4 Diseño del sistema

Del tiempo total invertido en la realización del presente proyecto, la mayor parte se ha dedicado a la creación de las librerías necesarias, por lo que se podría decir que si se representase el avance según una perspectiva ajena al proceso de desarrollo, se tendría una curva creciente a trozos.

Esto quiere decir, que la realización de librerías no aporta ningún resultado visible hasta que una aplicación de alto nivel las utiliza, dando la falsa impresión de que el proyecto va más lento de lo esperado y, de repente, en el último tramo, se observa un gran aumento ya que nada más realizar una versión de prueba de la interfaz gráfica, aun siendo pobre en instrumentos y grafismos, ya dispone de la capacidad para demostrar todas las funcionalidades implementadas en las librerías. de verdad

En los siguientes apartados se detalla el proceso de diseño seguido para llevar a cabo el proyecto.

4.1 Arquitectura

El proyecto se divide en varias capas bien diferenciadas, pudiendo así separar el desarrollo en capas independientes, facilitando el diseño, la implementación y la mantenibilidad.

Figura 23: División por capas del proyecto.

La primera capa obviamente la conforman los distintos dispositivos hardware susceptibles a ser integrados en el proyecto. En principio no hay impedimentos para incluir cualquier tipo de maquinaria o instrumentos, ya que todos ellos ofrecen datos o capacidad de acción, bien mediante la API que proporcione el fabricante, o bien directamente ya sea de forma visual, por red, etc.

En definitiva, datos o comandos a los que se podrán acceder gracias a la capa siguiente: la *HAL Service* o simplemente HAL (*Hardware Abstraction Layer*), que no es más que una envoltura o *wrapper* de los drivers o controladores del dispositivo físico proporcionados por el fabricante, o la abstracción de la comunicación directa con el dispositivo. La capa HAL será la encargada entonces de servir datos a la capa superior, y/o comandar acciones recibidas desde dicha capa al dispositivo que lo permita.

El siguiente escalón en la arquitectura, es la capa que permite la comunicación por red, utilizando para ello una tecnología que pertenece al estado del arte actual, llamada DDS (*Data Distribution Service*), la cual se explica más adelante. Esta capa de comunicación (denominada internamente en el proyecto como CL) se compone de un envoltorio a la inmensa API de DDS, para mostrar una interfaz sencilla, acorde a las necesidades del proyecto.

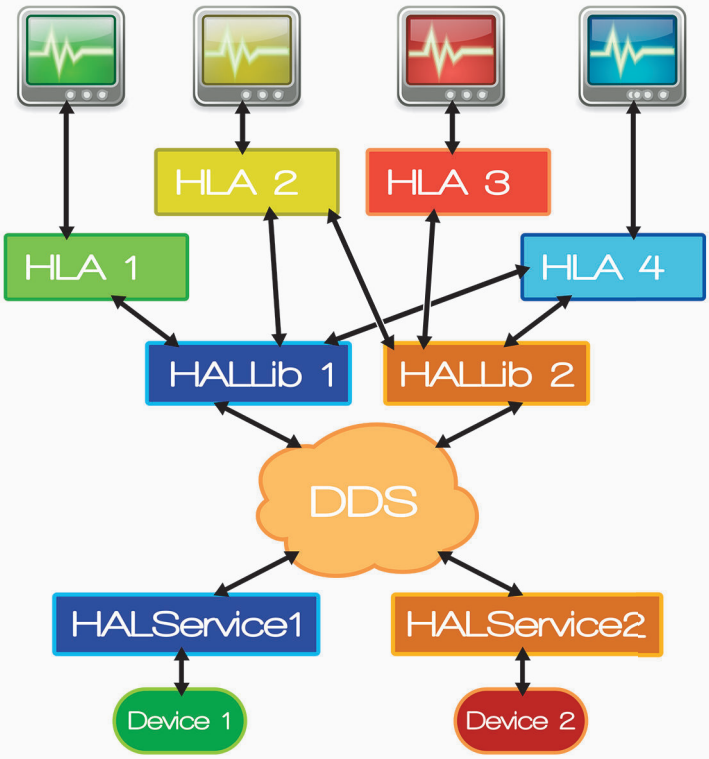

Figura 24: Arquitectura del proyecto.

La capa superior a CL, se denomina *Hal Library*, o HALlib. Esta capa permite a las aplicaciones de alto nivel (HLA) más fácilmente a la funcionalidad de CL, ya que son personalizadas en cada dispositivo para automatizar todo lo posible la comunicación y evitar así mala *praxis* en el acceso a la información.

Dicho esto, denotar la versatilidad de la arquitectura, ya que distintas aplicaciones de alto nivel pueden acceder a los mismos dispositivos simultáneamente en tiempo real. Esto es así dinámicamente, es decir, puede haber un dispositivo al que no acceda ninguna HLA, y a lo largo del tiempo haber una, o varias, y todas ellas recibirán los datos del dispositivo en tiempo real; por lo que el dispositivo es una fuente de datos continua, y las aplicaciones irán absorbiendo esos datos según los necesiten.

4.2 Capa 1: capa de comunicación

Esta sección del proyecto engloba todos los procedimientos necesarios para establecer la comunicación entre los diferentes nodos de la red [23]. De cara al equipo terminal, presenta una API con los métodos suficientes para enviar o recibir los datos que interesen sin tener en cuenta si existe o no una red, quien va a recibir los datos que envía, quien envía los datos que recibe, o cuántos equipos hay en la red.

4.2.1 Diseño

La capa CL se encuentra dividida en dos módulos: uno llamado *ddsLibs* y otro denominado *idlLibs*. El primero se encarga de implementar el ya mencionado *wrapper* o envoltura de los métodos de DDS, tales como la creación de participantes, publicadores, registro de tópicos, etc.

Por otro lado, el módulo *idlLibs* está creado para servir como punto de entrada a la librería. Este módulo instancia todos los *templates* de *ddsLibs*, por lo que el usuario final de la librería CL se encuentra totalmente abstraído de utilizar ni instanciar *template* alguno, haciendo muy fácil el uso del protocolo DDS.

El nombre del módulo hace referencia al tipo de dato utilizado para la comunicación como se verá en el apartado *4.2.4*. Si el usuario requiere la utilización por parte de DDS de un nuevo tipo de dato, únicamente tiene que crear el tipo de dato *idl* deseado en el directorio destinado a albergar todos los *idl*, y seguidamente volver a compilar el proyecto. Esta acción genera automáticamente todos los *templates* necesarios por lo que no es necesaria la modificación del código existente en la librería.

Como punto de entrada a la librería, el módulo esta formado por unos componentes base que son los únicos que debe tener en cuenta el usuario:

- **DdsParticipant**: Es el punto de entrada de *idlLibs*. Cada instancia de esta clase representa a un participante de DDS y, como tal, es la que ofrece la funcionalidad de crear instancias del tipo de dato deseado, la calidad de servicio o QoS tanto a nivel global como específico para los publicadores, subscriptores, etc.

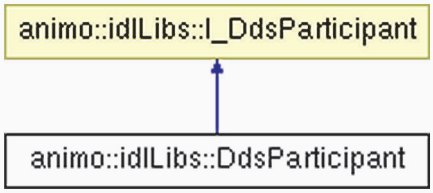

Figura 25: Jerarquía del participante.

▷ **PublishService**: Esta clase crea el servicio de publicación de los diferentes tipos de datos, dónde especificando el tópico que representará a los datos la librería enviará por red la instancia de dato pasada como parámetro.

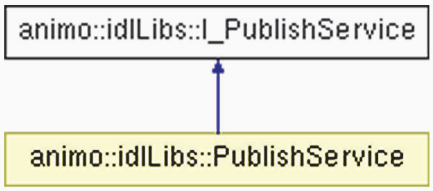

Figura 26: Jerarquía del publicador.

▷ **SubscribeService**: Análogamente, esta clase es la que se encarga de suscribir a la aplicación a los diferentes tópicos. Mediante el simple paso de parámetros al método de subscripción, se consigue conectar DDS con el método de recepción de datos de la aplicación cliente, consiguiendo que cuando un dato sea enviado, DDS lo replicará en el método de recepción.

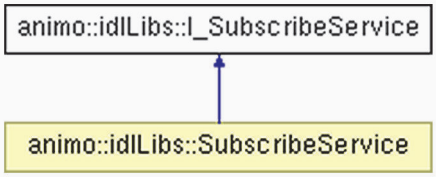

Figura 27: Jerarquía del subscriptor.

▷ **Idl*Type*Listener**: Donde *Type* indica el tipo de dato que se esté manejando, pudiendo ser por ejemplo posición, velocidad, aceleración, velocidad angular, nivel de batería, errores, etc.

