manda un mensaje de *Bid Accepted* al *seller* y este inicia la actividad con el UAV (véase Figura 4.9). Aunque no está reflejado en el diagrama de flujo, dado que solo representa la negociación entre un *buyer* y un *seller* es posible que el *buyer* cancele su negociación con el *seller* y este tenga que cesar la actividad con el UAV.

Como base para la comunicación entre *agentes* se ha optado por desarrollar una derivación de una arquitectura de servicios general (véase Figura 4.10). En esta arquitectura se observa como un tercero presenta a los agentes y estos inician una conversación una vez presentados sin necesidad de que vuelva a intervenir el agente presentador. En la arquitectura propuesta para el proyecto, se ha utilizado el módulo de *Yellow pages* como *presentador* de los agentes.

La parte del sistema relacionada con los UAVs ha sido definida siguiendo la estructura

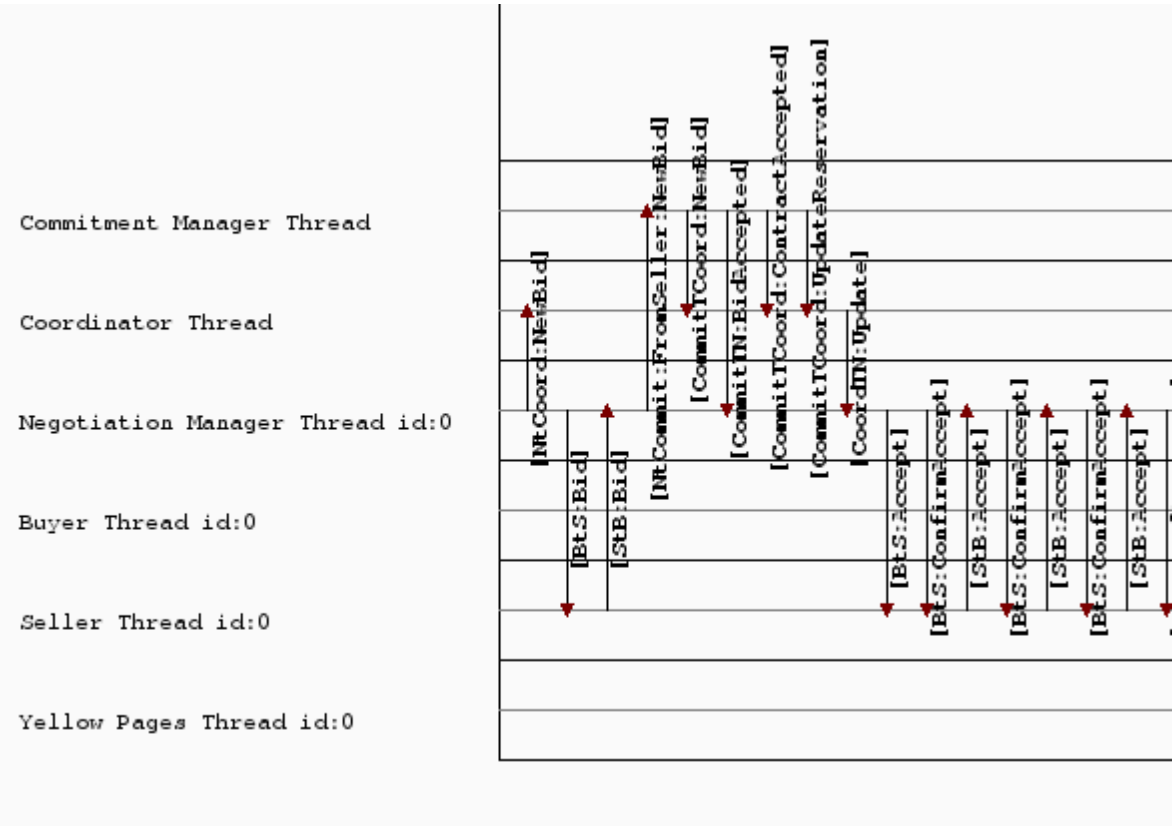

Figura 4.8: Mensajes enviados durante la aceptación de una puja por parte del *buyer*

básica de un servidor API REST de *Node.js* donde se define un enrutador para las direcciones de la API REST y una serie de acciones a realizar.

Han sido definidas las siguientes rutas para la API REST:

- **/UAV/connect:** realiza la primera conexión con el UAV. Acepta un JavaScript Object Notation (JSON)[36] con el UUID
- **/UAV/start:** acepta un JSON con el área a vigilar por parte del UAV
- **/UAV/stop:** finaliza la tarea del UAV por parte del agente.
- **/UAV/status:.** obtiene el estado de la batería del UAV.

4.6 Módulos y agentes

En esta sección se definirán las partes de la arquitectura, lo que se ha definido como módulos, y los agentes, en caso de contenerlos, que los componen.

Cada módulo plantea una situación o necesidad que tiene que ser cubierta para satisfacer al cliente y por consiguiente al proyecto.

[36] http://www.json.org/json-es.html

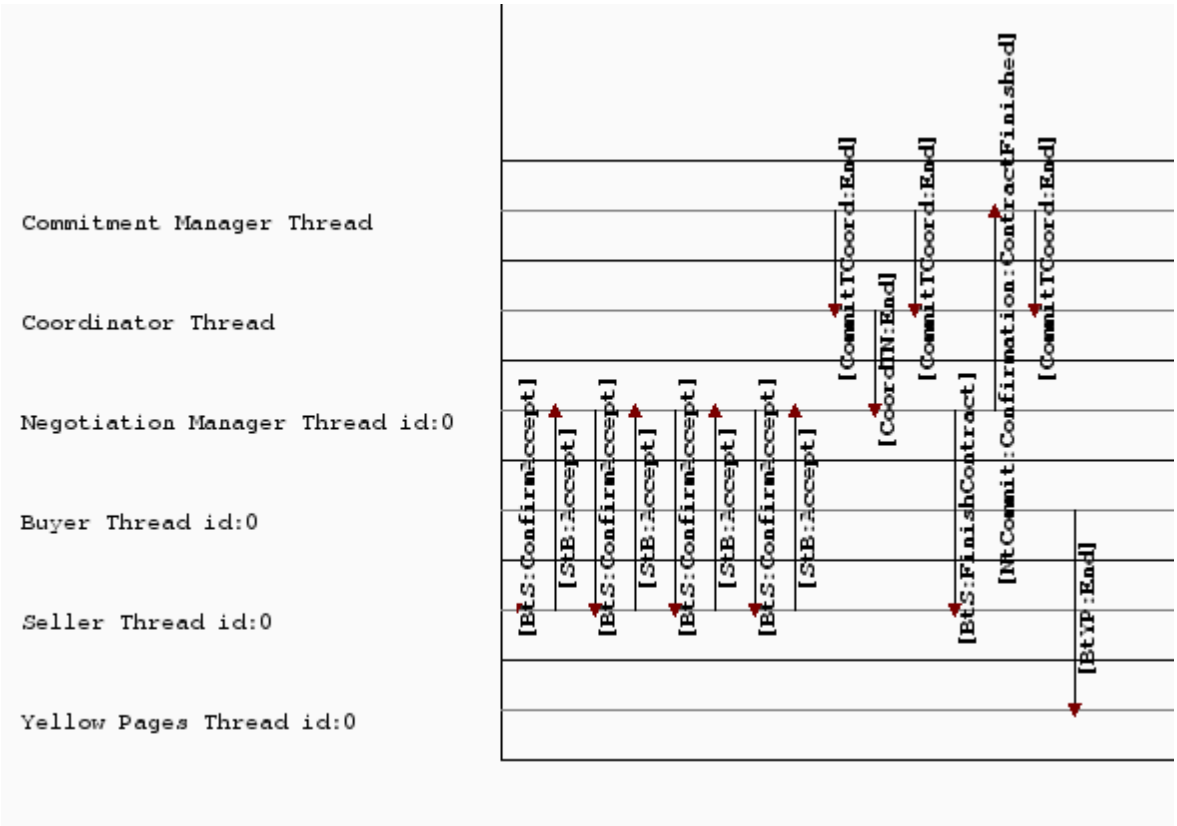

Figura 4.9: Mensajes enviados durante la finalización de la negociación entre un *buyer* y un *seller*

Figura 4.10: Diseño general de una arquitectura orientada a servicios

A continuación se detallaran dichas necesidades o problemas y como la inclusión de estos módulos los ha solucionado parcial o totalmente.

4.6.1 Módulo Comunicaciones

Este módulo, es la base de las comunicaciones del sistema.

Cuando existen distintos agentes e incluso estos agentes realizan procesos paralelamente

gracias al uso de *hilos*, surge la necesidad imperativa de definir bien las comunicaciones entre dichos agentes e *hilos*. Este problema queda resuelto con la creación de un módulo capaz de gestionar las comunicaciones de forma fácil y sencilla definiendo unos mecanismos de comunicación y de mensajes entendibles por el desarrollador.

Como podemos observar en el diseño UML de la Figura 4.11 este módulo incluye una clase **MessageManager** que es utilizada para enviar los mensajes, abstrayendo al desarrollador del uso del módulo *Queue* que incluye *Python*. Como podemos observar, se han definido dos funciones: **get_message** u **send_message** que utilizan reciben y envían los mensajes por una cola recibida. En el caso de *get_message* este código controla que el mensaje recibido sea el esperado y que la cola no esté vacía. Al recibir el mensaje devuelve un identificador de salida, un mensaje y el tipo del mensaje. El código de *send_message* envía por la cola recibida además de registrar en el **CommandTraceDiagrams** la acción realizada (ver Listado 4.1).

```
1  //...
2  def get_message(select_queue, block=False):
3      message_type = None
4      exit_id = NO_ERROR
5      message = None
6      try:
7          message = select_queue.get(block)
8      except queue.Empty:
9          # Handle empty queue here
10         exit_id = NO_MESSAGE_ERROR
11         pass
12     else:
13         message_type = get_type(message)
14         if message_type is None:
15             exit_id = TYPE_ERROR
16     return exit_id, message, message_type
17 def send_message(invoker_name, select_queue, message):
18     select_queue.put(message)
19     CommandTraceDiagrams().add_command(invoker_name, select_queue, message)
```

Listado 4.1: Fragmento de MessageManager

Este módulo también define una interfaz abstracta utilizada para definir a las clases mensaje que utilizaran los módulos y agentes para comunicarse entre ellos. Esta interfaz obliga a los mensajes a tener una función *to_string* y una función *to_diagram_string* que se utilizará para mostrar los mensajes.

Como se ha podido observar en el diagrama UML de la Figura 4.11 los submódulos: **Buyer**, **Seller**, **YellowPages**, **Coordinator** y **Commitment** contienen definiciones de los mensajes utilizados por dichos módulos o agentes. Por ejemplo, el sub-módulos que define los mensajes del agente *Buyer* contiene dos clases **BuyerToSellerMessages** y **BuyerToYe-**

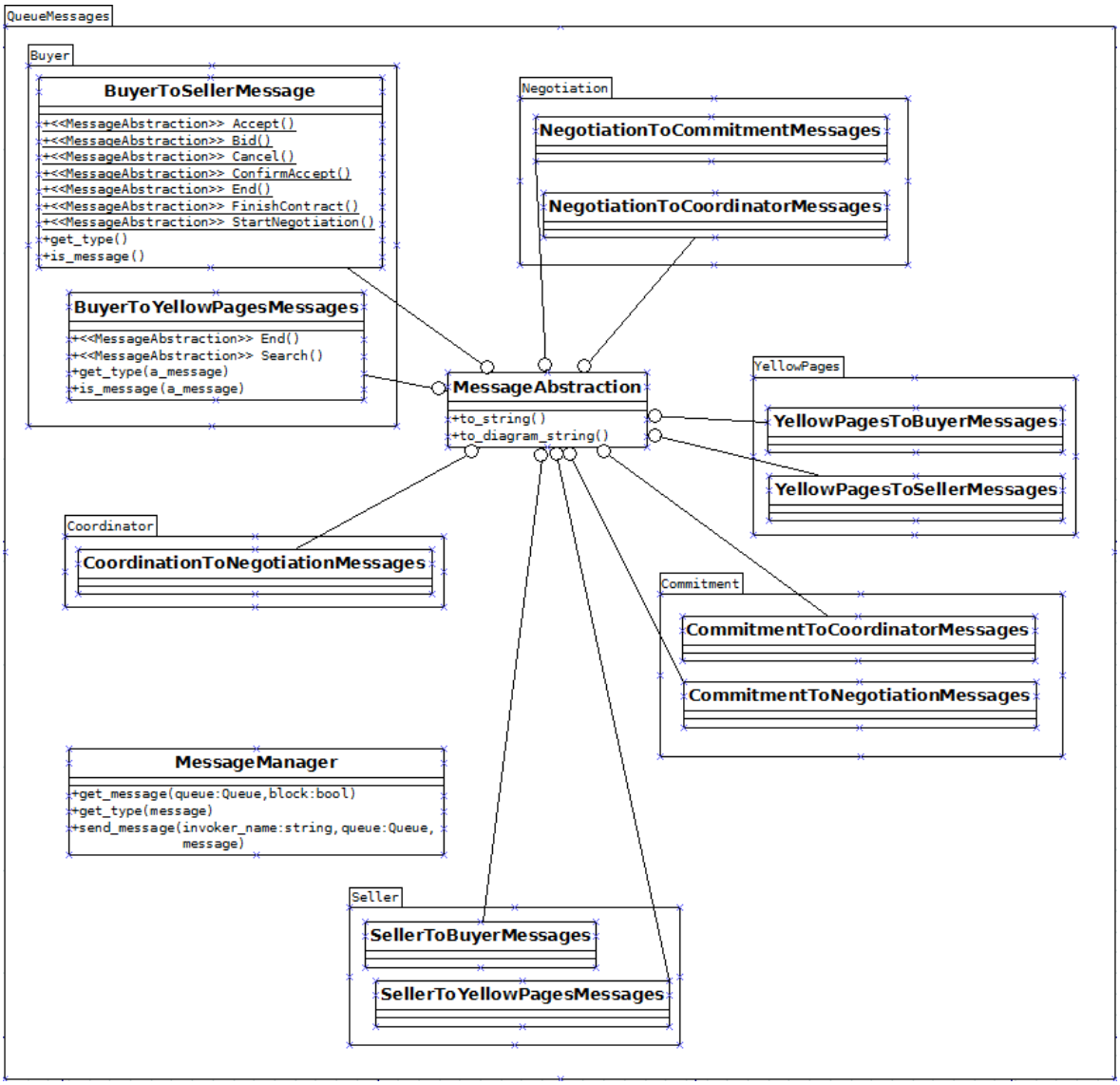

Figura 4.11: Diseño del módulo QueueMessages

llowPagesMessages que definen los mensajes, que como indica el nombre de la clase, son enviados desde el *Buyer* al *Seller* o desde el *Buyer* al módulo *Yellow pages*. Se ha utilizado una nomenclatura definida por el nombre del agente o módulo que envía el mensaje seguida de *To* y a continuación el nombre del módulo o agente que recibe el mensaje seguido de *Message*. Estas clases a su vez contienen otras clases que definen los mensajes que serán enviados.

Todas los sub-módulos incluyen sus propias clases y agrupan los mensajes que utilizan con el exterior. Cierto es, que estas definiciones podrían estar incluidas en sus propios módulos, pero se han agrupado en el módulo de comunicaciones para tener mejor control de ellas.

4.6.2 Módulo de Logger

Cuando se procesan gran cantidad de datos es necesario, para su entendimiento, poder procesarlos de forma automática y generar informes entendibles por el ser humano. El módulo **Logger**, gestiona los datos generados durante la subasta y otorga al cliente una serie de diagramas e informes que aportan un conocimiento del estado de la subasta fácil de comprender. Aunque este módulo está enfocado a satisfacer al cliente también es útil para el desarrollador ya que le permite realizar una trazabilidad de las comunicaciones entre los agentes y los hilos. Esta trazabilidad es necesaria cuando se trata de programas concurrentes multi-hilo.

Han sido definidas dos clases **CommandTraceDiagram** (mencionada anteriormente) y **UtilityHistoricDiagram**. Estas clases otorgan funcionalidad para un análisis forense de los datos generados durante la subasta por parte de los agentes.

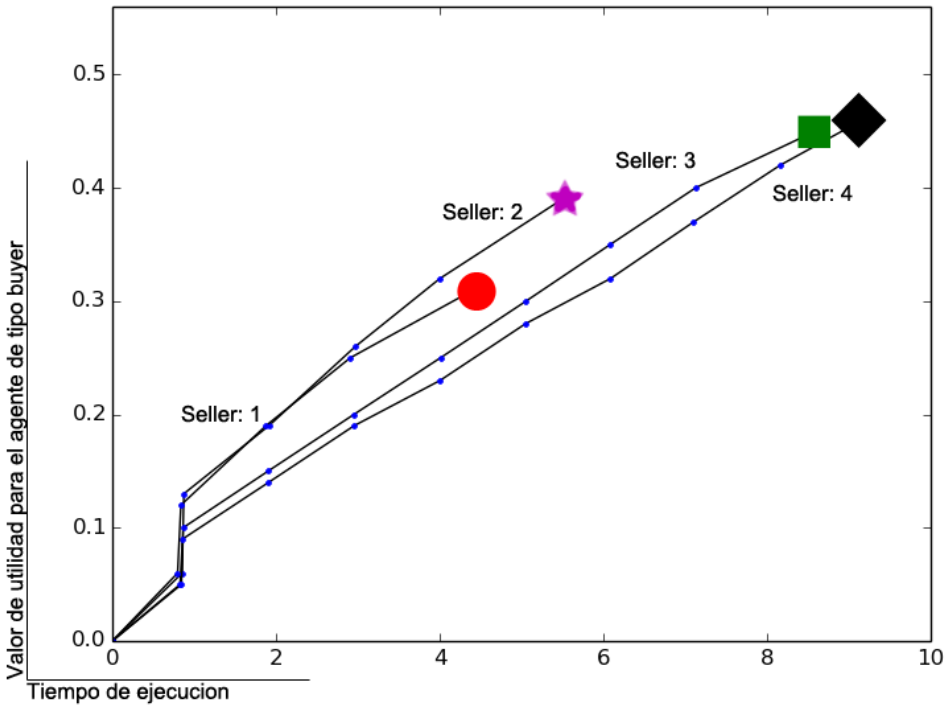

Figura 4.12: Gráfica con el historial de los valores de *utility* enviados por los agentes de tipo *seller*. Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

La clase *UtilityHistoricDiagram* utiliza la biblioteca **matplotlib**[37]. Esta clase define un **Singleton**[38] por el cual solo se puede tener una instancia de la clase *CommandTraceDiagram*

[37] `https://matplotlib.org/`

[38] *«A singleton is a class that allows only a single instance of itself to be created and gives access to that created instance. It contains static variables that can accommodate unique and private instances of itself. It*

en todo el sistema. Este *Singleton* controla que todo acceso del exterior sea manejado por una sola instancia de la clase y así quede toda la información almacenada en el mismo punto. Está centrada en el valor de *utility* que el agente *buyer* obtiene de las pujas de los *seller*. La clase otorga una función **create_diagram** que genera un gráfico de puntos que muestra el valor de *utility* de los agentes, dicho gráfico es exportado a formato *png* al finalizar la subasta. La Figura 4.12 es el resultado de una subasta con tres *sellers* con una estrategia *linear*. El cuadrado verde significa contrato finalizado, el círculo rojo contrato cancelado por el agente de tipo *buyer*, el diamante negro subasta finalizada sin llegar a un acuerdo, estrella morada es un contrato cancelado por parte del agente de tipo *seller*.

Por otro lado, la clase *CommandTraceDiagram* y al igual que *CommandTraceDiagram* utiliza un *Singleton* para recabar todos los datos generados durante la subasta. Utiliza la biblioteca Python Imaging Library (PIL)[39] para generar los diagramas e importarlos a formato *png*. La clase recibe los mensajes enviados por los agentes y los almacena separados para su posterior tratamiento. Una vez finalizada la subasta se invoca la funcion **draw_diagram** que dibuja el diagrama y lo importa a un archivo externo. El diagrama está formado por los agentes que han participado en la subasta (representados por su nombre y una línea horizontal) y los mensajes que han enviado (representados por una línea vertical que va desde el agente emisor al agente receptor junto con un identificador del tipo de mensaje). Las figuras 4.7, 4.8 y 4.9 han sido generadas por esta clase.

4.6.3 Módulo Ontología

Si se quiere dotar al sistema de una buena capacidad de adaptabilidad es necesario otorgarle un método de adaptación a las distintas estructuras definidas por los servicios.

Este módulo otorga, al sistema, una estructura definida a partir de una ontología orientada a servicios de vigilancia. Mientras se mantenga la estructura básica definida en dicha ontología es posible adaptar el sistema a cualquier servicio que pueda realizar un UAV.

Para diseñar y definir dicha ontología se ha utilizado la herramienta *Protégé* que permite definir las clases y propiedades de la ontología mediante una interfaz gráfica. Para este proyecto se ha descrito un sistema orientado a servicios de vigilancia, donde se han definido una serie de clases (véase Figura 4.13):

- **Area**
- **Buyer**
- **Device**

*is used in scenarios when a user wants to restrict instantiation of a class to only one object. This is helpful usually when a single object is required to coordinate actions across a system.»*https://www.techopedia.com/definition/15830/singleton

[39] http://www.pythonware.com/products/pil/

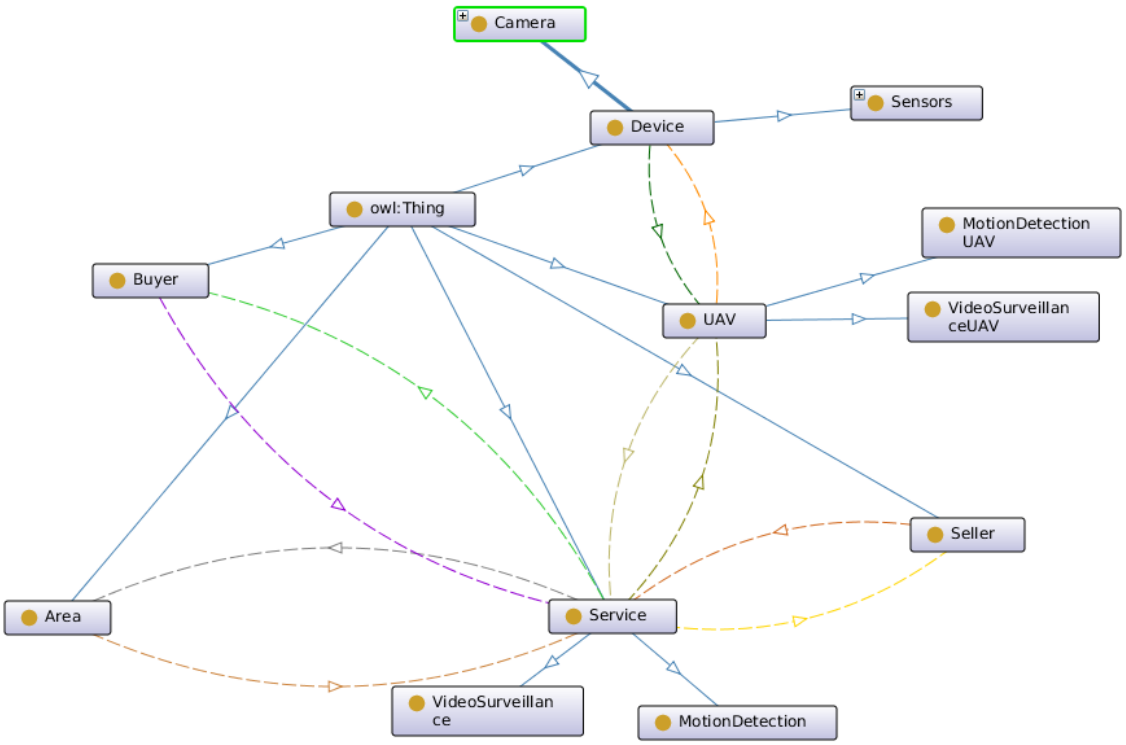

Figura 4.13: Diseño general de una arquitectura orientada a servicios

- **Camera**
  - **HD**
  - **Infrared**
  - **Thermal**
- **Sensor**
  - **LiDar**
  - **SONAR**
- **Seller**
- **Service**
  - **MotionDetection**
  - **VideoSurveillance**
- **UAV**
  - **MotionDetectionUAV**
  - **VideoSurveillanceUAV**

Estas clases están relacionadas mediante una serie de propiedades:

- **hasAssignedAn**
- **hasDevices**
- **hasSellA**
- **hasHiredAn**
- **isAssignedTo**
- **isDeviceOf**
- **isHiredBy**
- **isPerformedBy**
- **isSoldBy**
- **performsA**

La herramienta *Protégé* permite exportar la definición a un archivo XML con una estructura basada en OWL.

El módulo de *ontología* carga el archivo XML generado por *Protégé* en *Python* gracias a la biblioteca **Owlready**[40]. Dicho módulo define la **clase ontology** donde define las clases multi-heredadas de: *Buyer*, *Seller* que heredan la funcionalidad de los agentes y la estructura de la ontología y las clases heredadas de: *MotionDetectionUAV*, *VideoSurveillanceUAV*,

[40] `http://pythonhosted.org/Owlready/`

Area, *VideoSurveillanceService* y *MotionDetectionService* que incorporan funcionalidad a las clases definidas en la estructura.

4.6.4 Módulo YellowPages

Dado que se trata de un sistema descentralizado para la coordinación de agentes, es necesario definir un sistema que permita a esos agentes conocerse.

Entre las posibles soluciones se ha escogido la definida en la Figura 4.10, donde se observa como un tercero, externo al sistema, hace el rol de **presentador** y presenta a los agentes del sistema.

Por lo tanto, se ha definido un módulo que registra a los agentes *seller* en el sistema y los lista a petición. Esta interacción con el exterior se realiza mediante dos canales externos: uno para registrar a los *seller* y otro para pedir la lista de *sellers*.

Este módulo registra a los agentes *seller* en el sistema y los lista cuando se le invoca con un mensaje de tipo *search*. El núcleo de este módulo es la clase **YellowPages** que procesa los mensajes recibidos y dependiendo el tipo registra a los *sellers* o contesta con un listado de estos. La clase, en caso de recibir un mensaje de tipo **SellerToYellowPagesMessages.SUBSCRIBE_MESSAGE** (contiene: **service**, **queue** e **id**), registra a los *sellers* y registra el tipo de servicio que realizan gracias a la estructura definida por la *ontología*. En el caso de que el mensaje recibido sea de tipo *BuyerToYellowPagesMessages.SEARCH_MESSAGE* (contiene: **service**, **queue** e **id**), busca, entre los *sellers* registrados, cual ofrece el servicio requerido por el *buyer*.

4.6.5 Módulo Agentes

Este módulo satisface la necesidad del sistema de diseñar un mecanismo de subasta entre agentes concurrentes.

Este módulo engloba a todos los **agentes** utilizados por el sistema en el proceso de la subasta.

Otra de las necesidades derivadas del sistema de subastas es la posibilidad de utilizar distintos tipos de estrategia por parte de los agentes siendo el módulo **estrategias** donde se definen y agrupan las posibles estrategias. Esta agrupación, y gracias a la estructura modular, permite al desarrollador agregar nuevas estrategias de forma fácil y sencilla.

Formando parte de los agentes encontramos los sub-módulos: **Buyer** y **Seller**.

4.6.5.1. Sub-módulo Estrategias

Los agentes participes de una subasta tienen que ser capaces de adaptar sus estrategias de forma ágil. Para ello se ha implementado el módulo de **Estrategias** que define una estructura básica y los métodos obligatorios que tienen que tener las estrategias del sistema.

Estas estrategias definen un comportamiento a la hora de aceptar o rechazar ofertas y de realizar contraofertas. Las estrategias aquí implementadas están están basadas en el tiempo y definen un valor acorde a él.

Se han definido tres clases donde se define un tipo de estrategia: **StrategyLinear**, que hereda de la interfaz abstracta **StrategyAbstraction** y **StrategyConceder** y **StrategyTough** que heredan de la clase **StrategyLinear**.

La interfaz define un método **logic** que las estrategias tienen que implementar y que definirá la lógica de la estrategia. Dicho método devuelve una puja o **Bid** cuando se le invoca. En la clase *StrategyLinear* se ha definido una lógica que devuelve la utilidad calculada siguiendo una progresión lineal relacionada con el tiempo que lleva el agente pujando. Las clases *StrategyConceder* y *StrategyTough* basan su cálculo en la funcionalidad heredada de su padre *StrategyLinear* pero la aplican, o no, dependiendo de si el tiempo de funcionamiento del agente ha superado un límite.

4.6.5.2. Agente Seller

El agente **Seller** implementa toda la lógica que se utilizará durante la subasta. Funciona en un hilo independiente. Entre las funciones que implementa destacan:

- **Manage_message**: que gestiona los mensajes recibidos en la *queue* del *seller*. Según el tipo de mensaje realizara una acción de oferta o contra-oferta o cambiara de estado entre los definidos por el sistema. Dependiendo del estado procesara unos mensajes u otros.
- **Uav_start**: que realizara una llamada **HTTP** con método **POST** donde enviara el área que queda por vigilar.
- **Subscribe_to_yellow_pages**: como su nombre indica, se subscribirá al módulo *Yellow pages* enviándole el servicio que realiza y su cola de mensajes.

4.6.5.3. Agente Buyer

Este agente es el más complejo de todos los agentes implementados (véase Figura 4.14). Como el agente *seller*, el **Buyer** funciona en un hilo independiente y crea, según necesidad, una serie de hilos hijos para que le den soporte. El *buyer* crea un hilo **CommitmentManager**, **Coordinator** y tantos **NegotiationManager** como necesite.

Buyer: este hilo controla al agente *buyer*. Dicho agente pide al módulo *Yellow pages* la lista de *sellers* que realizan el servicio que desea. Una vez obtiene dicha lista, pasa a crear los hilos de *CommitmentManager*, *Coordinator* y tantos *NegotiationManager*. Podemos observar en la Figura 4.14 el diseño de la clase *buyer*. Podemos observar como las funciones implementan gran parte de la lógica de la clase. Por ejemplo: **search_seller** es la función encargada de ponerse en contacto con el módulo de *Yellow pages* y pedir la lista de agentes

sellers.

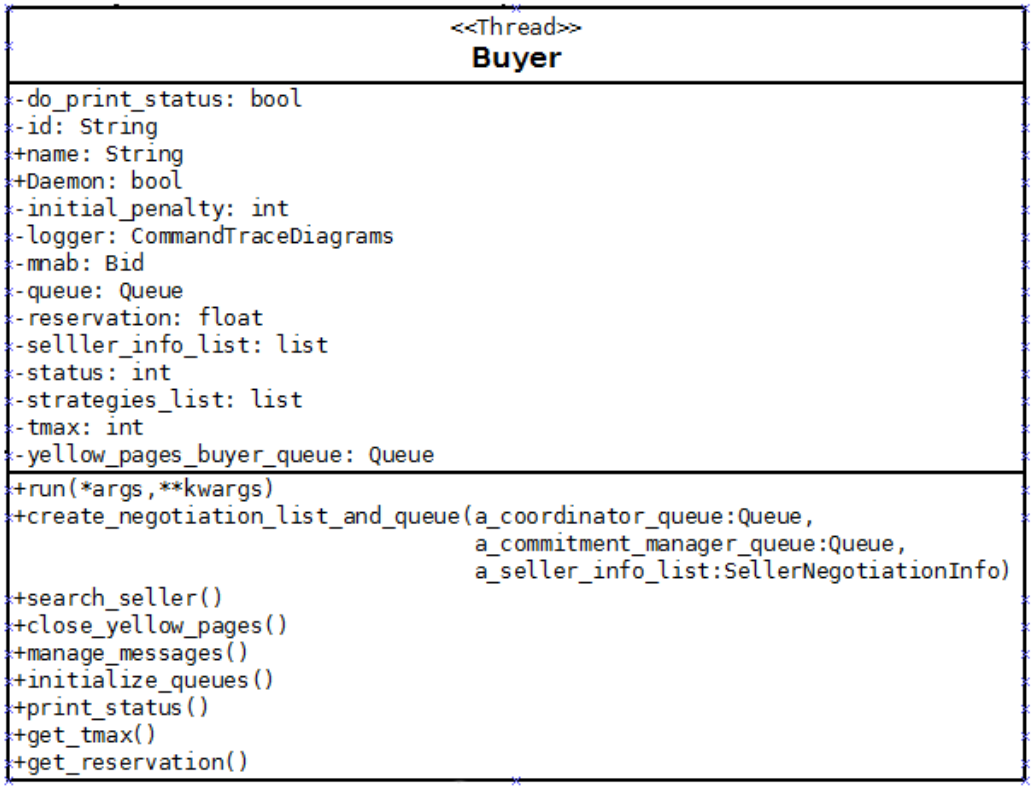

Figura 4.14: Diseño UML del Agente Buyer

CommitmentManager: es el hilo encargado, entre otras cosas, de:

- **Decidir** si las pujas que le envían los hilos *NegotiationManager* son las mejores o no. Esta decisión es tomada en la función **is_better_contract**.
- **Actualizar** los nuevos valores de **reservation** de los hilos de negociación.
- **Actualizar el histórico** de pujas. EL histórico de pujas se utiliza para calcular la estrategia a seguir por el *negotiation manager*
- **Pagar** la penalización. Los agentes deben pagar una penalización si cancelan un contrato parcial.

Coordinator: este hilo crea los hilos de *NegotiationManager* y actualiza:

- **Estrategias** mediante la función **update_seller_strategy** y **calculate_best_strategy**.
- **Matrices de utilidad histórica** y de éxito mediante la función **update_successful_-matrix** y la función **update_utility_historic_matrix**.
- **Calcular la utilidad esperada** de todos los *sellers*.

NegotiationManager: encargado de negociar con los agentes *seller*. Este hilo procesa las ofertas recibidas y contesta al *seller* asignado. Para la definición de sus funciones se ha

utilizado una nomenclatura definida por verbos, tipos y nombres, por ejemplo: **send_bid_-to_seller_and_get_response** o **manage_bid_offer_message**, por lo que, permite, al que lea el código, seguir el flujo de los mensajes, aunque se trate de un código asíncrono (véase Figura 4.15). Como podemos ver en el diseño UML la clase *NegotiationManager* hereda de *Thread* y se comunica con el agente de tipo *seller* mediante las funciones: **do _seller_**. También realiza tareas de control con las funciones: **manage_**.