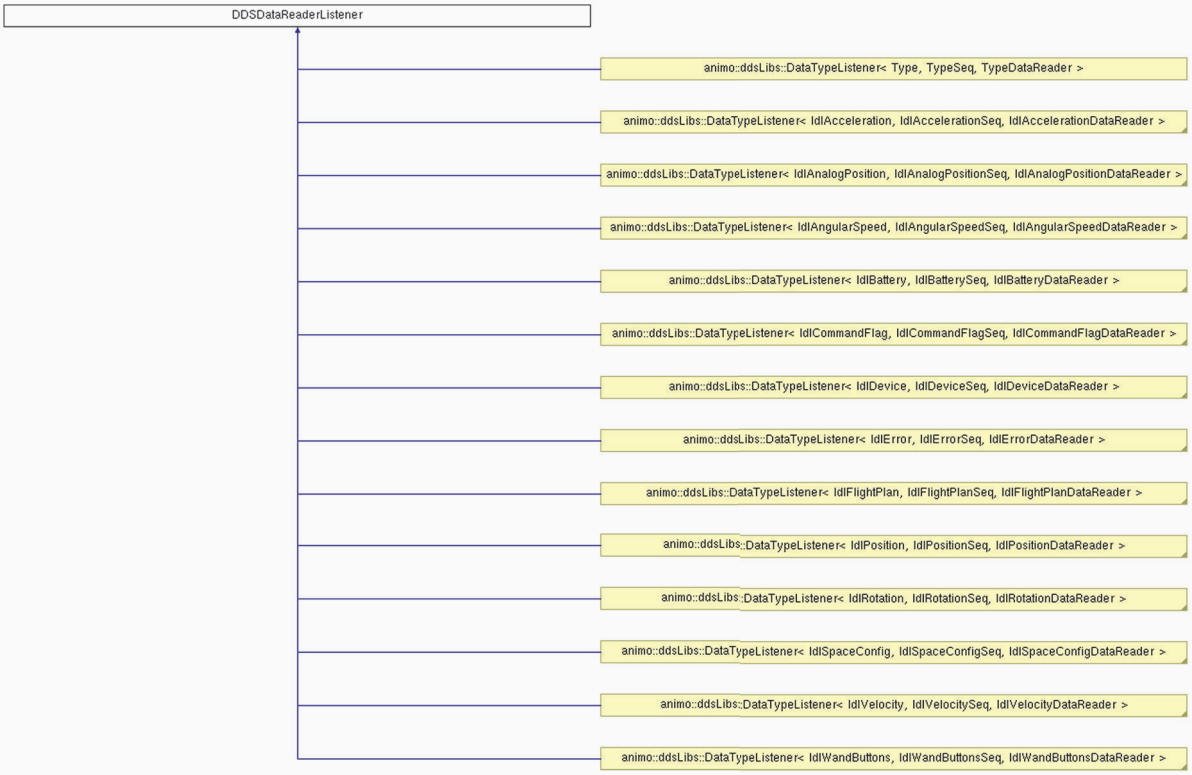

Figura 28: Jerarquía de los *listeners*

4.2.2 Aplicaciones

El modelo de comunicación que presenta DDS, abre un abanico de posibilidades muy interesante, ya que ahora los programadores no se tienen que preocupar de la transmisión y recepción de datos lo que conlleva a un gran ahorro de tiempo y código. Por un lado, el desarrollador se centra en la funcionalidad de la aplicación, y por otro, el administrador de la red tiene libertad para realizar la topología que se necesite, y modificarla si es preciso sin que ello conlleve cambios en la aplicación.

Las aplicaciones más aclamadas últimamente llegan de los centros de investigación, ya que posibilita el envío de datos de múltiples fuentes distribuidas geográficamente en tiempo real, recibiéndolos todos los equipos que se subscriban sin necesidad de estar presentes en el mismo lugar y añadir nuevos equipos o no dinámicamente según las necesidades ya que los subscriptores y publicadores se ajustan automáticamente gracias

al descubrimiento dinámico de DDS. Los equipos que procesen dicha información, no tendrán que interpretar todos los datos que se obtengan del experimento, sino centrarse únicamente en los de interés, lo que conlleva más potencia de cálculo en cada equipo al poder analizar los datos entre todos los participantes también en tiempo real.

Por otro lado, se minimizan las barreras de entrada al proyecto ya que se da lugar a utilizar como medio de comunicación redes heterogéneas ya existentes como Internet, incluso redes en desventaja o limitadas como la red de telefonía móvil. Esto da lugar por ejemplo a sistemas de control en tiempo real (por ejemplo Scadas) con una alta capacidad de movilidad geográfica.

Todo esto con una alta capacidad de gestión y control, ya que es parte del estándar incorporar mecanismos de seguridad, como cifrado, autenticación y gestión de dominios.

4.2.3 Abstracción de DDS

Se habla de abstracción cuándo es necesario ocultar al usuario el funcionamiento del código. En el caso de DDS, al ser una API extensa, dispone de numerosas opciones y métodos diversos que en su mayoría no son relevantes para la aplicación objeto de este proyecto, por lo que resulta necesario cubrir dicha librería para mostrar al usuario una interfaz sencilla, que configure automáticamente las opciones de DDS para que únicamente haya que crear un objeto de dicha interfaz que presente unos métodos sencillos para publicar y subscribirse a los tópicos necesarios.

La Capa de Comunicación (CL en adelante), es la abstracción de la librería de DDS ofertada por RTI. Presenta al programador una interfaz en dónde podemos encontrar métodos para publicar/subscribirse a tópicos previamente definidos en los tipos de datos que se van a manejar, ya que una de las maneras en que CL automatiza el proceso de configuración, es basarse en el tipo de datos a enviar para obtener el tópico y, de esta forma, tanto por seguridad como facilidad, se abstrae al desarrollador de esta tarea. Esto es así, ya que definiendo el tipo de datos una única vez, cualquier otro proyecto que quiera utilizarlos sólo tiene que incluir la librería creada (que ya contiene los tipos de datos) y enviar o recibir datos, evitando así la posibilidad de un error humano en cuanto al nombre del tópico se refiere dado que deben coincidir exactamente para poder realizar la comunicación.

La siguiente tarea a automatizar es la creación de las entidades que llevan a cabo la

comunicación. En lugar de tener que crear un participante, de añadir un publicador y/o subscriptor, de registrar un tópico, de activar un *data writer/data subscriber* (el último eslabón antes de publicar/subscribirse) y tener que preocuparse de la liberación de memoria de todos ellos, CL se encarga de que crear las entidades necesarias para poder realizar la comunicación, y de eliminarlas limpiamente una vez termine la aplicación.

4.2.4 Tipos de datos

A través de DDS se puede enviar cualquier tipo de dato, si bien, previamente hay que definir en código la estructura que lo va a albergar, para que la aplicación pueda rellenarla con información y posteriormente publicarlo o recibirlo y procesar los datos. Esto se hace para que el envío de datos sea común para todos los equipos que participen, ya que no tienen por qué tener el mismo sistema operativo, ni la misma arquitectura (pueden ser *Little Endian*, o *Big Endian*), por lo que se asegura la comunicación entre sistemas heterogéneos.

La API de RTI reconoce los siguientes tipos de estructuras para representar datos: *OMG Interface Definition Language* (IDL), XSD/WSDL, XML o las llamadas UPI (*User Programmatic Interface*), que son un conjunto de APIs que dan soporte para el acceso, gestión y manipulación de bases de datos.

De entre las distintas opciones, se escogió IDL por su simplicidad y semejanza con C++, haciendo más sencilla la comprensión del código a programadores ajenos al proyecto. La forma de definir un tipo de dato IDL es exactamente de la misma manera en que se define una estructura en lenguaje C/C++. Una vez definida hay que transformarlos a ficheros de código fuente, función que la realiza automáticamente la utilidad de RTI llamada *rtiddsgen*.

Como se dijo al principio de este documento, una de las tareas es capturar datos de vuelo de cuadricópteros enviados desde el equipo que controla el *testbed*. Por lo que se define un IDL por cada una de las siguientes medidas:

- ▷ Posición. Se definen las posiciones exactas de cada quadrotor referenciadas al origen de coordenadas del volumen por dónde se desplazan. La estructura dispone de un campo por cada coordenada del sistema (X,Y y Z).

- ▷ Velocidad. Expresada en $\frac{m}{s}$ en cada uno de los ejes anteriores.

- ▷ Aceleración. Expresada en $\frac{m}{s^2}$ en los mismos ejes.
- ▷ Actitud. Información del *roll*, *pitch* y *yaw*.
- ▷ Nivel de batería. Cada quadrotor dispone de una batería de litio, lo que las hace delicadas en mantenimiento, por lo que resulta necesario monitorizar la descarga de la misma.
- ▷ Configuración. El sistema puede simular vuelos de aeronaves comerciales, es por ello que se pueden configurar las escalas para que la interfaz muestre el espacio aéreo con las mismas dimensiones y coordenadas que el vuelo real.

4.2.5 Calidades de servicio (QoS)

En cualquier comunicación de red que se precie, deben de existir mecanismos que aseguren un control sobre lo que se esta transmitiendo y cómo se está haciendo. En el caso de las QoS, brindan al usuario la capacidad de controlar lo bien o mal (fiable o no) que se transmite la información. Dependiendo del tipo de información y del tipo de red, se tendrá que llegar a un compromiso para que la comunicación sea factible, y que la red la soporte.

En el caso de DDS, la QoS viene dado por unos ficheros en formato XML, en dónde se personalizarán las entidades que intervienen en la comunicación. Esto quiere decir que desde el participante hasta el tópico pueden estar bajo políticas de calidad de servicio.

Mediante DDS, se puede transferir cualquier tipo de información, algunos serán muy pesadas para la red, y otros no tanto. DDS es capaz de realizar transferencias de información fiables, y para ello utiliza mecanismos tales como *buffers* de salida y llegada para tener un historial de paquetes y comprobar que todos lleguen y que lo hagan en orden.

Por ejemplo se puede enviar con esta tecnología vídeo en *streaming*, codificando los fotogramas como *char**, pero resultando paquetes grandes y a menos que la red sea fiable, se perderán datos. Para una situación así, lo que importa es el ancho de banda disponible y, en caso del streaming, que sea en tiempo real. Por tanto, habrá que liberar a DDS de carga extra como la comprobación de datos transferidos, ya que la pérdida de un fotograma no debe retrasar el envío y recepción de los siguientes. Con esa idea,

se ofrecen al usuario tres perfiles básicos de QoS que se pueden utilizar tal cual, o modificar según las necesidades concretas de la aplicación:

- *Reliable*.
  Es el perfil por defecto, y el más utilizado. Transfiere información de manera fiable, utilizando un historial para albergar los paquetes y comprobar el correcto orden de llegada. El hecho de utilizar el historial, hace que el equipo necesite más recursos como memoria RAM, además de cargar bastante más el procesador que otros perfiles. El perfil tiene la capacidad de dividir los paquetes largos de forma transparente al usuario para optimizar el envío de información.

- *High throughput*.
  Destinado a redes con gran ancho de banda y computadoras potentes y con recursos suficientes. Capacidad de transmisión de gran caudal de información, pensado para contenidos multimedia principalmente, por lo que tampoco se implementan historiales grandes, ni se comprueba la correcta recepción de los datos. No fragmenta los paquetes ya que se considera el uso de redes fiables.

- *Lossy Network*. Pensada para redes heterogéneas limitadas de alguna manera o con pérdidas importantes tales como la red de telefonía móvil. Se fragmentan los paquetes para optimizar la comunicación y no se guarda historial ninguno así como tampoco se comprueba la pérdida de paquetes. El perfil está pensado para bajos volúmenes de datos pero con la necesidad de tiempo real, como por ejemplo aplicaciones de telemetría, dónde la información a enviar son valores numéricos principalmente.

4.3 Capa 2: Servidor de dispositivo

Hasta ahora se dispone de una librería para poder comunicarse de forma sencilla con el resto de elementos de una red local. El siguiente paso lógico es generar la información del dispositivo, lo que supone crear una aplicación que obtenga los datos y posteriormente utilice la librería CL para enviarlos.

4.3.1 Testbed service

Como ya se ha mencionado anteriormente, el testbed es un banco de pruebas para la simulación de vuelos de aeronaves de todo tipo mediante el uso de cuadricópteros. El sistema está formado por una batería de cámaras infrarrojas que rodean la jaula por donde se desplazan los cuadricópteros, pudiendo obtener en todo momento la posición exacta de estos. La manera en que las cámaras detectan el movimiento se basa en fijar unos elementos reflectantes en dicha banda electromagnética al cuerpo de los cuadricópteros.

El sistema de cámaras se conecta a un equipo QNX que permite controlarlas y recibir los datos en tiempo real. La máquina QNX proporciona mediante el protocolo UDP la información recabada, es decir, se le programa la dirección del *host* destino que va a procesar los datos y los envía constantemente sin esperar asentimiento alguno. Se tiene por tanto que el servidor de datos del testbed podrá ubicarse en la misma máquina o en cualquier otra con la que tenga visibilidad de red.

Esto último también puede conseguirse con DDS. Entonces, ¿ por qué se utiliza UDP para enviar datos a una aplicación que lo reenviará por DDS?. La respuesta es simple, en cuanto que en QNX no es viable la implementación de DDS debido a las limitaciones del sistema operativo en lo que se refiere a compatibilidad de librerías y compilación necesarias para desarrollar el servidor de dispositivo nativamente, por lo que queda relegado a funcionar en una máquina externa.

4.3.2 Diseño

A esta entidad software se la denomina *server* o *wrapper* ya que tiene ambas funcionalidades, siendo el *server* la parte que ofrece los datos del dispositivo a la red, y el *wrapper* la capa que envuelve los drivers del fabricante para poder unificar en la medida de lo posible el acceso al hardware para todos los dispositivos. En otras palabras, se crea con el objetivo de mostrar al programador una interfaz lo más parecida posible para todos los sistemas, ofreciendo los mismos métodos y funcionalidades.

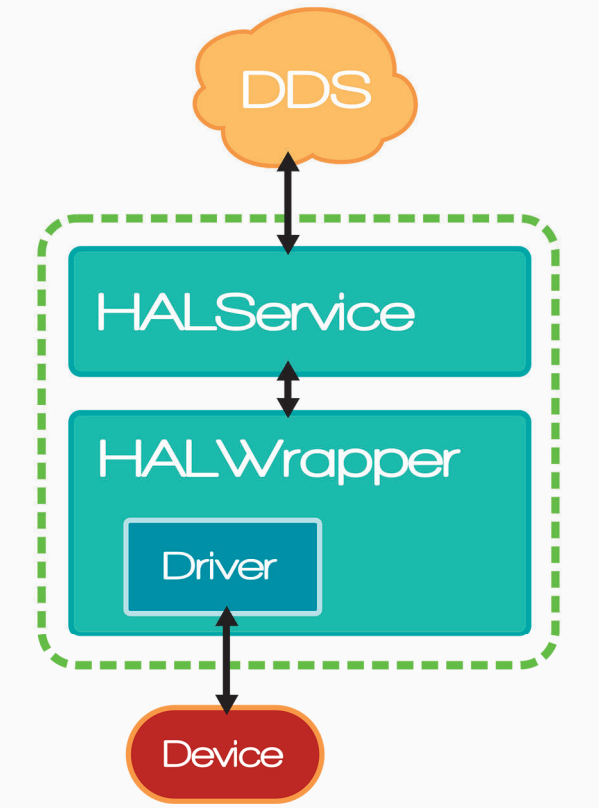

Figura 29: Servidor de dispositivo.

La parte que hace de *server* puede ir separada o no del *wrapper* (ejecutarse en distintos equipos se refiere), según se considere mejor una implementación u otra. De esta manera, se puede delegar carga computacional a un equipo más potente que el disponible para controlar el dispositivo.

El diseño del *wrapper* es claramente más complejo que la capa anterior como se verá a continuación. Ahora se requiere algo más que un envoltorio para una librería ya existente; lo que se necesita es un ejecutable que actúe como demonio o servicio. Esto quiere decir que el ejecutable no necesita interaccionar con ningún hipotético usuario, únicamente vigilará continuamente el dispositivo para envíar los datos que genere.

La aplicación hace uso de funciones de red para comunicación por UDP como se ha mencionado anteriormente, por lo que se decidió utilizar las bondades que la librería Qt ofrece para C++. Entre ellas destaca la posibilidad de disponer de un tipo de dato "datagrama" con métodos asociados que facilitan la lectura y modificación del mismo,

envío y recepción por UDP (sin necesidad de abrir puertos a mano, ni crear *sockets*, etc.), además de la creación de "señales" y "*slots*" que permiten conectar eventos con el método deseado para manejarlos, lo que permite olvidarse de esperas activas mejorando el rendimiento y velocidad de la aplicación.

El diseño por tanto tendrá clases para utilizar la librería CL ya existente y clases para manejar funciones de red; para agrupar las diferentes funcionalidades se han creado *namespaces*:

- **Network**: Alberga todas las entidades relacionadas con el envío y la recepción de datos por UDP. El testbed genera información sobre los dispositivos móviles y recibe órdenes que puede hacer llegar hasta dichos dispositivos para que se posicionen en un punto específico del espacio disponible, es por ello que en este *namespace* se encuentran métodos para reconocer que datagramas son comandos, cuales son objetos de datos, y cómo se actúa en consecuencia. Es decir, al recibir un objeto de datos, inmediatamente se parsea a tipo de dato *idl* y se envía por DDS; análogamente, al recibir un comando, se "ejecuta", que no es más que enviarlo por UDP al testbed y éste reacciona en consecuencia.

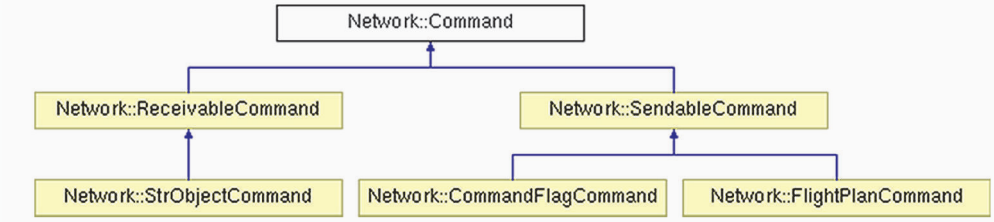

Figura 30: Jerarquía de clases dentro del namespace *Network*.

- **interfaceCL**: Es el *namespace* dónde se encuentran los métodos capaces de comunicarse a través de la capa CL. Creación de los *listeners*, publicadores y subscriptores.

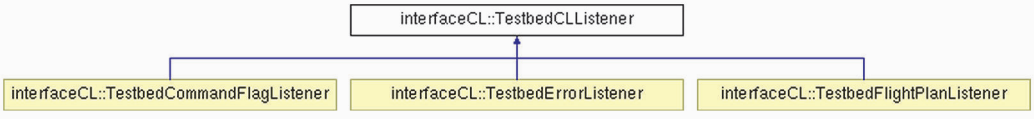

Figura 31: Jerarquía del *listener*.

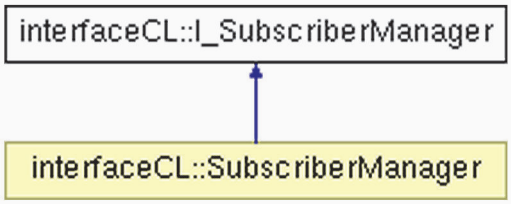

Figura 32: Jerarquía del manejador de subscripción.

- **Toolkit**: En todo proyecto siempre existen clases y métodos que no están relacionados directamente con la funcionalidad final, como por ejemplo funciones matemáticas requeridas, comprobación de cadenas de caracteres, etc. Todos estos métodos que son meras utilidades para el correcto funcionamiento que la aplicación se suelen agrupar en un *namespace* como éste. De esta forma, se intenta crear un "cajón desastre" para que esas pequeñas utilidades estén localizadas y no esparcidas por todo el código, lo que favorece su modificación y clarifica el código del proyecto. Las clases que lo componen son:
  - **EndianConverter**: Puesto que la comunicación por DDS no requiere homogeneidad de sistemas operativos ni arquitecturas, se debe tener en cuenta que cada dato puede venir en *BigEndian* o *LittleEndian*, por lo que esta clase está incluida para la correcta transferencia de datos entre equipos.
  - **QByteArrayTransform**: Como se dijo anteriormente, se hace uso de las bondades de Qt, por lo que para trabajar más cómodamente, se transforman los tipos de dato básicos a tipo *QByteArray*.
  - **Settings**: Es una clase orientada a configurar la aplicación nada más ejecutarse, ya que es la encargada de generar el fichero de log, y de leer el archivo de configuración de tipo *.ini* en dónde el usuario puede especificar algunos parámetros como una ruta específica para el archivo de log, el nombre de dicho archivo, etc.
  - **Singleton**: Clase abstracta para la creación de instancias únicas utilizando *templates*.
- **TestbedServerApp**: Es el *main* de la aplicación. Aquí se ejecutan los métodos iniciales para configurar el funcionamiento del ejecutable, como la inicialización del

archivo de log, lectura de los parámetros configurables por el usuario mediante un archivo .ini, y la configuración de las señales de Qt anteriormente mencionadas.

4.3.3 Datagramas

Para la correcta transmisión de los datos por UDP entre dos máquinas genéricas (pudiendo diferir en arquitectura y/o sistema operativo), se debe definir cómo van a enviarse los datos por la red. Se entra por tanto en la necesidad de diseñar el datagrama que se encapsulará en UDP, para fijar los campos de información útil y los de cabecera necesarios.