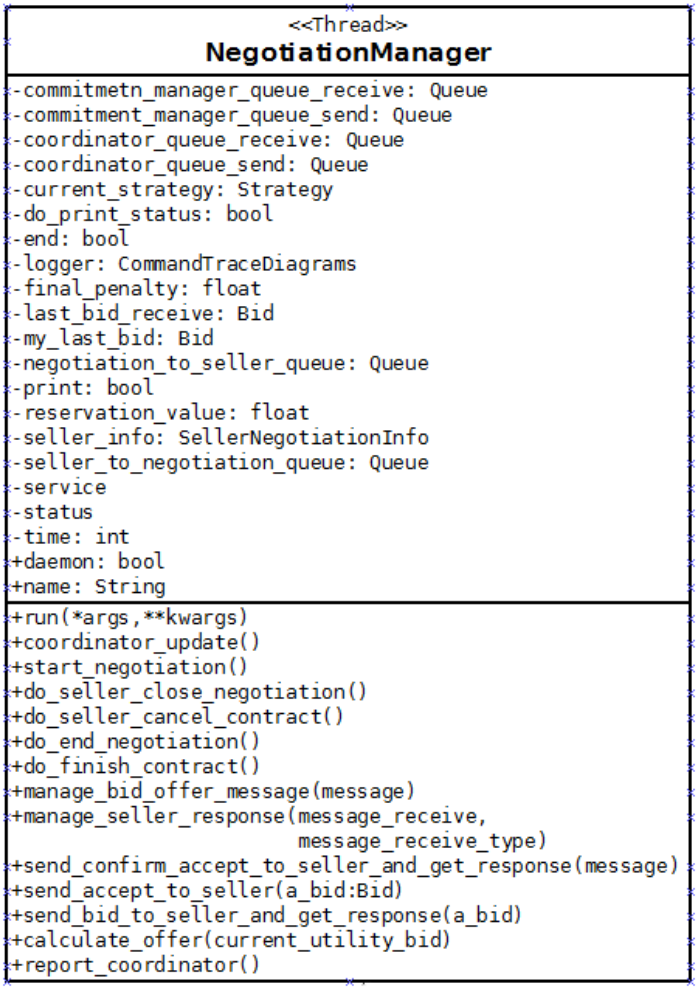

Figura 4.15: Diseño UML del hilo Negotiation Manager

Capítulo 5

Evolución, resultados y costes

ESTE capítulo recoge las fases por las que ha tenido que pasar el proyecto desde sus inicios. Entre los detalles que se expondrán, se detallaran los problemas surgidos y como se han solucionado además de detallar los resultados obtenidos al realizar los casos de estudio propuestos. Para finalizar el capitulo se desglosaran los costes derivados de elaborar este TFG.

5.1 Evolución

Este proyecto empezó a bocetarse a mediados de Octubre del 2016. Tomando como base el articulo visto en el apartado § 3.3.3.1 se decidió aplicarlo a un entorno orientado a la sincronización de UAV.

Desde el principio se decidió abstraerse del uso intensivo de los UAV y orientar el proyecto hacia un entorno más teórico realizando pruebas de concepto orientadas a la coordinación y no centrándose en como el UAV realizar el trabajo una vez finalizada la subasta pero si en que atributos del UAV pueden ser claves para dicha coordinacion. Por eso se eligió utilizar UAV que pudiesen ser controlados mediante un PC que hiciese la función de un GCS.

Aun así, dado que se trabaja con dispositivos HW y con SW actual, han surgido, durante algunas fases del proyecto, problemas, relacionados con el proyecto, que han sido solventados de forma ágil. Esto es posible gracias al uso de la metodología de XP (véase § 4.1.2) que permite adaptar el flujo en relación a las necesidades del grupo. También, y dado que el alumno ha realizado el proyecto de forma paralela a su jornada laboral, algunas fases se han visto ralentizadas dependiendo de su carga de trabajo.

A continuación se detallaran las fases, una a una, por las que ha pasado este proyecto (véase Figura 5.1) (véase Cuadro 5.1).

5.1.1 Fase 1: Inicio e Investigación. 15 de Octubre 2016 - 15 Enero 2017

Es la primera etapa del proyecto. En esta fase se cubrió la necesidad inicial de **investigar** la propuesta realizada por el **cliente**. Se leyeron los **documentos** oportunos (véase § 3.3.3.1) y se plantearon las posibles implementaciones llegándose a la conclusión de que era factible realizar una sincronización de forma concurrente por parte de los UAV. Otro punto a

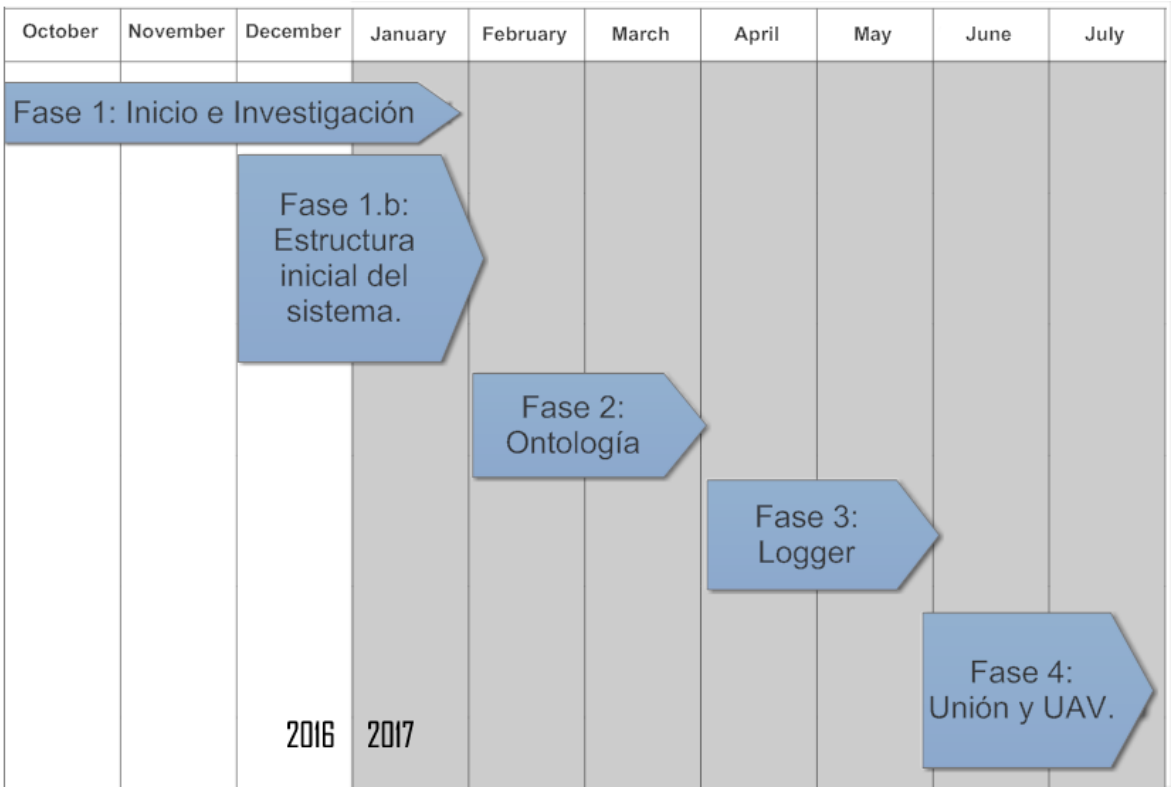

Figura 5.1: Histograma detallando las fechas y fases del proyecto

investigar fueron los posibles UAV del mercado enfocándose en cuales se adaptaban mejor a un entorno de pruebas reducido, entre las opciones se seleccionaron los MUAV de la empresa *Parrot*, en concreto sus modelos *Cargo*. Entre las variables que se tomaron en cuenta se determino que los distintos métodos de programación existentes y su tamaño reducido los hacían aptos para el proyecto. Posteriormente se escogieron cuatro UAV del modelo *Cargo* que habían sido utilizados para la carrera de UAV realizada por la *Escuela Superior de Informática de Ciudad Real* el 8 Noviembre del 2017.

5.1.1.1. Fase 1.b: Estructura inicial del sistema. 1 de Diciembre 2016 - 10 Enero 2017

Durante la fase de investigación se fueron realizando pruebas escritas en lenguaje *Python* durante las cuales fue evolucionando el sistema. Como se ha podido observar en el cuadro 5.1 se empezó por una estructura sencilla y se fueron añadiendo los hijos al agente *buyer* poco a poco. Los puntos realizados, cronológicamente, durante esta fase son:

- **Estructura inicial:** utilizada para separar los módulos y los agentes.
- **Creación del agente buyer:** se fueron desarrollando poco a poco las estructuras para los hilos del agente *buyer*.
- **Estructura inicial del modulo de comunicaciones:** se definió una estructura inicial para los mensajes que serian utilizados para la comunicación entre hilos.

Commit	Message	Date
7ecc204	doc and demo fixes	2017-06-22
f787b7f	Se ha añadido la funcionalidad relacionada con la API REST del UAV. Se han restructurado algunos modulos	2017-06-20
94e989f	correcciones y Programacion Concurrente	2017-05-31
76565b3	Documentacion: Introduccion a los sistemas multi-agentes	2017-05-21
22949df	Documentacion añadida	2017-05-21
7b4c8ea	Diagrama de historial de utilidad	2017-04-16
4373e41	En rojo los contratos cancelados, en negro los que no han llegado a un contraro y en verde los finalizados	2017-04-16
c450fd1	Log Command Tracer Diagrams	2017-04-16
5a9870c	Code Style	2017-04-14
ee48b04	Update to Python3.0	2017-02-11
72863eb	Historic, expected utility, etc etc	2017-01-10
0fc1b09	Expected Utility done	2017-01-03
af8621f	Atypes added : Conceder and non-conceder	2017-01-03
61e52c2	Demo completed with multiple agents and cancel contract from seller	2017-01-02
4d34f30	Demo completed with multiple agents	2017-12-28
5701110	Demo completed	2016-12-27
2026bc5	192 Unittest Pre changes in communications	2016-12-23
ce42cce	Unittest	2016-12-22
f473bee	Add Seller Agent Add Test Pre-change in Messages Class	2016-12-19
0b62ce8	Strategy change to Agents root Strategies: - Linear - Conceder - Tough	2016-12-23
2026bc5	192 Unittest Pre changes in communications	2016-12-15
3b073d3	Class Bid added.	2016-12-15
340a5ce	Utils errors to print errors Message Manager added Negotiation thread logic implement, demo	2016-12-14
224fece	Seller rename to Negotiation	2016-12-13
143defe	QueueMessages modificado, se han agrupado los mensajes por tipo	2016-12-13
f559dec	Seller Manager Thread Seller Manager se encargara de guardar la informacion del vendedor y gestionar las comunicaciones con el	2016-12-06
914d6f9	Commitment Manager and Coordinator Threads	2016-12-06
074906f	Estructura Inicial	2016-12-13

Cuadro 5.1: Registro de commits en el repositorio de git

- **Creación de estrategias:** se definió la lógica de las estrategias a usar por los agentes.
- **Creación del agente seller:** una vez se había definido bien el agente *buyer* se paso a crear el agente *seller*.
- **Demo:** se definió una demo inicial donde un solo *buyer* se comunicaba con un *seller* e intercambiaban ofertas durante un tiempo determinado.

Durante el desarrollo de estos puntos surgio un problema a causa del número elevado de comunicaciones, estados y formulas. Se opto por seguir la recomendación de la metodología XP de realizar **test unitarios** para así comprobar que todas las funciones realizaban bien su trabajo. Gracias a los 192 test unitarios que se realizaron se pudo solucionar de manera rápida y eficiente este problema encontrando bugs y errores en funciones y formulas mal implementadas.

También se hizo hincapié en el modo de mostrar la información al usuario y se implemento una estructura fácil de leer para mostrar por pantalla los datos que se generaban durante la subasta.

La **demo fue presentada** al cliente el día 10 de Enero del 2017 y se opto por pasar a la siguiente fase. Por último y dado que se ha llevado a cabo un enfoque orientado a pruebas, se detallaran las pruebas evolutivas realizadas a lo largo de estas fases.

5.1.2 Fase 2: Ontología. 10 de Febrero 2017 - 15 de Marzo 2017

En esta fase, el cliente añadió a los requisitos la implementación de una **ontología de servicios** que pudiese ser adaptada al sistema de forma fácil y sencilla. Se investigaron las distintas herramientas para desarrollar ontologías de manera gráfica y las distintas estructuras o formatos de las ontologías y se opto por utilizar la herramienta *Protégé* y la biblioteca de *Python OwlReady* para desarrollar esta **historia de usuario**. Cabe destacar que la mayoría de las herramientas gráficas para la definición de ontologías están obsoletas o son de pago. Una vez finalizada la investigación sobre las ontologías, se realizó con la herramienta *Protégé* un diseño general de servicio orientado a la vigilancia con UAV. Con el diseño ya realizado se opto por realizar una prueba de integración con *Python* en el cual se cargo la estructura y se modificaron las clases obtenidas del archivo generado por *Protégé*.

Durante esta fase surgió un problema de incompatibilidades entre las versiones de *Python* dado que la biblioteca *OwlReady* solo funciona con versiones de *Python* superiores a la 3.0. Se tomo la decisión de adaptar el código de *Python* de la versión 2.7 a la 3.0.

Paralelamente a la investigación relacionada con las ontologías, se fue mejorando el código ya implementado y corrigiendo errores y añadiendo implementación de formulas al sistema.

Entre todas las formulas del documento base se opto por implementar solamente las necesarias para la demo final. Mencionar que las formulas no implementadas si son invocadas por el sistema pero no realizan ninguna operación. Las formulas implementadas son:

- **Expected Utility:** se implemento la funcionalidad que calculaba la utilidad esperada de los *sellers*.
- **PO y PS:** se implementaron las matrices que intervienen en la decisión inicial de la matriz de tipos. Se crea la matriz desde cero sin intervención externa.

5.1.3 Fase 3: Logger. 1 de Abril 2017 - 17 Abril del 2017

Como su nombre indica, esta fase está centrada en la **creación de un sistema de logs** que aporten información al cliente. Durante esta fase se implementaron dos sistemas capaces de generar archivos *.png* con los datos obtenidos de la subasta.

El sub-módulo **CommandTracer** otorga información relevante al tipo de mensajes enviados por los hilos del sistema y el **UtilityHistoric** muestra una gráfica con los valores de *utility* recibidos por los *NegotiationManager*. Durante esta fase no surgieron grandes problemas, solo tres leves:

- **Superposición de mensajes:** inicialmente se planteo mostrar los mensajes en relación al tiempo de ejecución, pero el gran número de estos y sus tiempos similares hacia que los mensajes quedasen superpuestos los unos con los otros. Al final se opto por mostrarlos según orden de llegada y no según el tiempo de envió.
- **Rotación de textos:** la biblioteca de *Python* PIL añade ruido al rotar los textos, por lo que en ciertas ocasiones puede causar problemas de lectura en el archivo generado por la clase *CommandTracer*.

El día 17 de Abril del 2017 fueron entregadas, al cliente, las primeras gráficas generadas por el sistema.

5.1.4 Fase 4: Unión y UAV. 18 de Abril del 2017 - 30 de Junio del 2017

Esta fase está dividida en dos procesos que fueron realizados paralelamente: se inicio la **documentación de este proyecto** para informar de los resultados y se incorporaron los UAV al sistema.

Para agilizar la documentación se hizo uso de la biblioteca *esi-tfg* de LaTeX y se dividió la documentación en capítulos.

En el proceso de unión del UAV con el sistema se opto inicialmente por utilizar el SDK de *Parrot* pero una incompatibilidad con el dispositivo de comunicaciones del UAV hizo que se optara por utilizar *Node.js*.

Uno de los mayores problemas de esta fase fue el dispositivo de BLE necesario para la comunicación UAV-PC ya que la biblioteca *node-bluetooth-hci-socket* de *Node.js* solo acepta una serie de modelos BLE (véase Cuadro 5.2 y Cuadro 5.3).

Para la realización de este proyecto se seleccionó uno del modelo *CSR8510 A10*.

Una vez solucionado el problema con el BLE se continuó el desarrollo de la API REST en *Nodejs* y *Javascript*.

Esto supuso otro problema, que repercute en el funcionamiento del sistema de subastas y que no puede ser corregido. Este problema esta generado por el uso de la biblioteca *node-rolling-spider* de *Node.js* y está relacionado con la obtención del nivel de batería del UAV.

Name	USB VID	USB PID
BCM920702 Bluetooth 4.0	0x0a5c	0x21e8
BCM20702A0 Bluetooth 4.0	0x19ff	0x0239
BCM20702A0 Bluetooth 4.0	0x0489	0xe07a
CSR8510 A10	0x0a12	0x0001
Asus BT-400	0x0b05	0x17cb
Intel Wireless Bluetooth 6235	0x8087	0x07da
Intel Wireless Bluetooth 7260	0x8087	0x07dc
Intel Wireless Bluetooth 7265	0x8087	0x0a2a
Belkin BCM20702A0	0x050D	0x065A

Cuadro 5.2: Modelos BLE 4.0 aceptados por *node-bluetooth-hci-socket*

Name	USB VID	USB PID
BCM2045A0 Bluetooth 4.1	0x0a5c	0x6412

Cuadro 5.3: Modelos BLE 4.1 aceptados por *node-bluetooth-hci-socket*

El nivel de bateria del UAV queda fijado a $100\,\%$ durante los primeros segundos del uso del UAV hasta que este empieza a descargarse, por lo que durante las pruebas reales realizadas existe un pequeño problema durante las subasta.

Paralelamente, en el sistema, se implementaron los códigos necesarios para realizar las llamadas HTTP a la API REST creada.

La demo final fue realizada el día 23 de Junio de 2017. El sistema funcionó correctamente demostrando así que es posible coordinar distintos UAV mediante un sistema de pujas. En la primera demo final se realizó una subasta con 3 UAV los cuales levantaron el vuelo, simulando así la realización de una tarea, de forma ordenada y según su oferta era aceptada por parte del *buyer*.

Cabe destacar que los UAV, actualmente, solo alzan el vuelo simulando que realizan una tarea, dado que no es el objetivo de este proyecto el cómo realizan la tarea si no como deciden quién la realiza.

5.1.5 Evolución de las pruebas

En el desarrollo de este proyecto se han planteado dos pruebas evolutivas, aunque internamente cada prueba tiene sus sub-pruebas. En estas pruebas evolutivas se sigue una evolución lógica del problema empezando por una situación controlada.

En la primera prueba evolutiva se busca **implementar y poner a prueba el sistema de subastas concurrentes** realizando una pequeña demo que sirva como base del sistema. Lo que se busca comprobar con esta prueba evolutiva es ver la viabilidad del sistema al ser adaptado al uso de UAV. Esta prueba está sub-dividida en pequeñas pruebas que añaden, de forma incremental, complejidad al sistema. En esta prueba evolutiva no se implementara la parte de los UAV dado que este proyecto está enfocado más a la teoría de cómo funcionaría

un sistema de coordinación mediante subastas concurrentes entre UAV que en los UAV en sí, ya que serán controlados mediante un GCS en el PC. La parte relacionada con los UAV será implementada en la prueba evolutiva dos.