El equipo QNX enviará los mensajes hasta el *host* especificado sin esperar confirmación alguna, a una tasa de 100Hz. La elección de la tasa no es casual, ya que se ajusta a la tasa de obtención de datos por parte del testbed, siendo más que apta para la monitorización en tiempo real.

El datagrama a enviar, cubre las necesidades de precisión de los campos de información, dejando como cabecera un único campo identificador para discernir entre tipos de datos.

OBJECT

0	OBJ_ID
4	TIMESTAMP
12	POS_STATUS
16	VEL_STATUS
20	ACT_STATUS
24	BAT_STATUS
28	POS_X
36	POS_Y
44	POS_Z
52	VEL_X

DATA TYPES

STRUCT_ID	unsigned int	[1,50]	-
NUM_OF_STRUCT	unsigned int	[1,50]	-
OBJ_ID	unsigned int	[1,50]	-
TIMESTAMP	unsigned long	[-,-]	-
*_STATUS	int	[0,1]	-
POS_X, POS_Y	double	[0,16]	m
POS_Z	double	[-0.5,5]	m

60	VEL_Y	VEL_*	double	[-15,15]	m/s
68	VEL_Z	ACT_ROLL	double	(-pi,pi]	rad
76	ACT_ROLL	ACT_PITCH	double	[-pi/2,pi/2]	rad
84	ACT_PITCH	ACT_YAW	double	(-pi,pi]	rad
92	ACT_YAW	BAT_VOLTAGE	double	[0,50]	V
100	BAT_VOLTAGE	BAT_CURRENT	double	[-20,20]	A
108	BAT_CURRENT				

4.3.4 Parseo de datos

Una vez recibidos los datos por UDP, urge la necesidad de transmitirlos al fin por DDS. El envío con esta tecnología requiere ajustarse al abanico de tipos de dato definidos, por lo que se requiere de un adaptador que traduzca la información recibida del dispositivo a estructura de datos objetivo. Es en éste momento dónde entra en juego el código fuente generado automáticamente a partir de los ficheros IDL, ya que se crearán instancias de cada tipo según sea necesario para rellenarlas con los datos a enviar.

4.4 Capa 3: Librería de dispositivo

En la sección anterior se ha visto cómo transmitir por la red la información generada por un dispositivo. Para completar la comunicación, sólo resta agregar una entidad software que sea capaz de recibir e interpretar esos datos. Es lo que se denomina *librería de dispositivo*, la cual provee a la aplicación de alto nivel de interfaces para el acceso a los datos.

4.4.1 Diseño

Para la correcta actualización de los datos en la aplicación de alto nivel, se ha utilizado en el diseño de la librería de dispositivo (o *HALLib*) el patrón *Observer*, también conocido como *Spider*. Es un patrón de diseño que define una dependencia del tipo *uno-a-muchos* entre objetos, de manera que cuando uno de los objetos cambia su estado, el observador se encarga de notificar este cambio a todos los otros dependientes. El objetivo de este patrón es desacoplar la clase de los objetos clientes del objeto proveedor, aumentando la modularidad del lenguaje, así como evitar bucles de actualización (espera activa o polling).

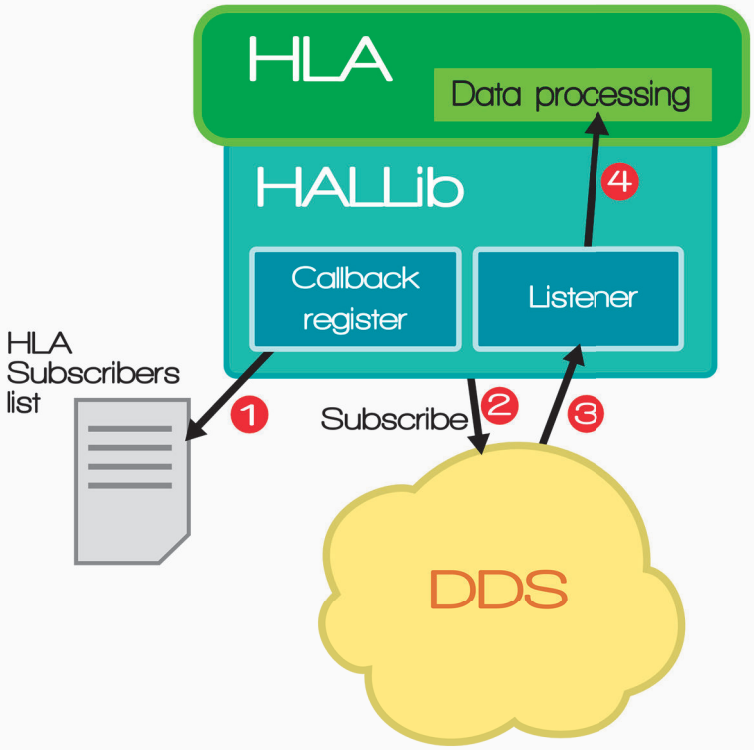

Figura 33: Diseño e implementación de la librería.

Este patrón también se conoce como el patrón de *publicación-inscripción* o *modelo-patrón*. Estos nombres sugieren las ideas básicas del patrón, que son bien sencillas: el objeto de datos, llamado ”Sujeto” a partir de ahora, contiene atributos mediante los

cuales cualquier objeto observador se puede suscribir a él pasándole una referencia a sí mismo. El Sujeto mantiene así una lista de las referencias a sus observadores.

Los observadores a su vez están obligados a implementar unos métodos determinados mediante los cuales el Sujeto es capaz de notificar a sus observadores suscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado. De manera que cuando se produce un cambio en el Sujeto, ejecutado por ejemplo por alguno de los observadores, el objeto de datos puede recorrer la lista de observadores avisando a cada uno.

Este patrón suele observarse en los frameworks de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir 'listeners' a los objetos que pueden disparar eventos.

Siendo esto así, se escogió dividir la librerí en el siguiente conjunto de clases:

- **RegisterCallback**: Es la clase que maneja la mencionada lista de referencias a los observadores. Dispone de métodos que añaden y eliminan observadores de la lista, además de realizar las comprobaciones oportunas para no duplicar elementos, ni tampoco eliminar un registro por error. Estas comprobaciones no son en vano, ya que, una misma aplicación de alto nivel puede subscribirse varias veces (con un número de identificación diferente por supuesto), y para construir una lista eficiente se ha optado por la utilización de mapas en C++ los cuales permiten borrar todas las entradas del mismo nombre que existan. Estos mapas no son más que un objeto parecido a una lista indexada que ofrece numerosos métodos para manejarlas y acceder a los datos de manera óptima en lo que a tiempo de ejecución se refiere.

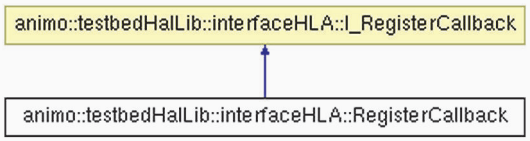

Figura 34: Jerarquía de RegisterCallback.

- **TestbedCLListener**: De esta intefaz heredan todos los listeners. Dispone de un atributo de la clase *RegisterCallback* que se le pasa mediante el constructor a

todos los objetos hijo, con el objetivo de que todos tengan la misma lista accesible. Los objetos hijo por otro lado, tienen un método virtual para la recepción de los datos, el cual debe de ser implementado en la aplicación final. De esta manera, se está realizando un *callback* o llamada inversa, ya que el objeto es el llamado cuando se recibe un dato por DDS en vez de ejecutar una función o método.

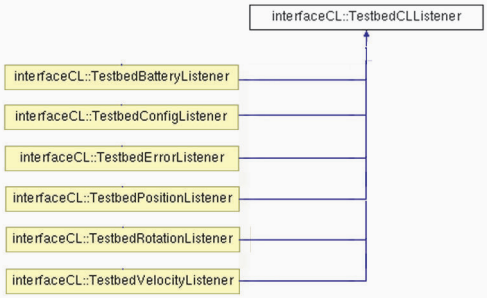

Figura 35: Jerarquía del *listener*.

- **TestbedAccessPoint**: Como su nombre indica, es el punto de acceso a la funcionalidad de la libreria, siendo la clase a la cual llamará el desarrollador para poder registrar la aplicación en la lista de observadores. Además, ofrece al usuario métodos para dar la orden de empezar a recibir datos y dejar de recibirlos.
- **DDSComunicationContainer**: Se encarga de automatizar el acceso a la librería CL, pero se deja al usuario crear el objeto de esta clase ya que permite la selección de las diferentes calidades de servicio y crear más instancias adicionales para otras comunicaciones diferentes pudiendo elegir diferentes QoS.

Figura 36: Jerarquía de DDSComunicationContainer.

- **DDSSender**: Permite al usuario enviar a la red los comandos para controlar los dispositivos del testbed.

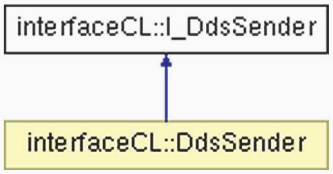

Figura 37: Jerarquía de DDSSender.

▷ **DDSReceiver**: Ofrece la capacidad de elegir que dispositivo o dispositivos del testbed se quiere subscribir el usuario.

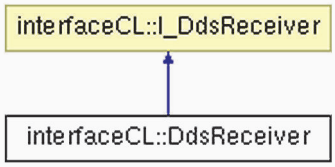

Figura 38: Jerarquía de DDSReceiver.

4.4.2 Testbed library

Se concibe como un elemento a incorporar en cada una de las aplicaciones de alto nivel que necesiten comunicarse mediante la red de datos de DDS. Es una librería dinámica, permitiendo así el acceso múltiple a los datos recabados ya que se piensa en un paradigma punto-multipunto (un servidor de datos, y numerosos equipos y aplicaciones a la espera de información). El método a seguir para utilizar su funcionalidad es muy simple: la librería dispone de unos métodos virtuales los cuáles se llaman al recibir un dato. Es por ello que la aplicación de alto nivel debe implementar todos los métodos virtuales (uno por cada tipo de dato posible).

4.4.3 Subscripción a datos

Antes de poder manipular información proveniente del dispositivo, es requisito indispensable dejar constancia de que se quiere recibir los datos identificados mediante un determinado tópico. Esto queda registrado en DDS mediante la capa de comunicación CL. En la librería del testbed se encuentran los métodos que permiten pasar los

identificadores de cuadricópteros a la capa CL, siendo ésta última la que selecciona el tópico adecuado correspondiente al número de quadrotor y subscribiéndose a sus datos.

4.4.4 Recepción de datos

Al realizar una subscripción, se crea una entidad DDS llamada *listener*, que no es más que un manipulador que "observa" la información transmitida por la red de datos. En cuanto llegue un dato a los cuales se está subscrito, se le transmite a la librería. Seguidamente, y a través de *callbacks*, la aplicación de alto nivel recibe los datos para procesarlos.

4.5 HMI (aplicación gráfica de alto nivel)

Las entidades software anteriores no tienen sentido sin una aplicación de alto nivel que las utilice, es por ello que éste proyecto culmina con el desarrollo de una interfaz gráfica de usuario (en adelante HMI).

Su objetivo es implementar de manera muy práctica lo que en aviación se considera una Estación de Tierra. Práctica porque se centra en mostrar al piloto u operario los datos más relevantes de los sistemas en vuelo rápidamente mediante indicadores, pero también cómoda ya que toda la información necesaria está siempre a la vista sin sobrecargar la pantalla y sin distraer al usuario. Además está pensada para poder acceder a sus distintas funcionalidades en los mínimos clicks posibles, lo que agiliza su utilización y aprendizaje.

El HMI es altamente flexible, es decir, se le pueden programar nuevos módulos según se requiera para ir ampliando sus funcionalidades. Un módulo es cualquier entidad que aporte funcionalidad a la interfaz, mientras que la interfaz por sí sola es únicamente un contenedor. Cada módulo es independiente tanto de la interfaz gráfica como de los otros módulos, por lo que se pueden desarrollar en paralelo por diferentes equipos de desarrolladores mientras se cumpla la misma estructura de proyecto.

4.5.1 Diseño

La arquitectura empleada para desarrollar el HMI es bastante sencilla de entender ya que también se separa en los módulos necesarios para poder realizar un mantenimiento, modificación y/o ampliación de las funcionalidades de forma eficiente.

Se tienen únicamente dos *namespaces* distintos: "HMI" y "disti". El primero se crea para contener toda la funcionalidad relacionada con la comunicación de red, desde la implementación de la librería del dispositivo o HALLib, hasta las clases que adaptan los valores recibidos para poder mostrarlos por pantalla, como por ejemplo el filtro de datos para eliminar ruido. El segundo es creado por el software GlStudio, y engloba toda la parte visual del proyecto.

Para ello lo primero es ver cómo aislar el código generado por GlStudio del resto del proyecto, ya que dicho código no debe ser modificado fuera de GlStudio y la intención es añadir o modificar instrumentos si es necesario, lo que haría volver a cambiar a mano el código cada vez que se genere y eso es altamente ineficiente. Por tanto se ha tratado el código como una caja negra que se irá haciendo mayor conforme se incluyan instrumentos en GlStudio (o módulos), pero es independiente del resto del proyecto. ¿Cómo se consigue esto?, la respuesta radica en la posibilidad que ofrece GlStudio para separar la función main del resto del código que genera y, embebiendo el contenido del main (que no cambia) en una clase más del proyecto, se puede controlar la aplicación sin necesidad de incluir código adicional.

Se tiene entonces la capacidad de desarrollar independientemente los módulos gráficos que representan instrumental en pantalla, ya que los desarrolladores no tienen que estar en el mismo proyecto; mientras sigan unas reglas para la creación de los módulos, el resultado será un paquete de archivos que únicamente habrá que incluirlo en GlStudio y generar de nuevo.

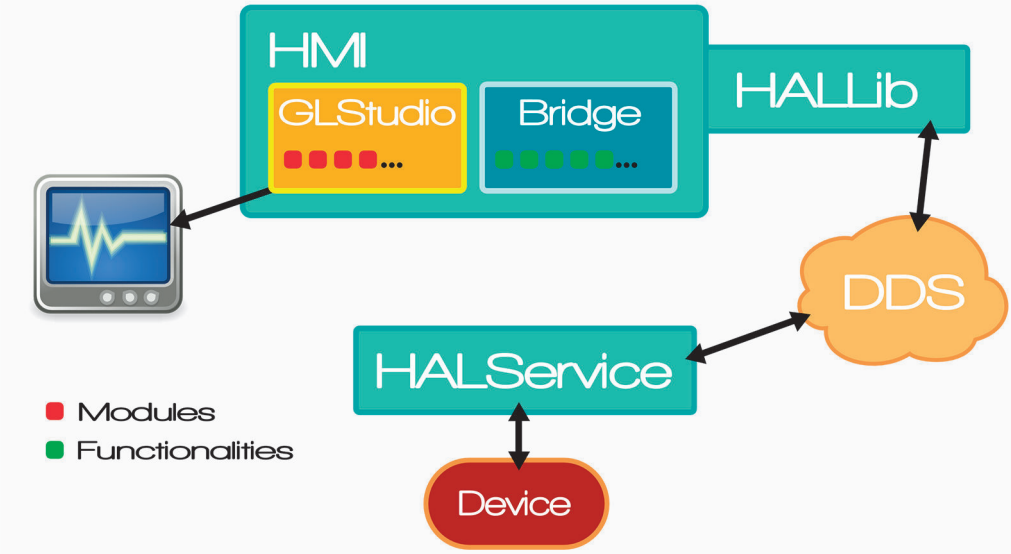

Figura 39: Arquitectura del HMI.

Con lo anterior se tiene la parte visual, pero no hará nada sin una entidad que la controle pasándole datos. Se crea entonces la entidad llamada *HMIBridge*, que hace de "puente" entre la parte visual y los datos. Ya que la información viene de la red mediante DDS, en el *HMIBridge* se irán creando las clases necesarias para la correcta obtención y manipulación de los datos. Por tanto toda funcionalidad adicional que se requiera en el HMI se incluirá en esta entidad, por ejemplo: la propia recepción de los datos haciendo uso de la librería de dispositivo, el filtrado de los valores (eliminación de ruido y valores incoherentes), el parseo de datos, y la inyección de los datos en la parte visual.

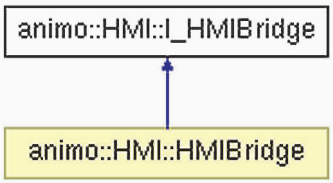

Figura 40: Jerarquía de HMIBridge.

Las demás clases incluidas dentro de este *namespace* son únicas y sin necesidad de herencias de ningún tipo:

- **Configuration**: Como en todos los demás módulos, esta clase genera el fichero de log. La diferencia esencial en este caso es que se crea en el directorio de instalación de la aplicación. En el caso de las librerías esto no es así ya que entonces se crearían los ficheros de log en el mismo directorio dónde se haya instalado dicha librería y en caso de instanciarse en varias aplicaciones se sobreescribirían, por no decir que con dos aplicaciones ejecutándose a la vez tendríamos error de acceso al fichero.

- **DataConverter**: Los datos enviados por DDS son de tipo *idl* y en éstos los valores se codifican como una cadena de caracteres ASCII, por lo que al recibirlos es necesaria la transformación de vuelta a valores *float*, que es el tipo de dato con el que se trabaja. En el caso del *float* se puede conseguir más precisión que la que es capaz de ofrecer el testbed, por lo que no es necesario recurrir al tipo *double*, que derrocharía memoria además de ralentizar la aplicación. Se lleva a cabo mediante los siguientes métodos:

  - **asciiToFloat**: El más sencillo, ya que únicamente traspasa la cadena de caracteres a dato de tipo flotante.
  - **asciiToFloatModule**: Calcula el módulo de tres valores que se le deben de pasar mediante argumento. Utiliza para ello el método anterior.
  - **radToAngle**: Los datos de ángulos recibidos están en radianes, mientras que la parte visual de la aplicación utiliza ángulos decimales por lo que es necesaria la transformación.s

- **Filter**: Clase necesaria para filtrar el ruido incluido en los datos, ya que para la representación visual mediante instrumentos virtuales, se hace necesaria una estabilidad en los grafismos los cuales se verían temblorosos si no se elimina. En el caso del HMI, sólo es necesaria la representación de hasta dos decimales, por lo que se filtran los demás y así se obtienen valores estables. Esto se consigue con los siguientes métodos:

  - **filterData**: Como su nombre indica, es la que ofrece la funcionalidad principal de la clase.

- **setAccuracy**: Método que provee la capacidad de elegir en todo momento que precisión va a ser requerida y por tanto el método anterior podrá filtrar el dato en consecuencia.
- **getAccuracy**: Método meramente informativo para comprobaciones de la precisión que se está utilizando.

- **RenewHMIData**: Es el nexo de unión entre la clase *HMIBridge* y la clase generada por GlStudio. Esta clase simplemente copia los valores recibidos por DDS a las variables públicas que cada instrumento virtual ofrece al usuario para leer la medida a representar.

4.5.2 GL Studio

Se ha hecho uso de la aplicación GL Studio (desarrollada por DiSTI) para el desarrollo de la interfaz [11]. Éste software ofrece una herramienta potente y sólida para diseñar e implementar instrumentos virtuales y garantizar la representación de los datos en tiempo real. Si bien el presente proyecto se centra en la monitorización y control de sistemas, otros ámbitos en los que GL Studio mejora la productividad es en la creación de aplicaciones de entrenamiento y aprendizaje virtual, dónde no es necesaria ninguna adquisición de datos puesto que el objetivo es simular un sistema real interactivo con el que un usuario pueda realizar prácticas sin tener acceso al equipo real.

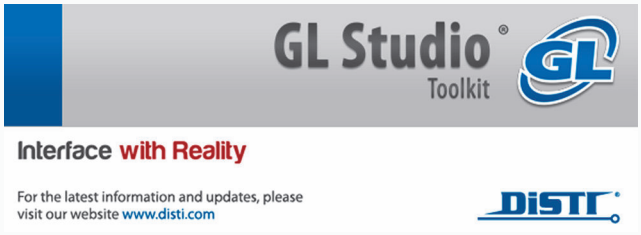

Figura 41: *Splash Screen* de GL Studio.