La prueba evolutiva dos se centrara en **unificar el sistema de subastas y el control de los UAV** realizando un sistema más orientado al uso de UAV y que utilizara los dato del UAV para configurar las negociaciones.

Este enfoque, de realizar inicialmente una **demo evolutiva** sin la intervención de los UAV para, una vez ya comprobado que el sistema de subastas funciona, pasar a implementar la demo con el funcionamiento de los UAV, permite:

- **Adaptar el proyecto** según las necesidades que vayan surgiendo durante las distintas fases.
- **Abstraerse y evitar problemas** relacionados con el uso de dispositivos HW.
- **Realizar simulaciones de coordinación** al realizar subastas con un mayor número de agentes, permitiendo así comprobar si se procesan de forma adecuada los mensajes.

En conclusión, la arquitectura modular planteada para la solución de este proyecto permite la realización de pequeños fragmentos que suman funcionalidad al sistema permitiendo así realizar una prueba por cada conjunto de funcionalidades implementadas.

Aunque, como contrapartida, la no inclusión de los UAV hasta la última prueba, repercute en la complejidad de la tarea implementada en la parte de los UAV debido a los posibles problemas en las fases del proyecto.

5.1.5.1. Prueba evolutiva 1: Demo incremental del sistema de subastas

Para llegar a cubrir la totalidad de la funcionalidad del sistema se han desarrollado sub-pruebas que irán evolucionando hasta llegar a implementar la funcionalidad total del sistema.

Se han definido tres sub-pruebas que cubrirán todas las pruebas iniciales. Estas se definen a continuación:

5.1.5.2. Sub-prueba 1: Demo inicial

En la sub-prueba de estudio 1 se realizó una **prueba de la demo inicial** con dos agentes, uno de tipo *seller* y otro de tipo *buyer* en un sistema de subastas básico donde los agentes compartían ofertas y contraofertas siguiendo distintas estrategias.

El problema que se plantea y que busca cubrir esta sub-prueba de estudio, es el siguiente: **certificar que la base del sistema de subastas ha sido definida correctamente** y que no surgen problemas adicionales relacionados con la elección del lenguaje ni con el sistema de comunicaciones elegido, también, además de comprobar que las estrategias y las lógicas de

los agentes son las correctas.

Se diseño una serie de pruebas para comprobar que la base del sistema funcionaba correctamente. Para cubrir todas las posibles opciones iniciales se han creado una serie de pruebas que comprenden la realización de una serie de subastas con los siguientes parámetros (véase Cuadro 5.4):

Nº	Estrategia	Buyer tmax	Seller tbmax	Reservation
1	Linear	10	>11	0.25
2	Conceder	10	>10	0.25
3	Tought	10	>10	0.25

Cuadro 5.4: Parámetros para las subastas realizadas en la sub-prueba 1

La finalidad de estas prueba es cubrir todos los posibles casos iniciales y comprobar que el sistema funciona para dos agentes y que las estrategias siguen la curva deseada.

Para la ejecución de esta solución se añadió al sistema la posibilidad de recibir argumentos por línea de comandos y se definió el siguiente comando:

```
$ cd SMACAUV
$ Python3 demo.py  [-h] [-p] [-l] [-c] [-t] numberOfUav buyerTMax reservation initialPenaltyFee finalPenaltyFee
      numberOfConcurrentCommitments
```

Esto permitió realizar distintas ejecuciones sin necesidad de cambiar variables en el código. Por ejemplo, para realizar la primera ejecución de esta sub-prueba invocaremos el sistema con el comando:

```
$ cd SMACAUV
$ Python3 demo.py  1 10 0.25 75 95 1 -p -l >output.txt
```

Una vez finalizada la ejecución el sistema generara tres archivos de salida:

- **output.txt:** en este archivo se encuentran los mensajes de estado que se han generado durante la subasta. En dichos mensajes puede verse el historial de ofertas y los cambios de estado de los agentes e hilos.
- **CommandTraceDiagram.png:** esta imagen representa las comunicaciones realizadas por los agentes. Este documento tiene una gran utilidad para los desarrolladores ya que se puede comprobar si las comunicaciones se están realizando correctamente.
- **UtilityHistoricDiagram.png:** en esta imagen puede verse la gráfica generada con el historial de ofertas de los agentes de tipo *seller* que han participado en la subasta. Sirve como comprobante de que se ha seleccionado la oferta con mayor utilidad. Cabe destacar que el valor de *utility* es calculado por parte del agente *buyer*.

Las conclusiones de esta sub-prueba han sido favorables al sistema. En las gráficas generadas (véase Figura 5.2, 5.3 y 5.4) se puede observar el resultado esperado.

- **En la primera ejecución** (véase Figura 5.2), con una estrategia de tipo *Linear*, se observa que el agente *seller* sigue una línea que incrementa en el tiempo, además el valor de *utility* es de 0.2591 superior al valor de *reservation* de 0.25.

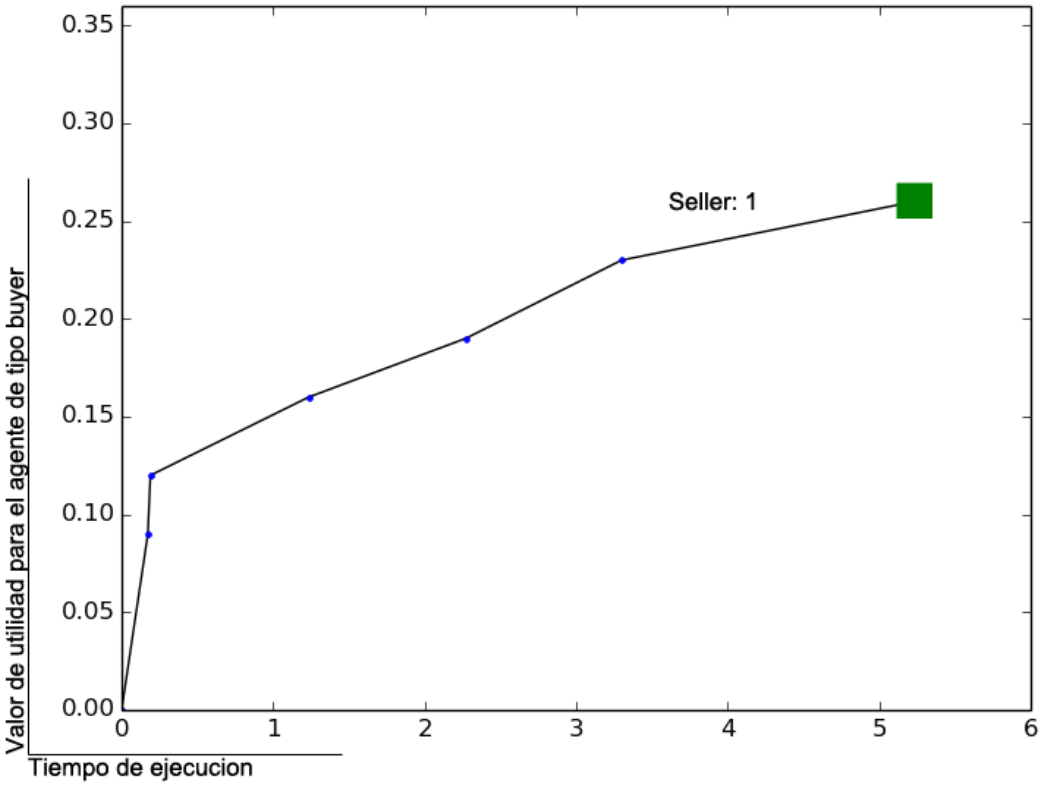

Figura 5.2: Sub-prueba 1: histórico de utilidad en la primera ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

- **En la segunda ejecución** (véase Figura 5.3), con una estrategia de tipo *Conceder*, se observa que el agente *seller* sigue una línea que incrementa en el tiempo, aunque como dictamina la estrategia de tipo *Conceder* esta línea es más pronunciada que con la estrategia de tipo *Linear*. También se puede observar que el valor de *utility*, al llegar a un acuerdo, es de 0.2617 superior al valor de *reservation* de 0.25 y que se ha llegado en el segundo 1.19.
- **En la tercera ejecución** (véase Figura 5.4), con una estrategia de tipo *Tough*, se observa que el agente *seller* no incrementa su oferta hasta llegado al momento correcto sigue una línea que incrementa en el tiempo. Como se puede observar, esta estrategia tardía ha hecho que no se llegue a ningún acuerdo por parte de los participantes ya que el valor de *utility* recibido en el segundo 8.6 es de 0.18, inferior al valor de *reservation* de 0.25. Cabe destacar que mirando los mensajes realizados durante la subasta, pode-

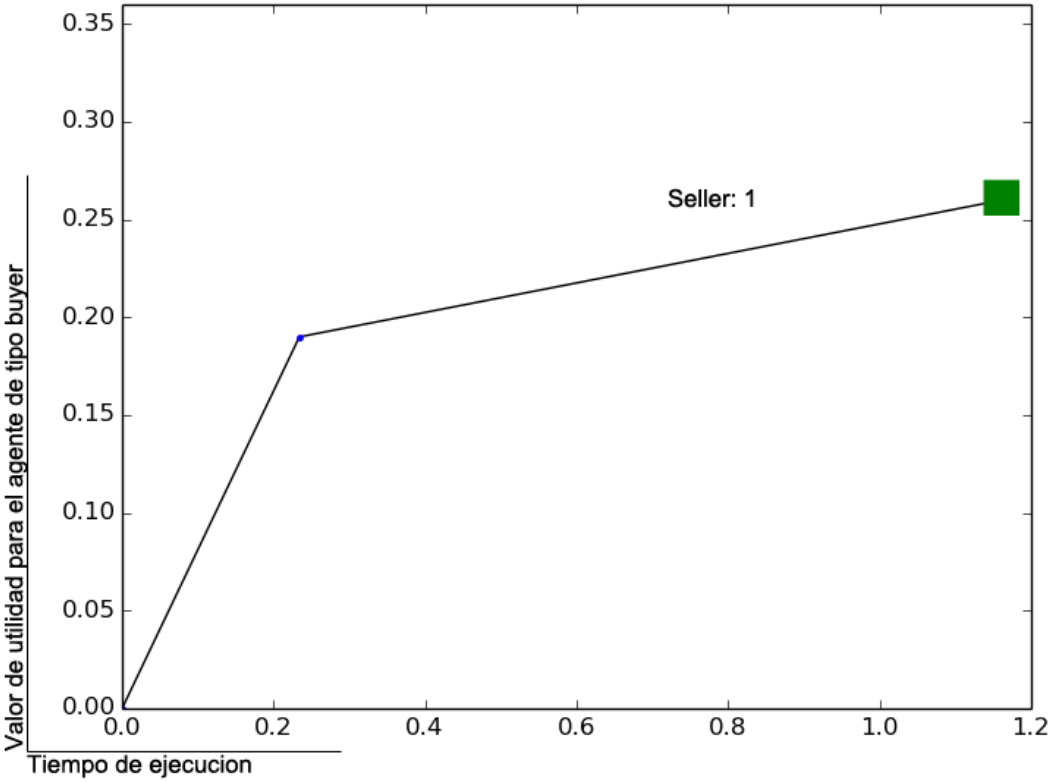

Figura 5.3: Sub-prueba 1: histórico de utilidad en la segunda ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

mos observar cómo, en el segundo 9.7 se recibió un valor de *utility* de 0.27, pero este mensaje no fue aceptado ya que el *Buyer* había iniciado el proceso de finalización de pujas.

Como se puede observar, el sistema de pujas y de estrategias funciona correctamente.

5.1.5.3. Sub-prueba 2: Demo multi-agente

En esta sub-prueba se incorporo el requisito de *coordinar múltiples agentes* añadiendo complejidad a la prueba evolutiva principal. En esta sub-prueba se realizó una **prueba de con múltiples agentes** de tipo *seller* compitiendo por un único contrato.

La dificultad del nuevo requisito que se busca solucionar con esta sub-prueba, es el siguiente: **certificar que el sistema puede coordinar múltiples agentes durante una subastas** y que no surgen problemas adicionales relacionados con la concurrencia y la comunicación de los distintos agentes involucrados.

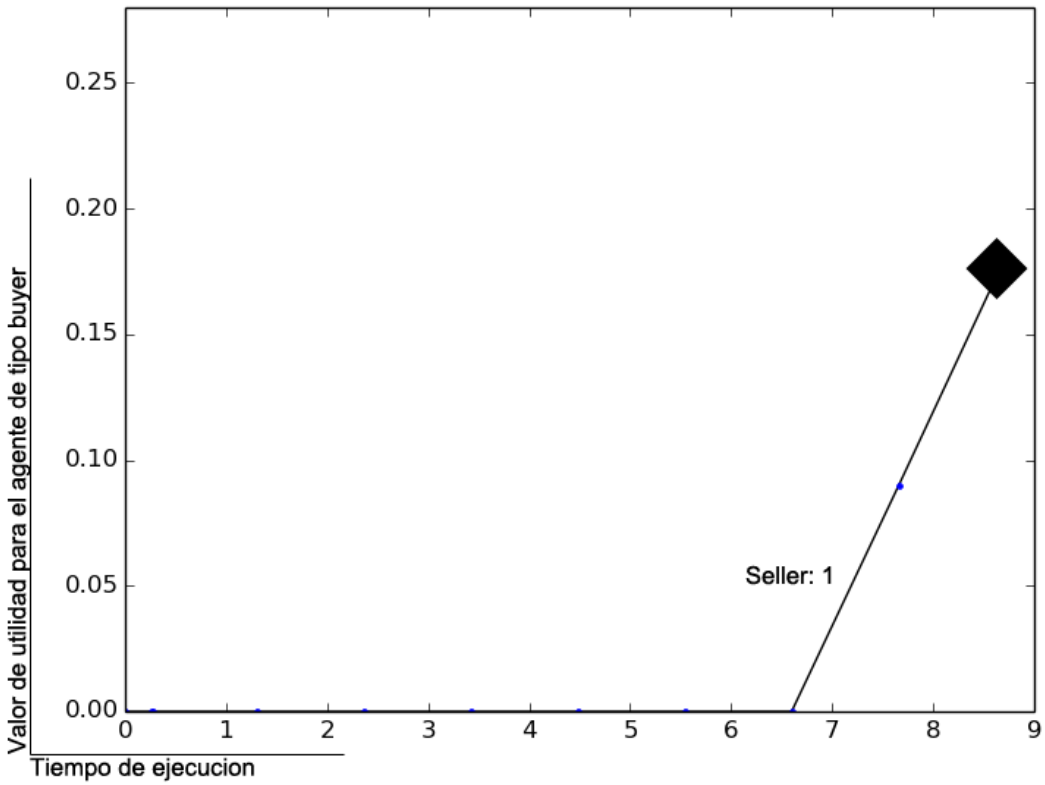

Figura 5.4: Sub-prueba 1: histórico de utilidad en la tercera ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

Se ha diseñado una serie de pruebas que añaden complejidad al sistema multi-agente. Se pretende ir cubriendo las necesidades exponencialmente. Para ello se han definido cuatro pruebas con distintos parámetros (véase Cuadro 5.5):

N°	**Estrategia**	**Buyer tmax**	**Seller tbmax**	**Reservation**	ρ_0	$\rho_{max} \geq \rho_0$	Ω
1	Mix	10	>10	0.25	75	95	4
2	All Linear	10	>10	0.25	75	95	4
3	All Linear	10	>10	0.25	10	15	4
4	All Linear	10	>10	0.25	10	15	1

Cuadro 5.5: Parámetros para las subastas realizadas en en la sub-prueba 2

- **Prueba inicial:** esta prueba consiste en determinar si el sistema puede manejar múltiples agentes que tengan distintas estrategias de forma concurrentemente.
- **Prueba de valor de penalización:** con esta prueba se busca comprobar si el valor de penalización está bien implementado. Todos los agentes usan la misma estrategia, pero el valor de penalización inicial y final es elevado.
- **Prueba de valor de penalización bajo:** una vez comprobado que el valor de penalización esta implementado, se busca determinar si, con valores bajos, el sistema añade un nuevo contrato a la lista.
- **Solo un contrato simultaneado:** con el sistema de penalización funcionando y la lista de acuerdos actualizándose, se busca determinar si el sistema es capaz de cancelar un contrato con un agente *seller*.

Para la ejecución de estas pruebas se definió un script Bourne again shell (BASH) que al ser ejecutado construía una estructura de ficheros y lanzaba las ejecuciones de las pruebas secuencialmente.

```
$ cd SMACAUV
$ ./Prueba_1.sh
```

Los resultados de ejecutar el script BASH serán almacenados en la carpeta *Prueba_1* que contendra los archivos descritos en en la sub-prueba 1 (véase § 5.1.5.2).

Gracias a estas pruebas se ha llegado a las siguientes conclusiones: primero, que el sistema es capaz de gestionar múltiples dispositivos de forma eficiente sin quedarse bloqueado, **segundo**, que las comunicaciones del sistema están bien implementadas en situaciones donde existe más de un agente y por **último**, que el sistema controla, de forma eficiente y precisa, la selección de nuevos contratos. En las gráficas generadas (véase Figura 5.5, 5.6, 5.7 y 5.8) se puede observar el resultado esperado.

- **En la primera prueba** (véase Figura 5.5) se observa que el resultado es el esperado. Dado que se ha implementado de forma eficiente la captación de mensajes y estados, la inclusión de múltiples agentes, no supone problema para el sistema.