Al iniciar el programa, se muestra una interfaz clásica, con barras de herramientas, barras de menús y espacio gráfico de trabajo como viene siendo habitual en aplicaciones de escritorio. Está organizado por un sistema de pestañas, mediante las cuales se puede ir viendo las propiedades de los objetos, la programación asociada al proyecto creado, sus propiedades, las texturas y las propiedades relacionadas con la generación de código.

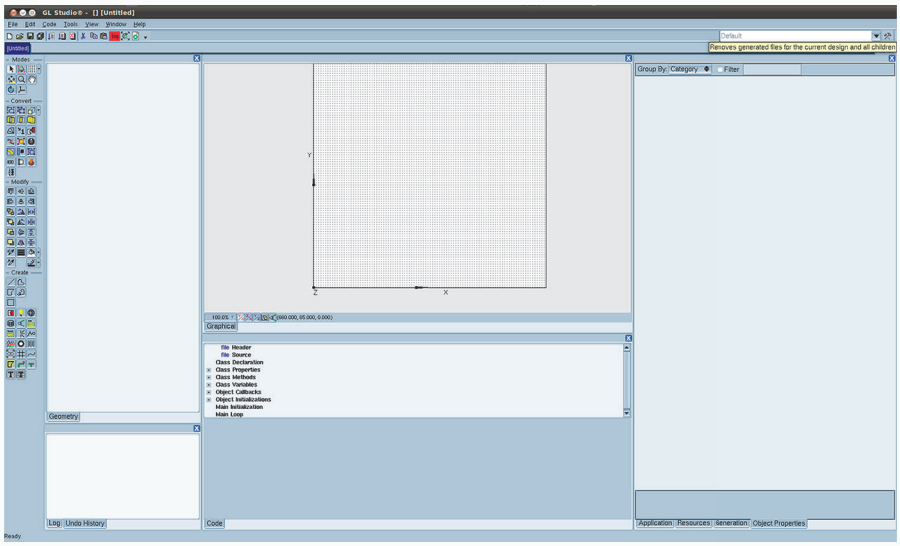

Figura 42: Entorno de trabajo de GL Studio.

Para elaborar una interfaz gráfica, GL Studio pone a disposición del usuario numerosas funciones que facilitan el diseño de los diferentes componentes tales como botoneras, indicadores de estilo digital o analógico, paneles con información en crudo, efectos 2D y 3D, etc. Después de que se haya maquetado la interfaz, se procede a generar el código fuente del diseño creado. En primer lugar se tienen los iconos agrupados bajo el menú *Modes*, que son las funciones más utilizadas ya que ofrecen las herramientas de selección de objetos, mover, ajustar tamaño de la textura a mano directamente, activar o desactivar la alineación a la rejilla, girar objetos, etc.

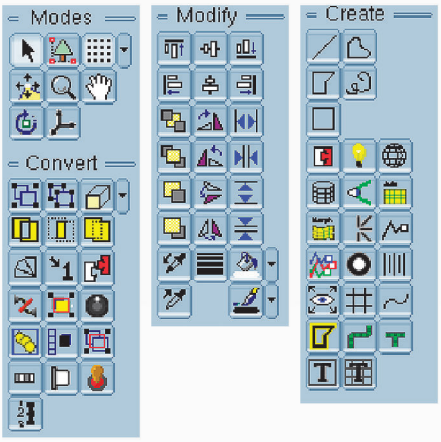

Figura 43: Menús para acceder a todas las funcionalidades nativas de GL Studio.

Por debajo, se encuentran el resto de menús lo cuales permiten crear los objetos avanzados anteriormente enumerados, conversión de tipos de objetos (agrupación, uniones lógicas de polígonos, extrusión, etc.) y ayuda al posicionamiento agrupados en el menú *Modify*.

En la pestaña *Application*, situada abajo a la derecha junto con las demás pestañas, se encuentran las propiedades del documento actual, pudiendo cambiar parámetros que modifican directamente el comportamiento de la interfaz en su ejecución. Entre estos parámetros, los más importantes son, a saber: la elección de aplicación de pantalla completa o enventanada, posición inicial en la pantalla de dicha ventana, su tamaño, posibilidad de escalado de ventana, borde de ventana (utiliza directamente las ventanas del sistema operativo), el título de la ventana, y su tamaño mínimo y máximo.

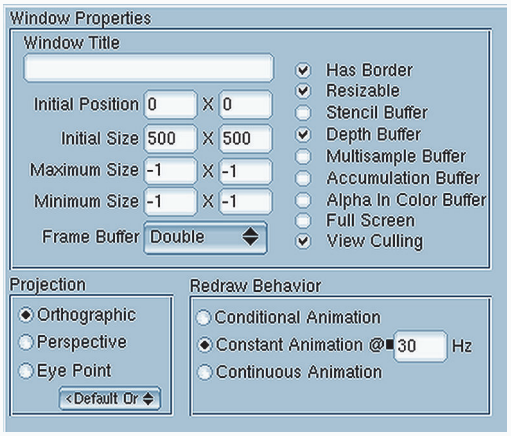

Figura 44: Pestaña de propiedades del documento actual.

Bajo la pestaña *Resources*, se da soporte para incluir las texturas, sonidos y tipografías que se quieran para poder personalizar la interfaz en gran medida . Referente a las texturas, podrán ser imágenes de mapa de bits JPG o PNG. Denotar que el hecho de no soportar imágenes vectoriales en las interfaces que genera supone un gran desventaja, ya que obliga al diseñador a tener que realizar una interfaz distinta para cada tipo de pantalla, tanto si se utiliza en pantalla completa o enventanada, ya que una de las metas de GL Studio, es la posibilidad de crear interfaces gráficas para dispositivos embebidos, cuyas pantallas difieren mucho en el tamaño de la pantalla del equipo de desarrollo.

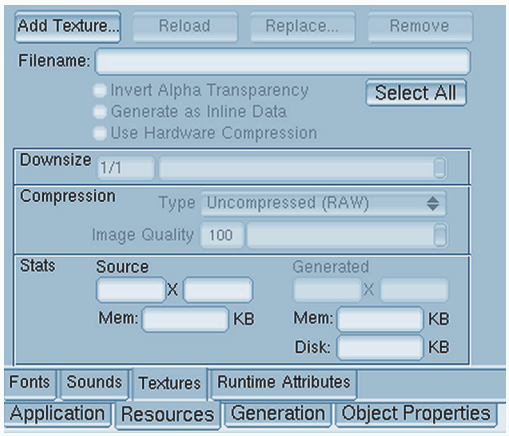

Figura 45: Funciones para la gestión de recursos.

Utilizando imágenes vectoriales, se daría un gran paso ya que diseñando una única interfaz, directamente se podría implementar en cualquier sistema, ya que los contenidos se escalarían adecuadamente pero sin perder calidad, cosa que no ocurre actualmente.

Conociendo las secciones anteriores, el usuario ya puede empezar la creación de los objetos necesarios para implementar por fin el diseño. En GL Studio, todos los elementos gráficos están soportados por un tipo de objeto base. Dicho objeto no es más que un polígono, el cual estará formado por tantos vértices como sean necesarios. Los objetos avanzados entonces, no son más que agrupaciones especiales de ciertos polígonos, a los que se les dan propiedades específicas para que su funcionamiento sea el deseado.

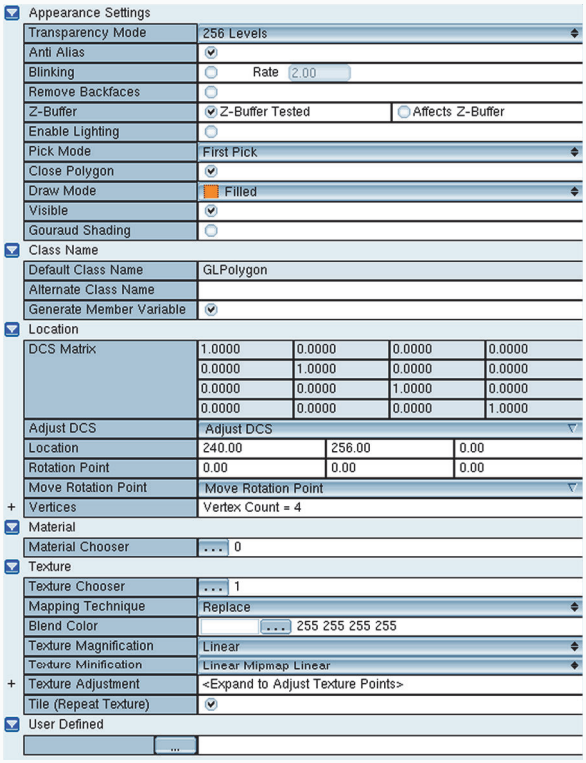

Figura 46: Propiedades de los objetos.

Por ejemplo, para crear un pulsador tipo botón (con efecto de pulsado, al igual que cualquier botón "Aceptar" de un formulario Windows), se agrupan dos polígonos: uno con la apariencia del botón sin pulsar, y el segundo con la apariencia del botón presionado o activo. El grupo resultante ya es directamente funcional, disponiendo en tiempo de ejecución de métodos que generan el efecto de pulsado activando y desactivando el polígono correspondiente.

Para la personalización de los parámetros de cualquier objeto, se encuentra la pestaña *Object Properties*, la cual muestra las opciones disponibles para el objeto actualmente seleccionado en la zona de trabajo. Se pueden ajustar entre otras cosas, la visibilidad del elemento, colores, parpadeo, posición exacta del objeto, textura, modo de textura, etc.

Por último, una vez creado el aspecto de la interfaz, se tienen las pestañas de *Code* y *Generation*, la primera de las cuales muestra por grupos las variables, métodos y propiedades que se van añadiendo para dotar de funcionalidad la interfaz. Decir que el código es únicamente para dar acción a los elementos, es decir, realizar animaciones,

giros, ocultar objetos, etc. por lo que si se genera y compila el código, se obtendrá una interfaz *dummy* con la que poder testar el diseño antes de seguir avanzando en el proyecto.