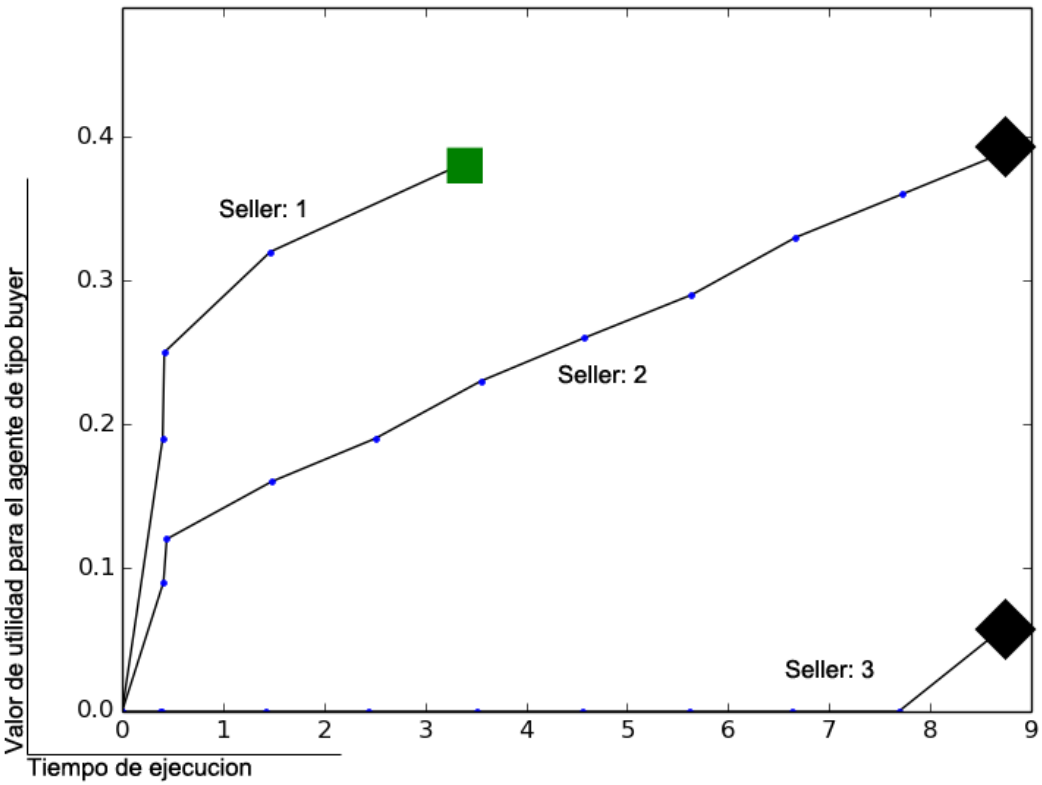

Figura 5.5: Sub-prueba 2: histórico de utilidad en la primera ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

- **En la segunda prueba** (véase Figura 5.6) se ha comprobado que el sistema maneja bien múltiples agentes con la misma estrategia. Se comprueba también que dado que la penalización actual es muy elevada no es aceptada un nuevo contrato por el sistema una vez ya ha aceptado el primero.
- **En la tercera prueba** (véase Figura 5.7) queda comprobado que la penalización está bien implementada ya que al bajarla se han aceptado dos contratos simultáneos. Cabe destacar que, como se ve en el archivo de mensajes *output.txt*, ha sido seleccionado el contrato con mayor valor de *utility* una vez finalizada la subasta.

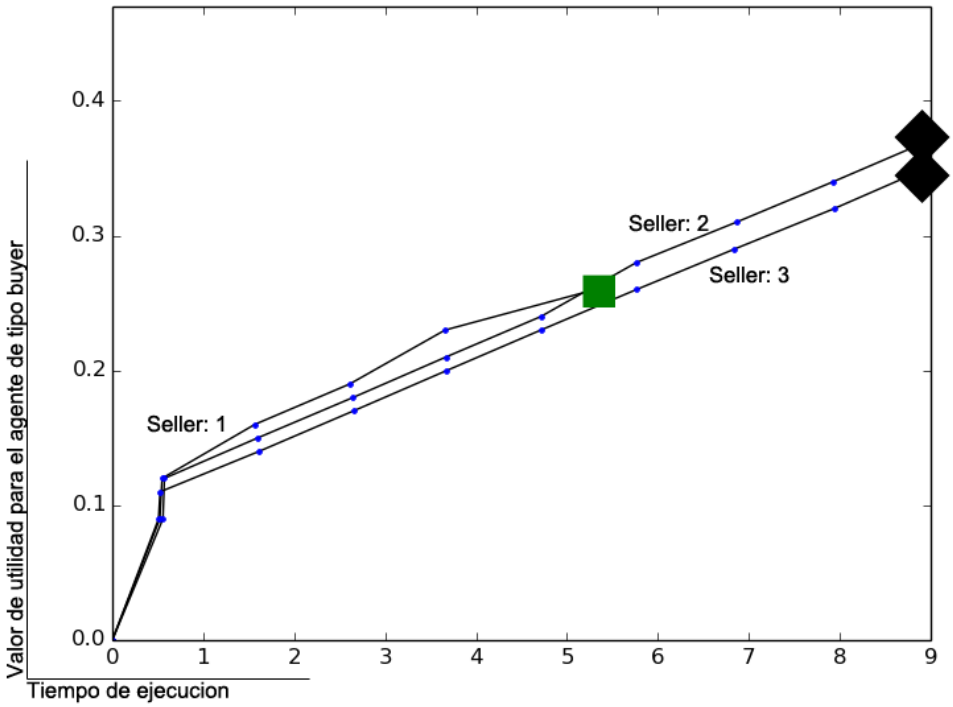

Figura 5.6: Sub-prueba 2: histórico de utilidad en la segunda ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

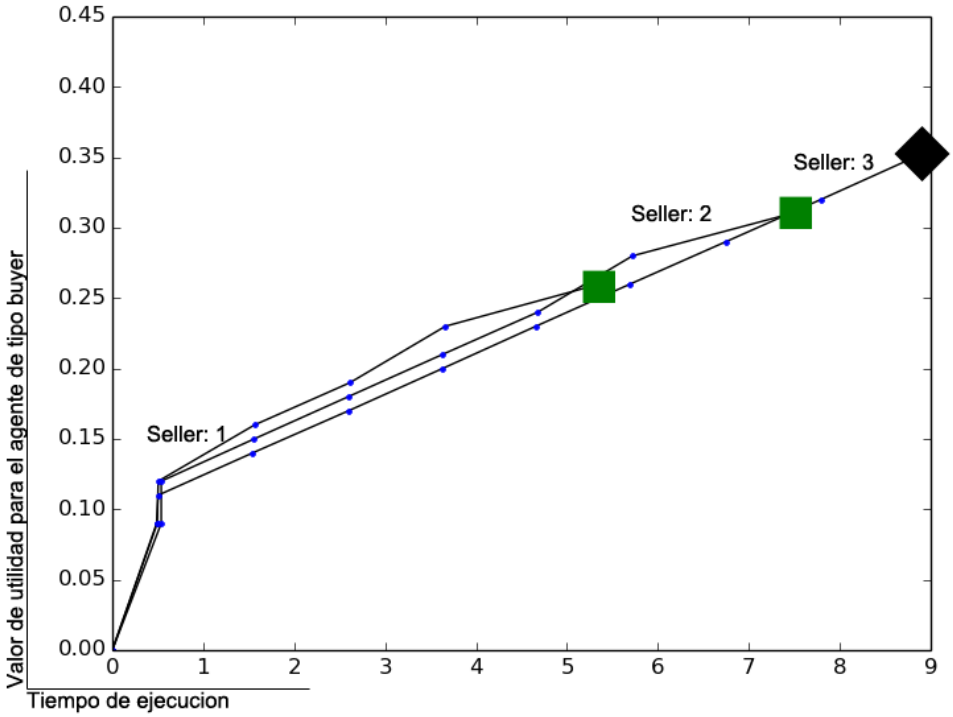

Figura 5.7: Sub-prueba 2: histórico de utilidad en la tercera ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

```
$ python3 demo.py 3 10 0.25 10 15 4 -l -p > output.txt
...
Commitment Manager Thread================================
Tmax: 10    Reservation value: 0.35219  Time: 13.048
Negotiations Accepted Bids:-----------------------------
0 - Negotiation Accepted ID= 0
Negotiation thread id: 0----------------------
New Bid from Seller Message:[Money:87.043 at 5.36]Utility:0.2591382575757575
-----------------------------------------------------
1 - Negotiation Accepted ID= 1
Negotiation thread id: 1----------------------
New Bid from Seller Message:[Money:84.688 at 7.462]Utility:0.30625
-----------------------------------------------------
Negotiation Status:-------------------------------------
-Negotiation ID: 0   Status: Contract finalizado
-Negotiation ID: 1   Status: Contract finalizado
-Negotiation ID: 2   Status: Negociaion cerrada por parte de del comprador
-----------------------------------------------------
=====================================================
++CLOSE++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Commitment Manager Thread================================
Tmax: 10    Reservation value: 0.35219  Time: 14.061
Negotiations Accepted Bids:-----------------------------
0 - Negotiation Accepted ID= 1
Negotiation thread id: 1----------------------
New Bid from Seller Message:[Money:84.688 at 7.462]Utility:0.30625
```

```
-----------------------------------------------------
Negotiation Status:-------------------------------------
-Negotiation ID: 0   Status: Contract cancelado por parte del comprador
-Negotiation ID: 1   Status: Contract finalizado
-Negotiation ID: 2   Status: Negociaion cerrada por parte de del comprador
-----------------------------------------------------
=====================================================
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

- **En la cuarta prueba** (véase Figura 5.8) se ha podido comprobar que, en caso de necesidad, el sistema es capaz de cancelar un contrato con un agente. Esta prueba demuestra que se ha implementado bien el requisito de controlar, para una mayor eficiencia, el número de contratos simultaneos.

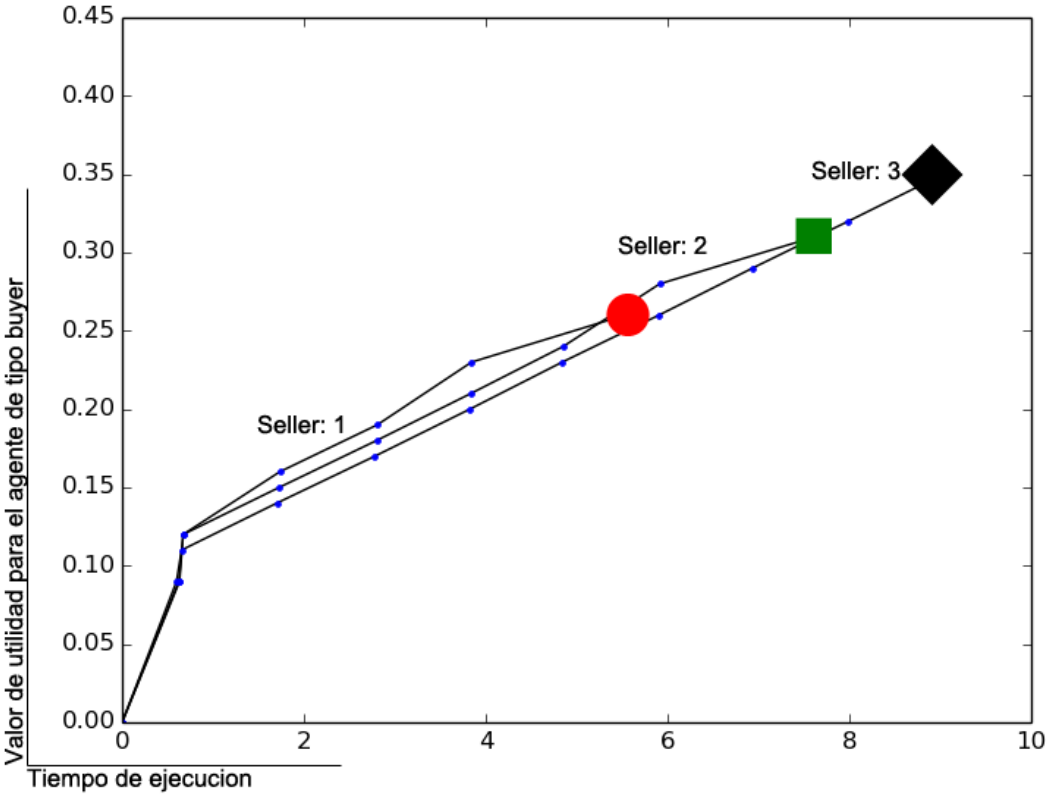

Figura 5.8: Sub-prueba 2: histórico de utilidad en la cuarta ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

5.1.5.4. Sub-prueba 3: Cancelación por parte de los agentes de tipo *seller*

Para esta sub-prueba, se ha añadido la capacidad para asimilar, por parte del sistema, que los agentes de tipo *seller* puedan cancelar sus contratos con el *buyer* dotandoles de la capacidad de mantener subastas simultaneas con distintos agentes de tipo *buyer*. En esta sub-

prueba se realizó una **prueba de cancelacion por parte de un agente de tipo *seller***.

La dificultad de esta nueva capacidad reside en controlar los mensajes enviados por parte del agente de tipo *seller* y gestionarlos en la parte del agente *buyer*. Se busca añadir, con esta nueva capacidad, la habilidad de toma de decisiones por parte del agente de tipo *seller*. Esta sub-prueba, busca: **certificar que el sistema puede controlar la toma de decisiones de los agentes** y que no surgen problemas adicionales relacionados con la eliminación de un parcipante de la subasta, como los derivados de la comunicación.

Se ha diseñado una prueba donde los agentes poseen tiempos sumamente diferentes, incurriendo así en la posibilidad de que el $t_{b_{max}}$ de un agente de tipo *seller* sea inferior al t_{max}, del agente de tipo *buyer*, situación en la cual, el agente de tipo *seller* podría dejar la subasta incluso teniendo un contrato parcial con el agente de tipo *buyer*.

Para la ejecución de esta prueba se lanzaran cuatro agentes de tipo *seller*, dos de los cuales tendrán un valor de $t_{b_{max}}$ inferior a t_{max}. Para esto se ha modificado la sección de código encargada de configurar el tiempo del que disponen los UAV. Cabe mencionar también que se ha ejecutado el código de forma que el agente de tipo *buyer* busque la realización de un servicio parcial[1], con el parámetro textit-ps, dado que este tipo de servicios puede ser realizado por todo tipo de UAV mientras que los servicios parciales requieren que el UAV disponga de suficiente batería para realizar la tarea en su totalidad, o lo que es lo mismo un $t_{b_{max}}$ superior a t_{max}.

El comando ejecutado para la ejecución de esta prueba es:

```
$ cd SMACAUV
$ python3 demo.py 4 10 0.25 10 15 2 -p -l -ps > output.txt
```

Gracias a la realización de esta prueba se ha demostrado que el sistema acepta bien la cancelación de contratos por parte de los agentes de tipo *seller*. Se ha podido comprobar que el sistema no se queda bloqueado ante la desaparición de un participante y que las comunicaciones se mantienen estables y permiten la finalización de un contrato entre participantes.

En el diagrama generado tras la ejecución del comando (véase Figura 5.9) se puede observar cómo se han realizado tres contratos parciales (cuadrado verde, círculo rojo y estrella morada) y que uno de ellos ha sido cancelado por parte del agente participante de tipo *seller* (estrella morada). El otro agente no ha consumado contrato alguno (diamante negro). Si se lee el informe realizado en la ejecución se puede ver también dicho resultado.

```
$ python3 demo.py 4 10 0.25 10 15 2 -p -l -ps
```

[1]Los servicios están divididos en **parciales** y en **completos**. Un servicio *parcial* representa un servicio donde no importa si se realiza la acción por un UAV o por varios ya que el servicio en si requiere de un largo periodo de actividad. Un servicio completo implica que solo un UAV adquiere el contrato y suelen ser realizados en un periodo de tiempo inferior

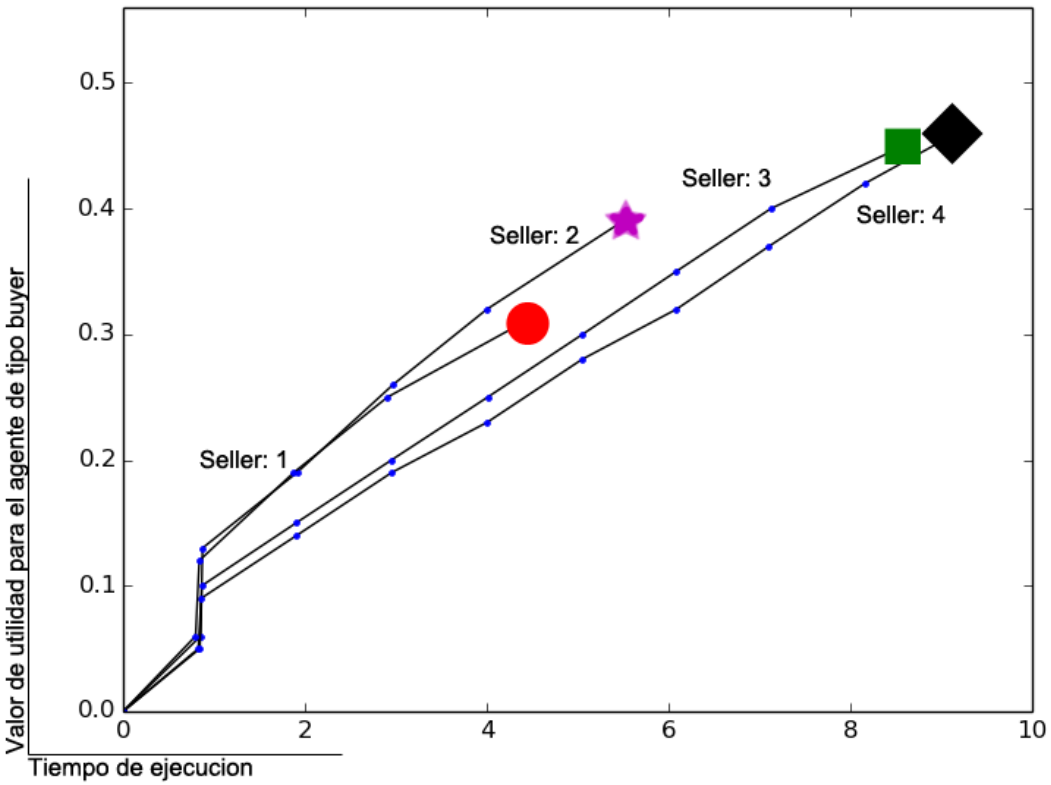

Figura 5.9: Sub-prueba 3: histórico de utilidad de la ejecución del sistema de subastas.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

```
...
++CLOSE++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Seller:0==================================================
Strategy:
--Strategy Linear:0-------------------------------------
--Deadline= 10.0    reservation= 0.35
----------------------------------------------------------
UAV Charge: 80  Status: Cerrado Time: 7.119
Last bid receive: [Money:69.345 at 6.007]
Last bid sent: [Money:80.5 at 5.498]
==========================================================
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
...
++CLOSE++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Commitment Manager Thread================================
Tmax: 10    Reservation value: 0.5175   Time: 13.033
Negotiations Accepted Bids:------------------------------
0 - Negotiation Accepted ID= 1
Negotiation thread id: 1----------------------
New Bid from Seller Message:[Money:77.5 at 8.588]Utility:0.44999999999999996
----------------------------------------------------------
Negotiation Status:-------------------------------------
-Negotiation ID: 0   Status: Contract cancelado por parte del vendedor
```

```
-Negotiation ID: 1   Status: Contract finalizado
-Negotiation ID: 2   Status: Negociaion cerrada por parte de del comprador
-Negotiation ID: 3   Status: Contract cancelado por parte del comprador
------------------------------------------------------
======================================================
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

5.1.5.5. Conclusiones de la prueba evolutiva 1

Una vez realizadas todos estas sub-pruebas evolutivas se puede llegar a la conclusión de que el sistema es fiable y que la inclusión de los UAV no deberá suponer ningún problema adicional, aparte de los que el propio HW pueda causar. También cabe destacar que la implementación del modulo de análisis forense de los datos ha ayudado mucho a la comprobación de las sub-pruebas aportando informes y gráficas con datos relevantes a la ejecución del sistema.

La división del caso en pequeñas sub-pruebas ha permitido arreglar pequeños fallos o *bugs* en el sistema de forma rápida y sencilla y sin llegar a tener problemas grandes en ningún momento.

5.1.5.6. Prueba evolutiva 2: Conexion con el UAV

En esta prueba evolutiva se busca comprobar si el **uso de los UAV y sus características en el sistema es factible**. Para ello se han definido unas pruebas que permitirán comprobar la funcionalidad de las partes añadidas al sistema, como la conectividad de los UAV y la gestión de la batería de los UAV.

Se dispone de cuatro UAV *Parrot* del modelo *Cargo* y de seis baterías para la realización de esta prueba.

Se han definido dos sub-pruebas, una para cada prueba, que serán descritos a continuación:

5.1.5.7. Sub-prueba 1: prueba de conexión de UAV

En esta sub-prueba se pretende realizar una **prueba de conexión entre el sistema de subastas y el UAV** mediante la implementación de una API REST.

El problema planteando en esta sub-prueba es *certificar que es factible la conexión con el* UAV *a través del sistema de subastas* y que no surgen problemas derivados del uso del HW o de bloqueos en el sistema causados por el uso de un dispositivo externo.

Se han diseñado dos pruebas que buscan certificar que la conexión se realiza de forma correcta y que el UAV recibe la información del sistema y que el UAV informa al sistema con los datos pertinentes. Para ello se van a realizar tres pruebas:

- **Prueba de conexión a la API REST:** en ella se ejecutara un script BASH que testeara todos los puntos de entrada definidos en la API REST.
- **Prueba de conexión múltiple a la API REST:** en esta prueba se realizara la prueba anterior pero añadiendo el control de más de un UAV
- **Prueba de conexión desde el sistema de subastas:** en esta prueba las llamadas a la API REST se realizaran desde el sistema comprobando así que es capaz de activar los UAV sin bloquearse.

Para la ejecución de las dos primeras pruebas se ha implementado un script BASH que realizara las llamadas correspondientes a la API REST mediante la aplicación *curl*[2] capaz de enviar datos en formato JSON a una dirección HTTP.

La primera ejecución controlara un único dispositivo mientras que la segunda conectara dos dispositivos simultáneamente. Para realizar estas pruebas es necesario ejecutar inicialmente el servidor API REST con todas las dependencias ya instaladas y disponer de un dispositivo BLE compatible y de un UAV de *Parrot* modelo *Cargo*. Para la ejecución del servidor se utilizara el comando:

```
$ cd rest
$ node server.js
Restful API server started on: 30000
```

Una vez lanzado el servidor se pueden realizar las pruebas con:

```
$ cd SMACAUV
$ ./single_UAV_test.sh > single_UAV_test.output.txt
$ ./multi_UAV_test.sh > multi_UAV_test.output.txt
```

Debido a estas dos pruebas se ha demostrado que la API REST implementada para el control de los UAV funciona correctamente y que como podemos observar, tanto en los datos recibidos en el servidor como en los de los scripts BASH realizados, se han recibido bien los datos del *Area* a vigilar y el estado de la *Battery* del UAV. A continuación se muestra la salida de la ejecución del *single_UAV_test.sh* y del servidor API REST:

```
$ cat ./Casos_de_estudio/SMACAUV/Prueba_2/single_UAV_text.output.txt
UAV Connect
{"Status":"OK"}
UAV Start
{"Start":"OK"}
UAV Status
{"Status":"OK","Battery":100}
UAV Stop
{"Stop":"OK"}
```

[2] https://curl.haxx.se/

```
$ cat ./Casos_de_estudio/Rest/Prueba_2/single_output.txt
RESTful API server started on: 30000
CONNECT...
{ UUID: 'Mars_145684' }
creando dron UUID:Mars_145684
Configured for Rolling Spider!  Mars_145684
ready for flight
START...
{ UUID: 'Mars_145684',
  Area: [ { x: 1, y: 2 }, { x: 3, y: 4 } ] }
START dron UUID:Mars_145684
takeOff...
{ UUID: 'Mars_145684' }
STATUS dron UUID:Mars_145684
Battery: 100%
Signal: -54dBm
{ UUID: 'Mars_145684' }
STOP dron UUID:Mars_145684
Pre landing...
Initiated Landing Sequence...
disconnect...
```

Para la ejecución de la última prueba se ampliara el código del sistema para que se conecte a la API REST que controla los UAV para comprobar que no se realizan bloqueos en el sistema. Se realizara una prueba con tres UAV y se simulara la descarga de batería. Para esta prueba se ejecutara el siguiente comando:

```
$ cd SMACAUV
$ python3 demo.py 3 100 0.15 10 15 2 -p -l -ps > output.txt
```

Dado que el agente de tipo *buyer* busca un servicio *parcial* se espera comprobar que los UAV inician el vuelo según obtienen el recurso subastado.

El resultado de esta prueba nos lleva a la conclusión de que el sistema es robusto y que permite controlar distintos UAV y coordinarlos entre sí para que realicen un proceso de subasta para adquirir un contrato de servicio.

En el grafico de valores de *utility* podemos ver como (véase Figura 5.10) se han ido seleccionando todos los agentes para realizar el servicio. Los valores de *utility* varían desde $0{,}15$ hasta $0{,}21$.

5.1.5.8. Sub-prueba estudio 2: baterías. Prueba realista con los UAV

En esta sub-prueba se pretende realizar una **prueba más fiel a la realidad** donde los agentes de tipo *seller* obtengan de los UAV su nivel de batería y así usarlo para gestionar su estrategia. Dicho valor de batería será obtenido a partir de la API REST.

El nuevo requisito planteando en esta sub-prueba es *utilizar el valor real de la carga de batería de los* UAV *obteniéndola a través de la acsAPI* REST *para utilizar dicho valor en el sistema de subastas* y que no surgen problemas derivados del uso de la obtención de los

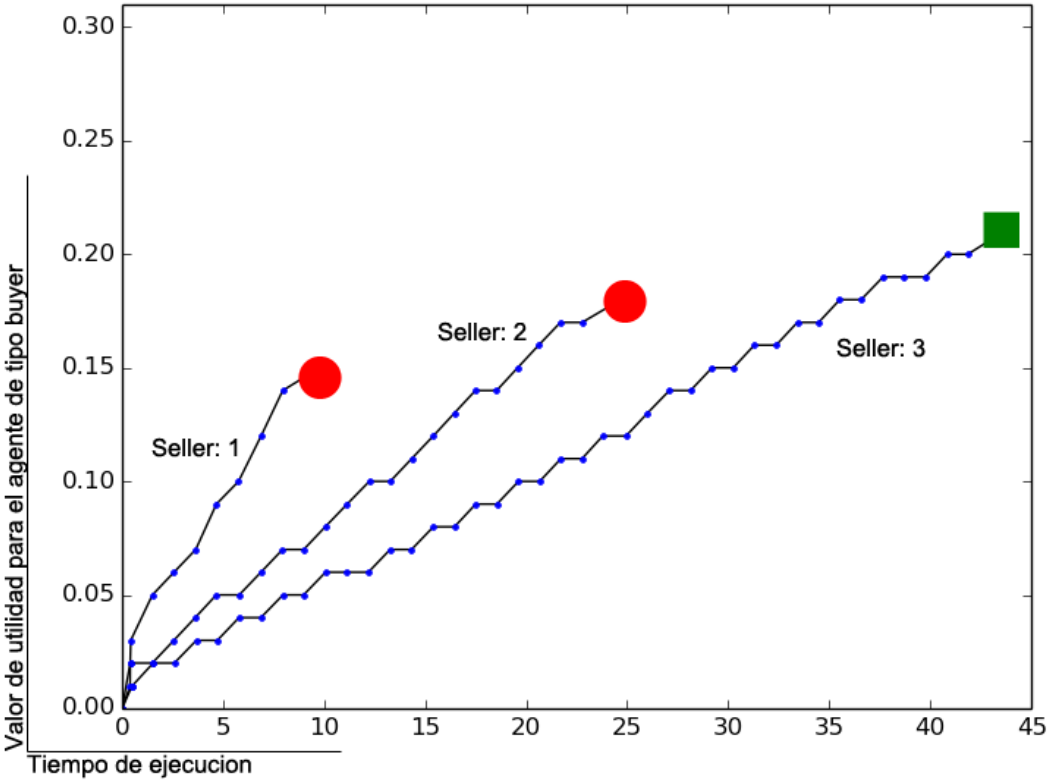

Figura 5.10: Sub-prueba 1: histórico de utilidad de la ejecución del sistema de subastas con UAV.Cuadrado verde - Activo. Círculo rojo - Cancelado. Diamante negro - finalizada. Estrella morada - Cancelado por parte del vendedor.

niveles de carga del UAV.

Se han diseñado dos pruebas que buscan certificar que los niveles de batería obtenidos son reales y que afectan al sistema. Para ello se van a ejecutar dos pruebas con un solo UAV del cual se sabe la carga actual (recién cargado, medio uso y casi agotada).

Para la ejecución de las dos pruebas se ha modificado el código del sistema a su ultima versión. Se ha eliminado la simulación de la descarga de batería y se ha añadido la parte de código que actualiza dicho valor obteniéndolo desde la API REST. El nivel de batería mínimo permito para el UAV ha sido fijado a un 15 %. Se ejecutara el siguiente comando para lanzar las pruebas:

```
$ cd SMACAUV
$ python3 demo.py 1 50 0.15 10 15 2 -p -l -ps > output.txt
```

Estas pruebas se harán bajo el entendimiento de que la biblioteca *node-rolling-spider* de *Node.js* falla, los primeros segundos, al actualizar el nivel de batería del UAV.

Debido a estas pruebas se ha demostrado que el sistema acepta sin problemas el cambio

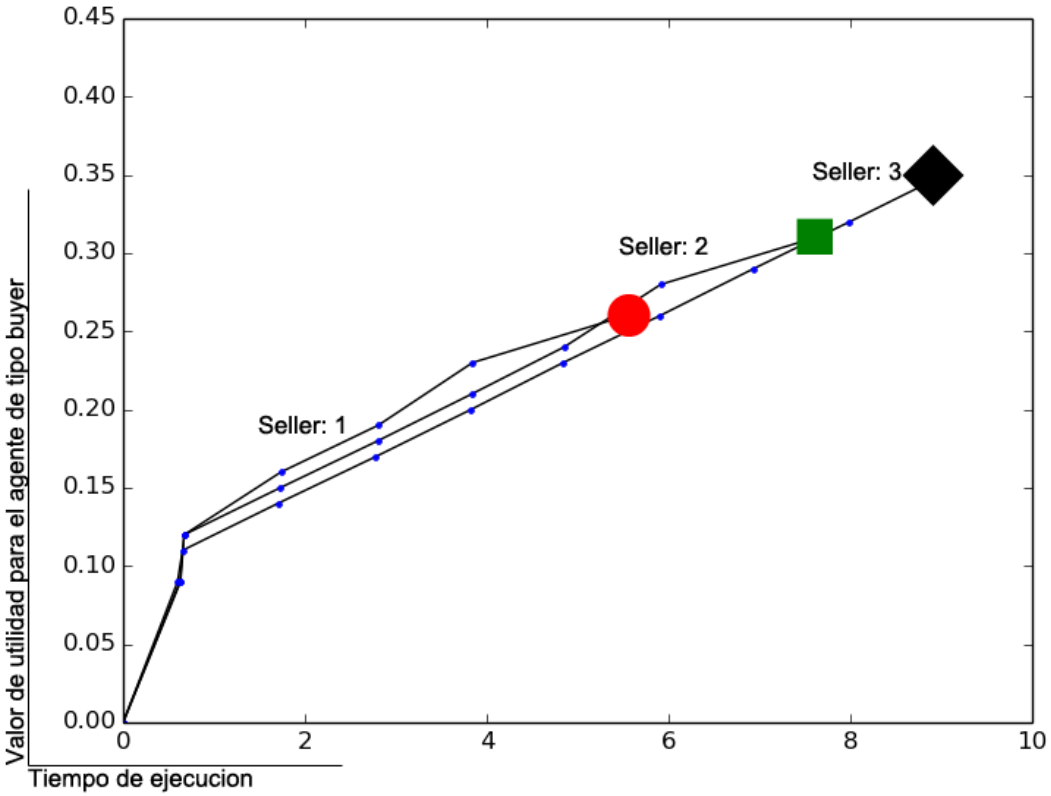

Figura 5.11: Distintas capturas de la ejecución de la subasta por parte de los UAV

en la gestión de la batería. Esto es gracias al uso de una arquitectura modular y una buena definición de las funciones de los agentes. Se ha podido modificar el código relacionado con la obtención del nivel de batería sin problema alguno.

Como se puede observar en los diagramas generados tras las pruebas como (véase Figura 5.12, 5.13, y 5.14) las ofertas realizadas por parte del agente de tipo *seller* están relacionadas con su valor real de carga de batería. También, podemos observar que en la prueba con la batería casi agotada el agente de tipo *seller* ha abandonado la subasta ya que su nivel de batería ha descendido por debajo de los límites permitidos.

5.1.5.9. Conclusiones del prueba evolutiva 2

Una vez realizadas todos estas sub-pruebas evolutivas se puede llegar a la conclusión de que el sistema puede ser utilizado para coordinar distintos UAV en situaciones reales dado que el control del dispositivo y la obtención de sus características puede ser llevado a cabo por el sistema. También cabe destacar que la arquitectura modular y la definición de unas clases y funciones bien definidas y separadas ha permitido modificar el código e incluir el uso de los UAV de forma rápida y sencilla.

Como ya paso en la prueba evolutiva anterior, la división del caso en pequeñas sub-pruebas

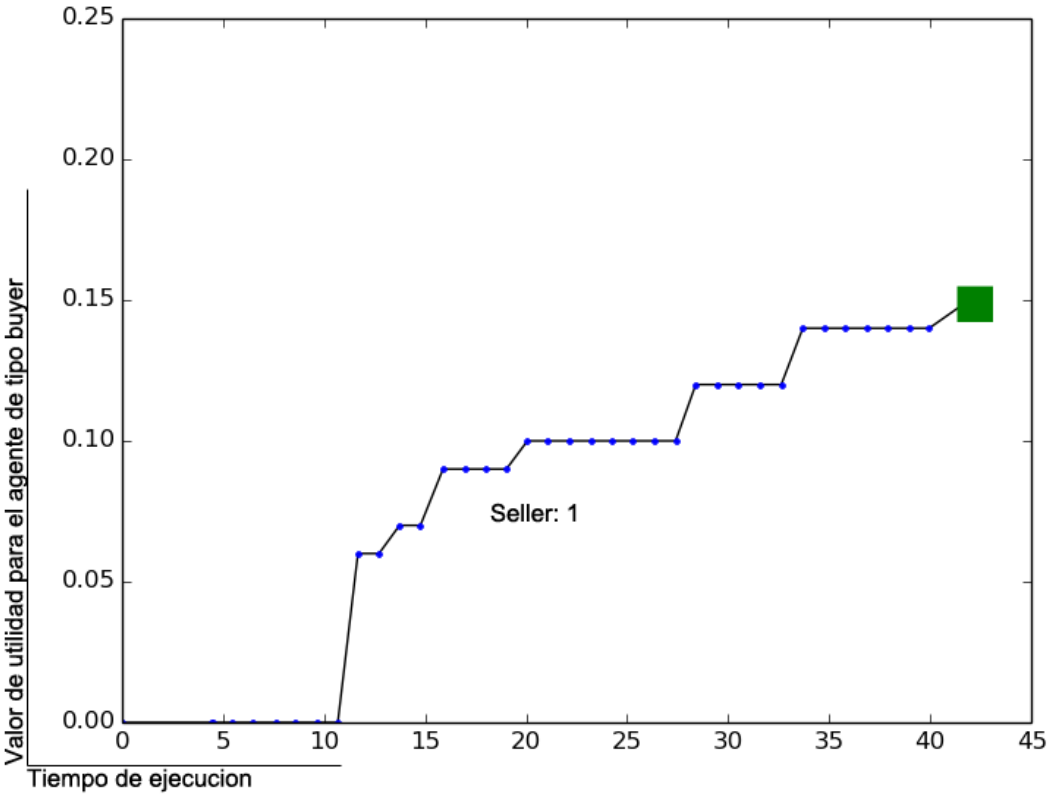

Figura 5.12: Prueba de ejecución del sistema de subastas con un UAV con la batería completa

ha permitido arreglar pequeños fallos o *bugs* en el sistema de forma rápida y sencilla y sin llegar a tener problemas grandes en ningún momento. Por eso, cabe destacar, que el enfoque llevado durante las pruebas ha sido el correcto.