En la pestaña *Generation*, se precisa insertar el nombre que tendrá el documento/aplicación, el cual será utilizado para denotar la clase de C++ que genera GL Studio. Además es dónde se da a elegir si se desea una aplicación *standalone*, una clase de C++ independiente (para lo cual genera la función main en un fichero a parte, el cual se puede modificar para incluirlo en un proyecto mayor), o como un componente X-Window, el cual no tiene interés para el presente proyecto. Una vez completado, se genera el proyecto lo que crearán únicamente dos o tres archivos dependiendo de si se eligió la primera o la segunda opción respectivamente.

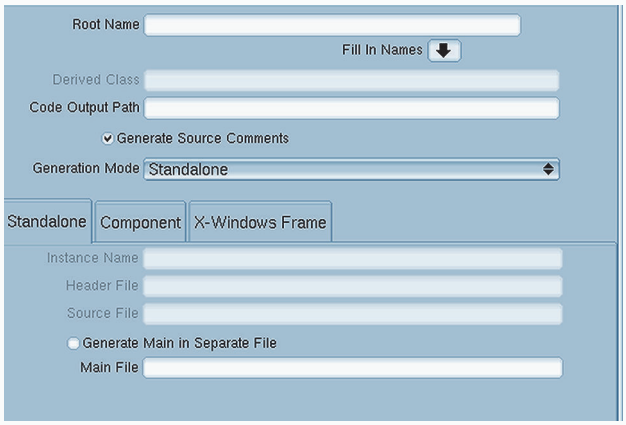

Figura 47: Pestaña *Generation*.

En el caso de la interfaz realizada, se escogió la segunda forma explicada, tratando la clase de C++ como una más del proyecto, e incluyendo el código de su función main en otra clase distinta para poder dar versatilidad y mantenibilidad al código. En éste caso los ficheros generados son un fichero de cabecera y dos de código fuente, siendo uno de ellos la implementación de la cabecera y el segundo la función main. Denotar que el contenido generado en la función main, no cambia nunca a menos que se especifiquen otras dimensiones para la aplicación, por lo que al añadir nuevos elementos gráficos únicamente se tienen que reemplazar los dos archivos que contienen la clase y su implementación.

4.5.3 Personalización de código

Evidentemente, una aplicación realizada íntegramente con la interfaz ofrecida por GL Studio tiene limitaciones. Para crear nuevas funcionalidades aparte de las de por defecto, se pueden crear nuevos métodos dentro del aplicativo, que al generar el código irán incluidos en la misma clase junto con todo lo demás. Esto da la opción hasta de incluir código propio de OpenGL, ya que está dentro del bucle de ejecución principal. Otra opción, y es como se ha procedido en este proyecto, es modificar el archivo fuente de la función main indicado en el apartado anterior, para incluirlo en otro método externo, y así formar una especie de "saco" en donde albergar la parte gráfica del programa y así controlarlo según convenga. En caso de necesitar una modificación en el código fuente de la clase creada por GL Studio, siempre es preferible intentar solventarlo mediante desde el propio GL Studio, ya que en caso contrario, habra que realizar la modificación cada vez que se genere el código.

4.5.4 Elección de instrumentos

Siendo el testbed un entorno para simular vuelos reales de aeronaves de todo tipo, existen numerosos instrumentos susceptibles de ser utilizados en el HMI. Una opción sería implementarlos todos, eso supondría la creación de varias pantallas e ir agrupando instrumentos por funcionalidad ya que mostrar todos a la vez por pantalla evidentemente inviable.

Si bien es cierto que todos son útiles, algunos lo son más que otros; es por ello que se decidió (gracias a la colaboración de opiniones expertas), seleccionar únicamente los más importantes, con un máximo de 5 instrumentos. El tope impuesto es por dos motivos: el primero es que todos los instrumentos seleccionados deben estar siempre presentes en pantalla, el segundo es que al poner un número mayor de instrumentos al operador le resulta casi imposible tenerlos en cuenta a todos. Los instrumentos que finalmente se han implementado en el HMI son el *horizonte artificial*, el *radioaltímetro*, el *anemómetro*, el *variómetro* y un *giróscopo direccional*.

- ▷ Horizonte artificial
  Es una representación de la línea de horizonte, coloreando las regiones que corresponden al suelo y al cielo para que sean fácilmente visibles. El instrumento irá girando la línea de horizonte según se incline la aeronave, por lo que muestra grá-

ficamente la información recabada del *roll* y el *pitch*. Además, es común incluir un indicador con el valor numérico del *roll* y del *pitch*, que suele representarse sobre un arco reglado y una escala vertical respectivamente.

Figura 48: Horizonte artificial.

▷ Radioaltímetro

Indica la distancia vertical absoluta que existe entre la aeronave y lo inmediatamente inferior, ya sea el suelo, un tejado, etc. En el mundo real funciona de una manera muy parecida a un radar, y mediante efecto Doppler se calcula la distancia. En el mundo virtual, y más concretamente en el testbed, indica la altura del quadrotor con respecto al suelo.

Figura 49: Radioaltímetro.

▷ Anemómetro

Se utiliza para la medición de la velocidad del avión respecto a la masa de aire que lo rodea. La toma estática recoge aire del lateral del avión y lo sitúa en el exterior de una cápsula aneroide y el tubo recoge aire de la parte frontal del avión y lo sitúa dentro. Esto hace que cuando el avión vuela más deprisa, la cápsula se expanda,

indicando más velocidad. La cápsula se expandirá más o menos dependiendo de la diferencia de presiones entre su interior y exterior. En el caso del testbed, sirve para indicar la velocidad del quadrotor puesto que al ser un sistema montado en interiores el aire se encuentra en reposo.
Dado que el sistema de cámaras devuelven valores de velocidad respecto a cada eje, para la representación final se tiene que calcular el módulo de los tres vectores correspondientes a los ejes X,Y y Z.

Figura 50: Anemómetro.

▷ Variómetro
Muestra la velocidad vertical del quadrotor, o mejor dicho indice el régimen de ascenso o descenso que tiene el avión. En su interior tiene una cápsula a la que le afecta el cambio de presión, y tiene un agujero micrométrico de forma que si el avión sube, la cápsula tiende a hincharse, y por lo tanto hace que la aguja suba.
El agujero empezará a soltar aire a medida que la cápsula se hincha y por lo tanto se estabilizará en algún punto con la aguja en una posición alta. Cuando se nivela el avión, el agujero terminará de dejar salir el aire, igualando la presión del interior de la cápsula con el exterior y la aguja volverá a su posición inicia la marcando 0. Al bajar el avión el proceso es el inverso.
No es muy descriptiva para los robots del propio testbed ya que pueden cambiar bruscamente de altura, pero en la simulación de otro tipo de aeronaves sirve para ver si se está excediendo los límites físicos soportables.

Figura 51: Variómetro.

▷ Giróscopo direccional
Proporciona al piloto la dirección del avión en grados magnéticos. Antiguamente también se usaba la brújula, pero ya no actualmente debido a que ésta se ve afectada por las variaciones magnéticas y, si el viento es turbulento, se vuelve aún menos precisa. En cambio, el indicador de rumbo es muy preciso y da al piloto una indicación mucho más fácil de interpretar, aunque todos los aviones deben disponer también de una brújula con la cual se toma referencia para ajustar el giro direccional.

Figura 52: Giróscopo direccional.

Para el diseño de todos estos instrumentos GL Studio ofrece dos posibilidades: crear los distintos elementos mediante las herramientas de dibujo propias del programa, o utilizar la compatibilidad con la herramienta de edición de imagen Photoshop (de la empresa Adobe). Evidentemente la última opción ofrece una alternativa de diseño muy potente, ya que de otra manera existen muchas limitaciones además de ser una tarea ardua lo cual no resulta productivo.

La compatibilidad que ofrece GL Studio, se basa en un programa comercial y no en uno de libre distribución por un sencillo motivo: los botones, *displays* LCD, etc. que

se pueden realizar, son grupos de capas que contienen los polígonos que lo conforman, por lo que a la hora de diseñar el instrumento en un programa de edición de imagen, se hace necesaria la capacidad de agrupar los polígonos para obtener el resultado deseado sin necesidad de retoque en GL Studio. Esta funcionalidad, aunque parezca trivial, en la fecha en que se escriben estas líneas no la incorpora prácticamente ningún otro software de edición de imagen ya sea *open source* o privativo, y los pocos que sí lo soportan no son tan potentes.

4.5.5 Maquetación

Es la planificación previa a la creación de todos los elementos gráficos que dan forma al HMI completo. Abarca desde bocetos a mano de las diferentes ideas, hasta la finalización de los mapas de bits que finalmente se incluirán en el proyecto. Durante el proceso, se utilizan los conocimientos sobre diseño gráfico ya explicados, además de investigar sobre el objetivo de la aplicación para así aclarar detalles de la funcionalidad, ubicación de elementos, apariencia, etc.

La idea buscada para este proyecto, es darle a la interfaz una apariencia futurista, ya que además de utilizar tecnologías del estado del arte como DDS, también está de moda en el sector aeronáutico denotar en los productos los avances conseguidos en I+D. Sin embargo, también se tiene en cuenta que el destino de la aplicación se encuentra en el ámbito profesional, y que un diseño puramente futurista pertenece más bien al mundo de los videojuegos y el ocio. Por tanto se entrecruzan varios objetivos: apariencia, funcionalidad y elegancia. Los dos primeros obviamente para conseguir un producto que rompa con el cliché de que aplicaciones serias requieren diseños austeros y complicados, y el último consigue casar los dos anteriores con la seriedad esperada de una aplicación de éstas características.

Figura 53: Búsqueda de ideas. Inspirarse viendo diseños de todo tipo.

La maquetación consiste en tener en cuenta todos los elementos que van a estar presentes, y pensar la mejor manera de ordenarlos y/o agruparlos. En el caso del HMI se requiere que siempre deben estar presentes los instrumentos, que debe de haber un menú de opciones, y que se irán añadiendo módulos de distinta funcionalidad como representación de trayectorias de los cuadricópteros por ejemplo. Hasta la fecha de realización de ésta memoria, se dispone de un módulo que representa en 2D el plano del testbed y que muestra unos iconos representativos de los cuadricópteros con la posición actual de cada uno de ellos en tiempo real. Por tanto, es obvio pensar que la pantalla va a estar dividida en dos: una para mostrar los instrumentos, y en la otra irá colocada la representación bidimensional (en adelante visor 2D).