5.2 Caso de estudio

Como prueba final se ha propuesto un caso de estudio donde se pretende probar el sistema en un entorno lo más real posible para así estudiar su comportamiento y poder sacar conclusiones que se asemejen a la realidad.

Dado que el desarrollo del sistema ha sido realizado mediante pruebas evolutivas, se sabe, a ciencia cierta, que el sistema realiza de forma correcta subastas donde intervienen dispositivos UAVs físicos y no simulados.

Por eso, para este caso de estudio, **se plantea el siguiente problema:** siguiendo la definición escrita en la ontología del sistema, un **buyer o comprador** quiere **contratar** un **servicio de video vigilancia** continua, en un **área** determinada, durante un tiempo determinado. Dicho **buyer busca** quién le **ofrezca el servicio** en unas **páginas amarillas**.

Para solucionar dicho problema **se ha diseñado una solución** que dada la capacidad de

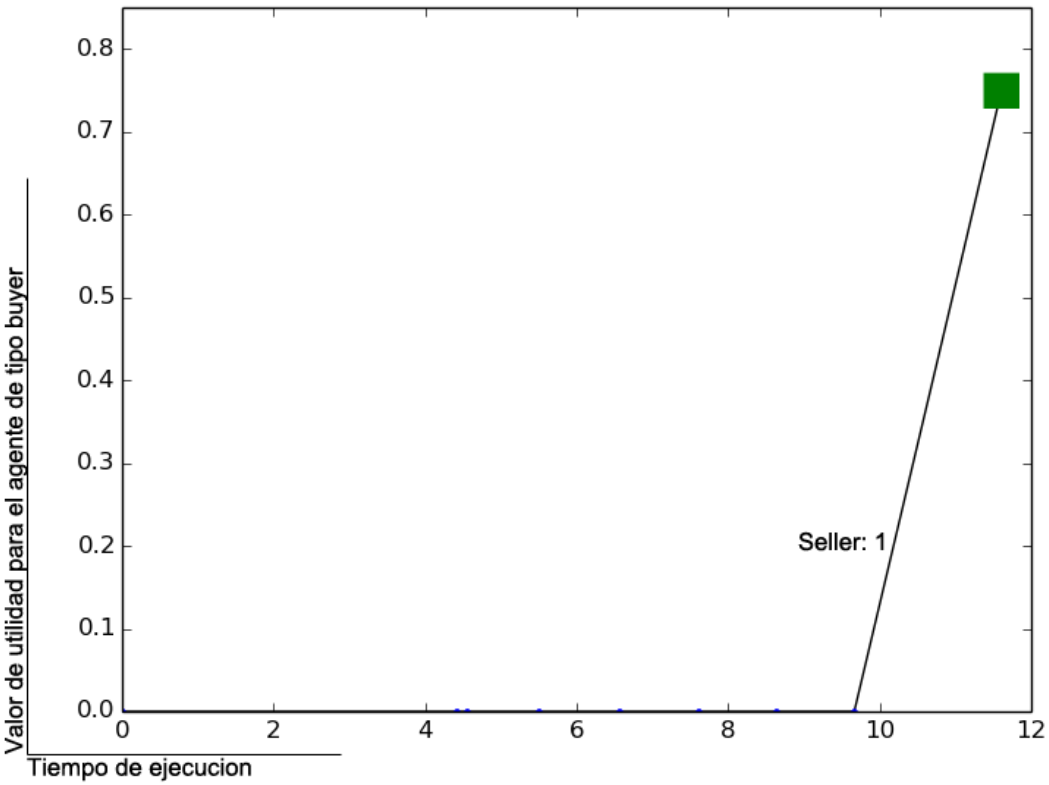

Figura 5.13: Prueba de ejecución del sistema de subastas con un UAV con la a media carga completa

tiempo de vuelo de los UAV del modelo *Cargo* de la marca *Parrot*, que oscila entre 3 y 4 minutos, se debe realizar un caso de estudio a escala, y que, aunque los tiempos puedan parecer irreales para el problema planteado, los resultados pueden ser extrapolados a una situación real. Para esta solución se utilizaran los siguientes recursos:

- **UAV *Parrot Cargo*:** 3 dispositivos con cargas varias.
- **Cámaras de video:** Para obtención de video frontal, lateral trasero y cenital.

Otros datos relacionados con el caso de estudio son:

- El tiempo de vigilancia se acotara a **60 segundos.**
- El **área** será definida como **A.**
- El **comprador** buscara un **servicio de video vigilancia en el área A.**
- **Los tres UAVs** se registraran para **realizar un servicio en el *area* A.**
- Los **UAVs** una vez tengan el control del **contrato** empezaran a realizar un **movimiento en forma de cuadrado**, de forma cíclica, **simulando la vigilancia** del perímetro definido por los puntos que han recibido al aceptar el contrato.

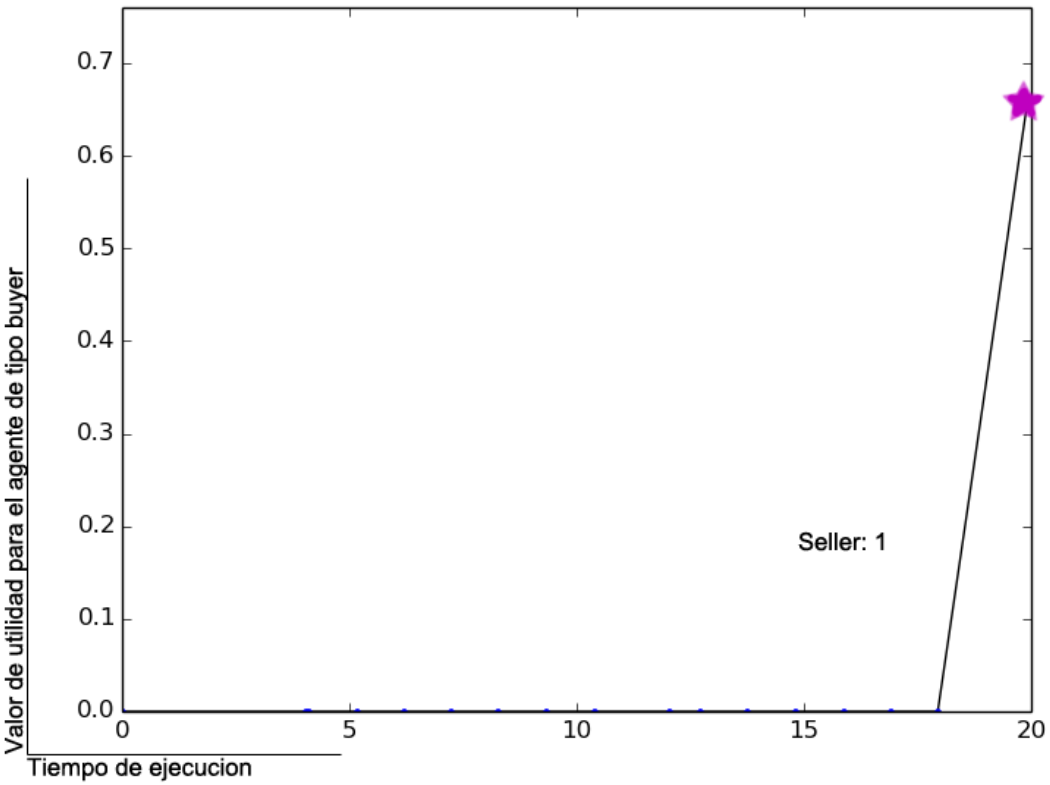

Figura 5.14: Prueba de ejecución del sistema de subastas con un UAV con la batería casi agotada

Para la ejecución de esta solución se dispondrá de un PC portátil con un dispositivo BLE con un SO *Ubuntu*. Se han utilizado tres UAVs por cuestiones de estabilidad y espacio. Se ha reducido el tiempo de subasta para minimizar el porcentaje de fallo debido a la poca estabilidad de los UAVs. El tiempo de subasta es suficiente para realizar una cantidad de ofertas y contraofertas elevada.

El comando utilizado para lanzar el sistema de subastas es el siguiente:

```
$ cd SMACAUV
$ python3 demo.py 3 60 0.15 5 10 2 -p -l -ps > output.txt
```

Para el servicio de API REST se utilizara el siguiente comando:

```
$ cd Rest
$ node server.js > server.output.txt
```

Tras la realización del caso de estudio (véase Figura 5.15) se han obtenido una serie de informes y tres videos, desde distintos ángulos donde puede verse el sistema en funcionamiento. Gracias a toda esta documentación y a las imágenes obtenidas se puede demostrar que el sistema funciona en un entorno real y que los UAVs realizan sus ofertas y contraofertas.

Figura 5.15: Captura de la realización del caso de estudio. En la foto puede verse al desarrollador utilizando el PC, el trípode con la cámara utilizada para grabar la escena desde un plano frontal y a los UAV volando.

Cabe mencionar que han surgido problemas relacionados con los niveles de baterías de los UAVs pero han sido solventados.

Como podemos ver en las gráficas (véase Figura 5.16), informes y los videos, uno de los UAVs, el UAV con el **ID:0** nada más iniciar la subasta decide marcharse ya que su nivel de batería no es el optimo. Los UAVs con **ID:1** y **ID:2** continúan la subasta con el agente de tipo *buyer*. El UAV con *ID:1* gana el recurso (cuadrado verde del histórico de utilidades. Véase Figura 5.17) dado que su nivel de batería es inferior por lo que el valor de *utility* $(0{,}25)$ que obtiene el agente de tipo *buyer* es superior al que obtiene del UAV con *ID:2* $(0{,}20)$. Una vez ganado el recurso por parte del UAV con *ID:1* este empieza a realizar el servicio contratado, como se puede ver en la Figura 5.16 , Aunque la penalización está fijada un valor de $5\,\%$ de inicial y un $10\,\%$ de final, dado que el nivel de batería del UAV con *ID:2* no decrementa drásticamente el valor de *utility* que obtiene al final el agente de tipo *buyer* no supera $0{,}26$ (rombo negro del histórico de utilidades. Véase Figura 5.17). Una vez finalizada la subasta los UAV descienden dando por finalizada la tarea y la subasta.

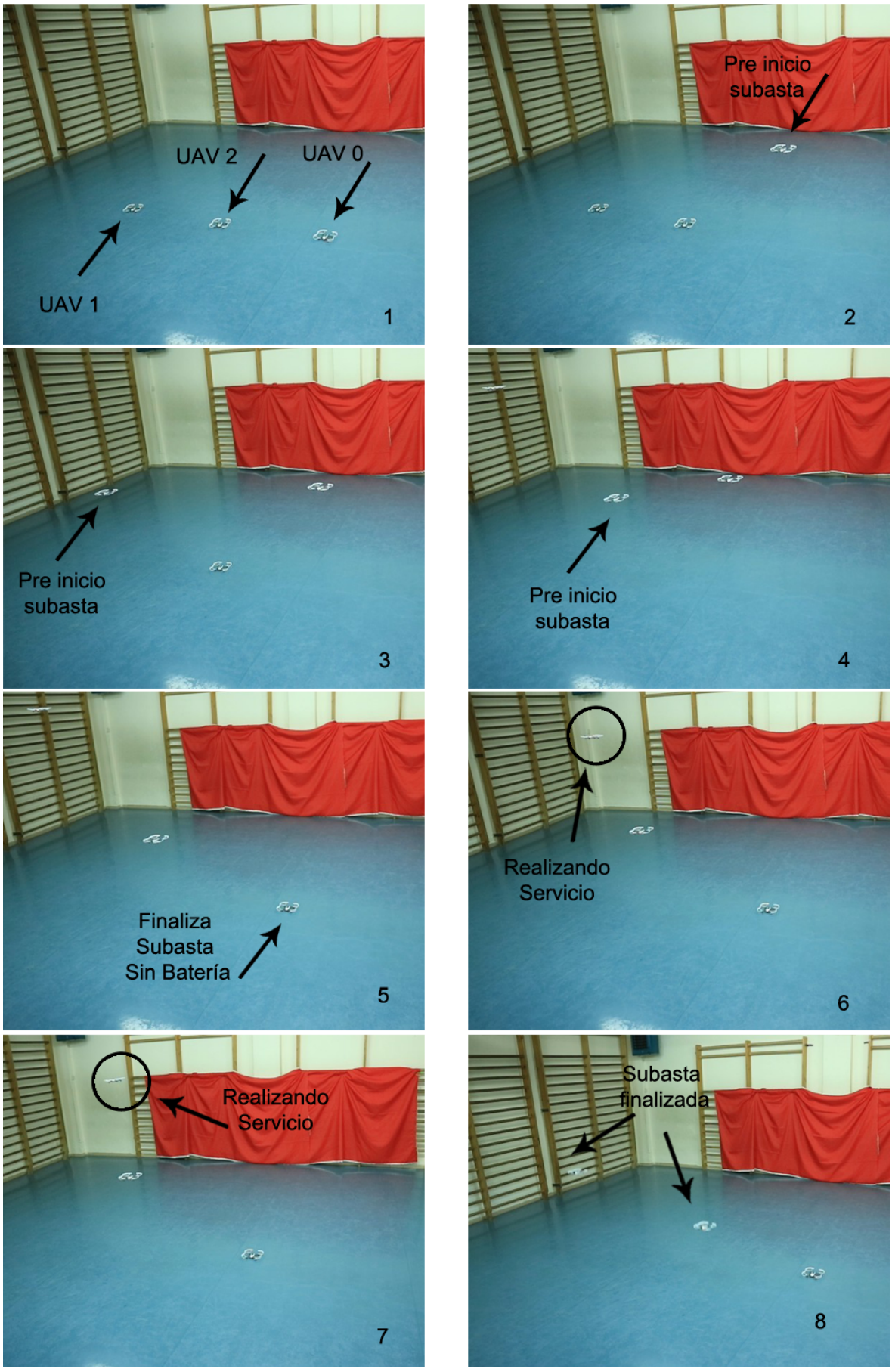

Figura 5.16: Distintas capturas de la ejecución del caso de estudio

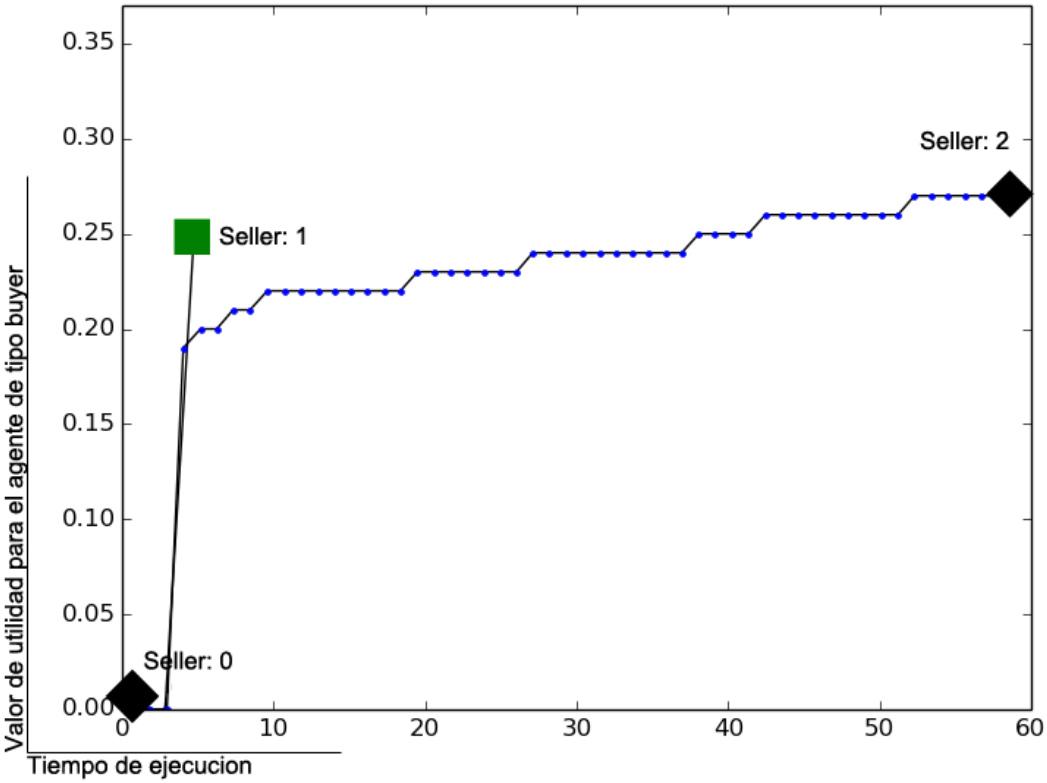

Figura 5.17: Gráfica de valor de utilidad obtenido durante el caso de estudio. 3 UAVs. Cuadro verde - Acuerdo. Rombo negro - Fin de contrato

En el informe generado por el sistema podemos observar el resultado final de la subasta.

```
$ cd SMACAUV
$ python3 demo.py 3 60 0.15 5 10 2 -p -l -ps > output.txt
....
++CLOSE++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Commitment Manager Thread==================================
Tmax: 60    Reservation value: 0.27225  Time: 63.028
Negotiations Accepted Bids:-----------------------------
0 - Negotiation Accepted ID= 1
Negotiation thread id: 1----------------------
New Bid from Seller Message:[Money:87.625 at 4.752]Utility:0.24750000000000005
------------------------------------------------------
Negotiation Status:-------------------------------------
-Negotiation ID: 0   Status: Negociaion cerrada por parte del vendedor
-Negotiation ID: 1   Status: Contract finalizado
-Negotiation ID: 2   Status: Negociaion cerrada por parte de del comprador
------------------------------------------------------
======================================================
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Gracias a este caso de estudio **podemos sacar la conclusión** de que el sistema funciona correctamente. En un futuro, no muy lejano, donde los UAVs realicen servicios y estos sean ofertados a clientes, sería posible coordinarlos siguiendo el diseño aquí propuesto. Aunque puede sonar muy pretencioso, si la evolución de los UAVs civiles continua con los niveles actuales, en un futuro serán más inteligentes y dispondrán de mayor autonomía y los medios para controlarlos remotamente o dotarlos de inteligencia serán más capaces que los medios actuales, por lo que el uso de **este sistema de coordinación podría ahorrar costes** a los dueños de los servicios dado que los UAVs podrían ajustar su precio a sus características como se ha podido observar en este caso de estudio.

5.3 Costes

Dado que este proyecto está enfocado mas a una prueba de concepto que a un producto final es complicado dar un valor total del coste del proyecto. Otro factor influyente es que el trabajo realizado no ha sido continuó durante el desarrollo del sistema debido a las siguientes causas:

- **Beca de 7 horas:** el desarrollador del proyecto ha realizado una beca con la empresa *Tecnobit* de Valdepeñas desde los meses de Febrero hasta Junio del 2017.
- **Jornada laboral:** el desarrollador empezo a trabajar en la empresa *DocPath* el 15 de Junio del año 2017.
- **Fallo de componentes HW:** han surgido problemas derivados de los dispositivos BLE que se han utilizado para el proyecto.

Estos puntos y otros más han influido en los tiempos de entrega y los costes. Antes de pasar al desglose de gastos del proyecto cabe mencionar una serie de puntos:

- **UAV:** los UAV utilizados no han costado dinero ya que se han utilizado los disponibles en la *Escuela Superior de Informática* de Ciudad real, pero para tener un precio lo más real posible se añadirán a los gastos totales de los dispositivos.
- **Sueldo desarrollador:** un desarrollador *Python* cobra 20€por hora.
- **Jornada laboral:** el desarrollador ha realizado una jornada laboral de 3 horas diarias salvo Sábados y Domingos que se ha realizado una jornada de 6 horas, lo que hace una jornada semanal, incluyendo sábados y domingos de 27 horas semanales.
- **Duración del proyecto:** aunque la duración del proyecto ha sido de 9 meses, solo 5 meses (20 semanas) han sido productivos.
- **Gastos HW:** los gastos derivados de HW han sido 469.47€ (véase Cuadro 5.6).
- **Gastos HW:** los gastos derivados del desarrollo han sido: $20 \times 27 \times 20 = 10{,}800$€

Dispositivo	Precio €	Cantidad	Total €
UAV Parrot Airborne Cargo Travis	99.90€	4	399.60€
Cargadores UGREEN 20383	9.99€	1	9.99€
Lote de cargador de bateria y bateria	22.99€	2	45.98€
Unotec Adaptador Bluetooth 4.0 USB para PC	13.9€	1	13.9€
		Total:	469,47€

Cuadro 5.6: Coste de los componentes HW

Se estima el **coste total del proyecto en 11.269,47€.**

Capítulo 6

Conclusiones y trabajo futuro

ESTE capítulo está centrado en la exposición de las conclusiones obtenidas a lo largo de todo el desarrollo del proyecto y en cómo los objetivos han sido alcanzados. También se detallarán una serie de mejoras y como éstas, en caso de realizarse en un trabajo futuro, podrían repercutir en la mejora de este proyecto.

6.1 Conclusiones

Como producto final de la realización de este proyecto se ha obtenido el esbozo de un diseño de un sistema multi-agente para la coordinación de UAV mediante negociaciones concurrentes basadas en un mecanismo de subastas flexibles. Cabe destacar que tras la ejecución de este proyecto se ha generado un conjunto de datos a partir del caso de estudio y de las pruebas realizadas y que gracias a estos datos es posible sacar unas conclusiones.

Al comienzo del proyecto se elaboró un objetivo principal y una serie de objetivos específicos (véase § 2). A continuación destacaremos las conclusiones a las que se ha llegado tras el desarrollo de este proyecto basándonos en los objetivos ya descritos.

6.1.1 Obtención de un esquema de coordinación descentralizada

El objetivo principal que este proyecto tiene es el diseño y desarrollo de un *esquema de coordinación descentralizada*. Dicho objetivo se ha cumplido y todas las metas que se pretendían obtener gracias a su implementación han sido alcanzadas:

- **El objetivo de la comunicación descentralizada ha sido alcanzada** y demostrada, es posible utilizar un sistema de **páginas amarillas** que permita a los agentes conocer su destinatario y comunicarse con él.
- **El objetivo del uso de hilos concurrentes ha sido alcanzado** y demostrado gracias a las pruebas realizadas. Es posible coordinar mediante hilos concurrentes a los UAV. Dada la arquitectura del sistema sería posible implementar ambos roles (*buyer* y *seller*) dentro de un UAV, siempre y cuando el UAV tenga capacidad multi-hilo y un ordenador de a bordo potente[1].
- **El objetivo de otorgar servicios a un coste óptimo ha sido alcanzado,** aunque con

[1] *«The Snapdragon Flight supports Linaro Linux and OpenCV.»*

ciertos matices, dado que actualmente los agentes de tipo *buyer* aceptan cualquier puja que sobrepase su valor de *reservation*. Es óptimo para el agente de tipo *seller*.

- **El objetivo de disminuir el tiempo de negociación se ha alcanzado** al realizar subastas concurrentes es posible que todos manden sus pujas de forma simultánea y el sistema pueda aceptarlas sin problemas disminuyendo así el tiempo de subasta.
- **El objetivo de realizar tareas conjuntas ha sido alcanzado** pero con matices dado que la parte de la realización de la tarea por parte de los UAV no ha sido implementada, si que puede ser aceptado, como concepto de tarea conjunta, la realización de una vigilancia conjunta como se ha visto en el *caso de estudio uno*.

Dado que todos las metas propuestas en el objetivo han sido alcanzadas, con mayor o menor acierto, se puede decir que el proyecto ha concluido de forma positiva.

6.1.2 Objetivos específicos

A parte del objetivo general, se establecieron una serie de objetivos específicos que se han querido cumplimentar con la elaboración de este proyecto. A continuacion, tal y como se ha desarrollado en el apartado anterior, se pasara a definir las conclusiones de cada objetivo específico:

- **Flexibilidad:** la flexibilidad por parte del sistema ha sido implementada. Gracias a las pruebas se ha podido comprobar que la **cancelación del contrato** es factible y que se tiene en cuenta el valor de **penalización**. **El cambio de roles puesto a prueba** pero a nivel de concepto es factible ya que , al igual que el agente de tipo *seller* es ejecutado en el PC a modo de GCS sería factible ejecutar el *buyer* simulando ser un UAV. **La ontología ha sido implementada** y ha dotado al sistema de una gran capacidad de adaptación.
- **Escalabilidad:** la escalabilidad por parte del sistema ha sido implementada gracias a una arquitectura modular y al uso de **hilos concurrentes** usados en el control de los UAV y los ayudantes del agente de tipo *buyer*. El sistema es independiente del número de agentes.
- **Descentralización:** el uso de unas **páginas amarillas** ha permitido que el sistema sea descentralizado y se hablan de agente a agente.
- **Análisis forense de los datos:** el análisis forense de los datos ha sido implementado gracias al **módulo logger** capaz de **generar imágenes de formato PNG** con diagramas utiles tanto para el cliente como para el desarrollador. Uno de esos diagramas representa el historial de utilidad alcanzado por los agentes de tipo *seller* y el otro muestra las comunicaciones entre hilos generadas durante una subasta.

6.2 Trabajo futuro

Al comienzo de este proyecto se decidió no implementar todo el documento en el que estaba basado [NJ05a] por eso uno de los trabajos futuros será terminar de implementarlo. También a lo largo del proyecto han ido apareciendo posibles mejoras para el diseño del sistema que serán destelladlas a continuación:

- **Implementación total del sistema:** como trabajo futuro principal se pretende implementar en su totalidad todos los apartados del documento utilizado como base. Por ejemplo: implementar el histórico de utilidad para así obtener unas estrategias más acordes con la realidad y aceptación de pujas por parte del agente de tipo *buyer* acorde a la estrategia del *negotiation manager*.
- **Mejorar el módulo de comunicaciones:** durante el proyecto se han ido ganando conocimientos que han llevado a pensar que el módulo de comunicaciones podría ser diseñado de otra forma más modular y fácil de utilizar. Como tarea futura se pretende reautorizar dicho módulo para definir interfaces de comunicaciones que posibiliten al sistema comunicarse a través de distintos medios.
- **Definir una ontología más completa en *Python*:** durante el proyecto se ha llegado a la conclusión de que lo más coherente seria implementar todo el sistema de comunicaciones del agente de tipo *seller* con el UAV dentro de la clase UAV definida por la *ontologia*.
- **UAV API REST:** esta posibilidad surgió al realizar la API REST del UAV para el control desde *Python*. Como trabajo futuro lo óptimo sería realizar los agentes utilizando *Node.js* y *Javascript* aunque esto limitaría su uso a un sistema controlado por un GCS o a UAV con la capacidad de ejecutar código *Javascript*.
- **Compra de UAV más inteligentes:** otra de las posibilidades que surgieron durante el desarrollo fue el uso de UAV con mayores capacidades que pudiesen ser utilizados para desarrollar y ejecutar código por sí mismos.
- **Integración con el Sistema de Vigilancia Adaptativo basado en la Coordinación de UAVs en Entornos Afectados por Catástrofes realizado por Juan Manuel Pérez Ramos [Ram16]:** se podría suplantar el sistema de coordinación realizado en el proyecto de Juan Manuel e integrarlo con el sistema de coordinación mediante subastas y utilizar las características de posición por GPS, autonomía y velocidad, para decidir que UAV acude a un punto.

ANEXOS

Anexo A

Instalación software y dependencias

ESTE proyecto hace uso de una serie de aplicaciones, lenguajes y bibliotecas que a continuación se detalla como instalar, paso por paso.

Cabe destacar, antes de empezar el manual, que ha sido testeado en Ubuntu 14.04 y que está orientado a un SO basado en *Debian*

A.1 Python3

En el terminal ejecutaremos el siguiente comando:

```
$ sudo apt-get install python3.
```

Una vez introducida la contraseña e instalado se puede acceder al terminal mediante el siguiente comando:

```
$ python3
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Este terminal permite testear código en lenguaje *Python*. Para salir del terminal se pude escribir la función **exit()**

A.1.1 Bibliotecas Python3

Para instalar las bibliotecas necesitaremos instalar antes **pip**[1] que es un gestor de descargas e instalaciones de *Python*.

```
$ sudo apt-get install python-pip
```

Una vez instalado podemos pasar a instalar PIL, *OwlReady* y *matplotlib*.

[1] https://pypi.python.org/pypi/pip

A.1.1.1. PIL

Para instalar la biblioteca de *Python* para el tratamiento de imágenes primero es necesario instalar los distintos formatos que es capaz de manejar PIL.

```
$ sudo apt-get install python-pip
```

Una vez instaladas se puede pasar al siguiente comando:

```
$ pip install PIL
```

A.1.1.2. Owlready

Para instalar Owlready solo es necesario ejecutar el siguiente comando:

```
$ pip install Owlready
```

A.1.1.3. Matplotlib

Para instalar matplotlib solo es necesario ejecutar el siguiente comando:

```
$ sudo apt-get install python-matplotlib
```

A.2 Node.js

Otra aplicación que se tiene que instalar es *Node.js*, para ello es necesario ejecutar el siguiente comando:

```
$ curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
$ sudo apt-get install -y nodejs
```

Node.js permite ejecutar aplicaciones basadas en *Javascript*. Tiene un instalador parecido a *pip* de *Python* que se instala a la vez que *Node.js*.

El instalado se llama *npm* y si el proyecto dispone de un archivo *package.json* (ver Listado A.1) es posible instalar y ejecutar scripts de forma rápida y sencilla. Como se puede ver en el Listado A.1 en el apartado **dependencies** se encuentran los módulos o bibliotecas externas que *npm* instala.

Si se ejecuta el siguiente comando *npm* instalara todas las dependencias:

```
$ npm install
```

Una vez ejecutado este comando se pueden ejecutar los comandos de la sección *scripts* del archivo *packages.json*, por ejemplo, en este caso, se podría ejecutar el comando:

```
{
  "name": "dronerest",
  "version": "1.0.0",
  "description": "Drone Api Rest",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "nodemon server.js"
  },
  "keywords": [
    "drone",
    "uav",
    "parrot",
    "manbo",
    "api",
    "rest"
  ],
  "author": "David Frutos Talavera",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^1.11.0"
  },
  "dependencies": {
    "body-parser": "^1.17.2",
    "express": "^4.15.3",
    "rolling-spider": "^1.7.0"
  }
}
```

Listado A.1: Ejemplo de un archivo *package.json*

```
$ npm start
```

que ejecutaría a su vez:

```
$ nodemon server.js
```

Estas son todas las dependencias del proyecto.

Referencias

[Ark92] R. C. Arkin. Cooperation without communication: Multiagent schema-based robot navigation. *Journal of Robotic Systems*, 9(3):351–364, 1992. url: http://dx.doi.org/10.1002/rob.4620090304.

[BA05] K. Beck y C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2005.

[Bec99] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 1999.

[BP15] A. Bengoa y E. E. País. Quiero ser piloto de drones. http://economia.elpais.com/economia/2015/08/13/actualidad/1439465207_638112.html, 2015. Economia.

[Bur16] S. Burke. Two biggest tech trends in 2016. http://money.cnn.com/2016/02/15/technology/drone-virtual-reality-tech-trends, Febrero 2016. CNNMoney.

[CS23] K. Capek y P. Selver. *R.U.R. (Rossum's Universal Robots): A fantastic melodrama*. Garden City, 1923.

[DAM01] Proyecto DAMMAD, Diseño y Aplicación de Modelos Multiagente para Ayuda a la Decisión. Technical report, Universidad de Málaga, 2001.

[Dij65a] E.W. Dijkstra. Cooperating sequential processes (EWD-123), 1965. url: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF.

[Dij65b] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 1965. url: http://doi.acm.org/10.1145/365559.365617.

[fITa] IEEE Standard for Information Technology. Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language]. IEEE Std 1003.1c.

[fITb] IEEE Standard for Information Technology. Portable Operating System Interfaces (POSIX) - Part 1: System Application Program Interface (API) - Amendment 1: Realtime Extension [C language]. IEEE Std 1003.1b.

[GS97] D. Gries y F. Schneider. *On Concurrent Programming*. Springer New York, 1997.

[Hoa74] C. A. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 1974.

[Hol91] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.

[Kni07] H. Kniberg. *Scrum y XP desde las trincheras: Cómo hacemos Scrum*. Lulu.com, 2007.

[Mye78] G. J. Myers. *Composite/structured design*. Van Nostrand Reinhold, 1978.

[NJ03] T. D. Nguyen y N. R. Jennings. Concurrent bi-lateral negotiation in agent systems. 2003.

[NJ05a] T. D. Nguyen y N. R. Jennings. Managing commitments in multiple concurrent negotiations. *Electronic Commerce Research and Applications*, 4(4):362–376, dec 2005. doi:10.1016/j.elerap.2005.06.005.

[NJ05b] T. D. Nguyen y N. R. Jennings. Managing commitments in multiple concurrent negotiations. *Electronic Commerce Research and Applications*, 4(4):362–376, 2005. url: `https://doi.org/10.1016/j.elerap.2005.06.005`.

[OdM15] M. Oñate de Mora. *Los Drones y sus aplicaciones a la ingeniería civil*, capítulo Tipología de aeronaves pilotadas por control remoto, páginas 54–55. Gráficas Arias Montano, S. A, Mostoles, Madrid, España, 2015.

[oS15] United States. Joint Chiefs of Staff. Joint Publication 1-02: Department of Defense Dictionary of Military and Associated Terms, 2015. url: `https://www.hsdl.org/?view&did=750658`.

[Par94] L. E. Parker. Heterogeneous Multi-Robot Cooperation. Master's thesis, Massachusetts Inst of Tech and Cambridge Artificial Intelligence Lab, 02/1994 1994. url: `ftp://publications.ai.mit.edu/ai-publications/1000-1499/AITR-1465/`.

[PFJ98] C. Sierra P. Faratin y N. R. Jennings. Negotiation decision functions for autonomous agents. *Robotics and Autonomous Systems*, 24(3):159–182, 1998. url: `http://doi.org/10.1016/s0921-8890(98)00029-3`.

[Ram16] J. M. Pérez Ramos. Sistema de Vigilancia Adaptativo basado en la Coordinación de UAVs en Entornos Afectados por Catástrofes. Master's thesis, Escuela Superior de Informática. Universidad de Castilla la Mancha, 09 2016.

[RN03] S. J. Russell y P. Norvig. *Artificial Intelligence: A Modern Approach*, capítulo 2. Prentice Hall, 2003.

[Sch89] A. Schiper. *Concurrent programming*. North Oxford Academic, 1989.

[Wei06] G. Weiss. *Multiagent systems*, capítulo 5. MIT Press, 2006.

[Woo01a] M. J. Wooldridge. *An introduction to multiagent systems*, capítulo 1. John Wiley & Sons, Ltd., 2001.

[Woo01b] M. J. Wooldridge. *An introduction to multiagent systems*, capítulo 7. John Wiley & Sons, Ltd., 2001.

[yES09] P. Letelier y E. Sánchez. *Métodologías ágiles para el desarrollo de software: eXtreme Programming (XP)*. Técnica Administrativa, 2009.

[ZR99] G. Zlotkin y J. S. Rosenschein. Compromise in negotiation: exploiting worth functions over states. *Artificial Intelligence*, 1999. url: https://doi.org/10.1016/0004-3702(95)00037-2.

Este documento fue editado y tipografiado con LaTeX empleando
la clase **esi-tfg** (versión 0.20170702) que se puede encontrar en:
`https://bitbucket.org/arco_group/esi-tfg`