Figura 54: Interfaz vacía del HMI en dónde se posicionarán los instrumentos.

Únicamente resta saber cómo incluir el menú, que como bien se ha explicado, debe ser fácilmente accesible y navegable. Es por ello que, y aprovechando las ventajas de una interfaz táctil, el menú se encuentra disponible bajo una pestaña desplegable. En estado de reposo, sólo se encuentra visible un botón en forma de pestaña. Al pulsarlo, aparece un panel prácticamente del tamaño de la ventana, mostrando la botonera que irá albergando los distintos módulos para mostrar, un botón que lanza el visor de log, y un último botón para salir de la aplicación. El panel se oculta al pulsar el botón de selección de un módulo, o bien pulsando nuevamente en el botón con forma de pestaña.

Figura 55: Interfaz completa.

De esta manera, se consigue un acceso a las funcionalidades con dos *clicks* exclusivamente, lo que agiliza el manejo de la interfaz; además, permanece lo más oculto posible mientras no sea útil por lo que no resulta molesto ni obstaculiza la visión de los instrumentos.

4.5.6 Creación del instrumental

Se detalla a continuación el procedimiento de creación de cada uno de los instrumentos implementados, tanto el diseño como la funcionalidad. Como es de esperar, la programación resultante es ínfima ya que GL Studio se encarga de facilitar la tarea de representación y animación de los componentes.

▷ Visor 2D.
Es el elemento más vistoso, ya que consigue crear una visualización sintética de lo que esta ocurriendo en el testbed en tiempo real. Lo primero a tener en cuenta, es conocer en qué punto se encuentra el origen de coordenadas, para poder así mapear las posiciones reales, que llegan a través de la red, en el punto de la pantalla dónde se debe mostrar gráficamente el icono representativo correspondiente.

Figura 56: Visor 2D con todas las entidades posibles del testbed.

▷ Horizonte artificial.
Es el indicador más utilizado, teniendo por tanto el privilegio de ser el que más espacio ocupe en pantalla, sin más razón que la de ser el primero en el que se fije la vista. Es un instrumento que muestra dos magnitudes y de varias formas a la vez. Dispone de un fondo que simula la línea de horizonte, separando un semiplano que se hace parecer al cielo, de uno que simula el suelo. El fondo se rota y se traslada según los valores del *roll* y del *pitch* respectivamente, para dar

la sensación del movimiento que sufre la aeronave. Ésta forma sin embargo, sólo es cualitativa, por lo que se añaden dos escalas para representar el movimiento de sendas marcas a lo largo de ellas y así mostrar el valor numérico de ambas magnitudes.

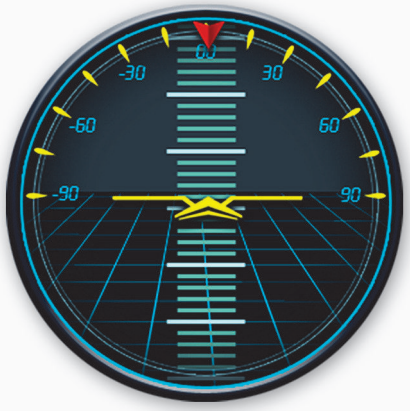

Figura 57: Horizonte artificial del HMI.

▷ Giróscopo direccional.

A imagen y semejanza de su homólogo en la vida real, se ha dispuesto como un dial circular, con marcas cada 2º. Se incluye también la marca de referencia ya que no tiene por qué coincidir el rumbo comandado y el rumbo que realmente sigue la aeronave. Dado que para hacer lo suficientemente visible el instrumento, debe tener un tamaño considerable, se necesita un espacio en pantalla que no está disponible. Por este motivo, y teniendo en cuenta que sólo interesa la porción del dial que marca el valor del rumbo, se ha ocultado el resto con el consiguiente ahorro de espacio sin mermar la funcionalidad del instrumento.

Figura 58: Giróscopo direccional del HMI.

▷ Radioaltímetro.
Dados como incógnitas los límites mínimo y máximo de la altura (ya que en sistema real se conocen, pero al realizar una simulación se pueden imponer límites acordes a cualquier situación deseada), se ha diseñado el instrumento como una rueda infinita, con marcas en forma de flecha grande para hacer coincidir con los valores enteros y marcas para los valores intermedios, y continuamente podrá ir girando para mostrar visualmente el aumento o disminución del valor nominal representado. De esta forma, el instrumento será válido para todo tipo de situaciones pero además ocupando menos espacio en pantalla ya que no hay necesidad de representar un círculo completo con la aguja correspondiente como en un radioaltímetro real.

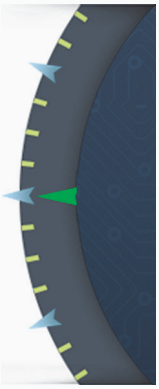

Figura 59: Radioaltímetro del HMI.

▷ Anemómetro.
Dada la similitud en funcionamiento con el anterior, se diseñó de la misma forma pero cambiando únicamente el color del indicador numérico, ya que de esta forma al cerebro le resulta más sencillo diferenciarlo con respecto al radioaltímetro aún teniendo la misma forma, evitando posibles confusiones al operario.

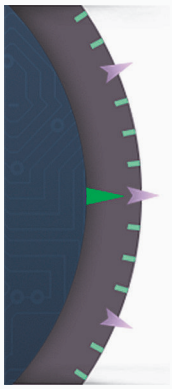

Figura 60: Anemómetro del HMI.

▷ Variómetro.
La idea en éste instrumento no es tanto la de mostrar el valor numérico concreto de la medida, sino más bien de dar cuenta de lo muy rápido o lento que se esté moviendo verticalmente el quadrotor. El motivo es la rapidez con la que un robot de estas características puede desplazarse, por lo que el valor numérico oscilaría muy deprisa entre un valor y otro por lo que no aporta información relevante al observador, mientras que representando una escala de colores, donde el verde significa viable y gradualmente pase a rojo que significa peligro, aporta la idea fundamental de los límites físicos de los dispositivos.

Figura 61: Variómetro del HMI.

▷ Botonera.
Ofrece una simple función: seleccionar el quadrotor deseado y mostrar su información en los instrumentos. La botonera ofrece a simple vista un pulsador para

cada uno de los distintos dispositivos posibles. Una opción sería utilizar un teclado numérico para elegir el dispositivo, pero resultaría en una acción más lenta, por lo que se recomendó poner tantos botones como fuesen necesarios para poder conmutar entre cuadricópteros con solo un *click*. La idea es simple: cada botón lo forman dos capas, una para el estado inactivo, y otra superpuesta sólo visible en estado activo, y así se recrea el efecto de iluminación para denotar que ha sido pulsado. Una opción descartada es la adicción de un sonido breve (como el que disponen los teclados de los teléfonos móviles), ya que con una respuesta luminosa es suficiente para el tipo de persona que se espera que maneje la interfaz. En otras palabras, en la interfaz se incluye lo estrictamente necesario para cumplir con los objetivos, y las señales acústicas se utilizan para personas con dificultades auditivas, que en cualquier caso verán la señal luminosa correspondiente al cambio de estado del botón por lo que añadir un sonido es redundante e innecesario.

Figura 62: Botonera de selección del HMI.

4.5.7 Adquisición de datos

A efectos de programación, el código generado por GL Studio no es más que una caja negra la cual presenta múltiples entradas de datos, siendo la pantalla la única salida. Si bien en GL Studio se programa cada elemento a gusto y necesidades del usuario, no es recomendable la implementación de los métodos de adquisición de datos directamente. La solución más acertada, es programar únicamente el comportamiento de los distintos elementos que conforman los instrumentos y dejar que las variables que alberguen el dato a representar sean las entradas de dicha caja negra.

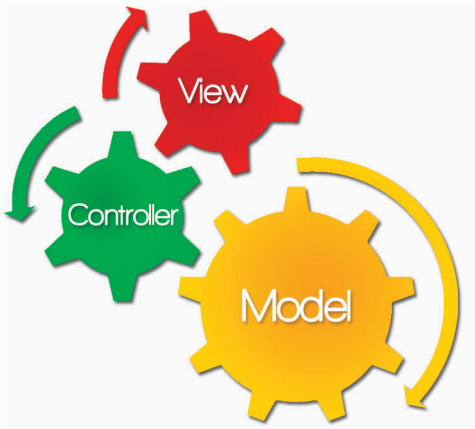

Figura 63: Modelo Vista Controlador (MVC).

Esto no es más que utilizar la metodología Modelo Vista Controlador (MVC), que ayuda a separar las responsabilidades de cada parte del proyecto haciéndolas independientes, lo que facilita la modificación o ampliación de las demás partes. Se tiene entonces que el "modelo" es el servidor de dispositivo, la "vista" estará formada por el código generado en GL Studio y el "controlador" lo formarán las clases que se utilicen para acomodar los datos recabados a las entradas de dicha caja negra.

Para llegar a esto, no hay más que incluir en el proyecto los archivos generados por GL Studio, y crear un objeto de dicha clase al que se le van a ir pasando los datos recibidos por DDS. Así, en caso de una modificación del formato de envío de los valores, siempre se puede actualizar el código fácilmente sin tener que recurrir a generar de nuevo el código de GL Studio.

4.6 Conclusiones

En esta sección se ha explicado detalladamente el proceso seguido durante la ejecución del proyecto, viendo cómo se crean las herramientas necesarias como lo son las diferentes librerías utilizadas, para posteriormente darles uso en la elaboración de una interfaz gráfica. Dicha interfaz muestra mediante instrumentos virtuales, diferentes datos relativos a vehículos aéreos (altura con respecto al suelo, rumbo, etc.), para que el piloto u operario los pueda controlar y monitorizar de forma sencilla.

Se ha visto por tanto, que es un proyecto software de grandes dimensiones al compaginar numerosos sistemas y arquitecturas. Esto implica largos tiempos de desarrollo

por lo que se hace necesaria una buena planificación de las tareas. Todo esto indica que ha sido acertado utilizar técnicas de programación ágil y metodologías de gestión de proyectos, sin los cuales no habría sido posible obtener un código tan limpio, fiable y robusto.