Escola Politècnica Superior
de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL DE FI DE CARRERA

TÍTOL DEL TFC: Intelligent Communication Gateway for Unmanned Airborne Systems

TITULACIÓ: Enginyeria Tècnica de Telecomunicació, especialitat Telemàtica

AUTORS: Daniel Giménez Verdugo
Joan Miquel Luque Oliver

DIRECTOR: Juan López Rubio
Pablo Royo Chic

DATA: 8 de Mayo de 2009

Títol: Intelligent Communication Gateway for Unmanned Airborne Systems

Autor: Daniel Jiménez Verdugo
Joan Miquel Luque Oliver

Director: Juan López Rubio
Pablo Royo Chic

Data: 8 de Mayo de 2009

Resumen

En la actualidad el uso de vehículos autónomos no tripulados está cada vez más extendido tanto a nivel industrial como a nivel comercial. Estas naves son capaces de realizar tareas que, por su peligrosidad, duración, carácter rutinario, realización en condiciones extremas o su coste económico, no pueden ser realizadas por personas o vehículos tripulados. El bajo coste y la gran versatilidad de estos vehículos los hace muy atractivos para la realización de multitud de tareas: misiones militares, exploración espacial y submarina, misiones de salvamento y reconocimiento, vigilancia perimetral, control forestal, etc.

Uno de los factores más importantes para la utilización de vehículos autónomos no tripulados es su gran capacidad de adaptación para la realización de múltiples cometidos. Con el fin de minimizar el precio de la puesta en producción de estos vehículos es importante que un mismo vehículo sea capaz de realizar el máximo número de misiones con unos ajustes mínimos. Para conseguir este propósito es necesario estandarizar las comunicaciones entre los sistemas de la nave, de manera que la creación de nuevas funcionalidades resulte más rápida y eficaz.

El control de las misiones realizadas por los vehículos autónomos no tripulados no se deja al azar. Están controladas por un equipo humano que, bien en tiempo real o después de la misión, analiza y estudia la información generada por el vehículo. Para ello es necesario que exista una comunicación entre el vehículo y la estación de control, ya sea mediante un medio cableado o en el caso de misiones con un radio de acción elevado, mediante medios inalámbricos.

El presente Trabajo Final de Carrera, Intelligent Communication Gateway for Unmanned Airborne Systems, es el diseño y implementación de un módulo de comunicaciones dentro de un middleware llamado MAREA diseñado específicamente para su uso en aeronaves autónomas no tripuladas.

Title: Intelligent Communication Gateway for Unmanned Airborne Systems

Author: Daniel Jiménez Verdugo
Joan Miquel Luque Oliver

Director: Juan López Rubio
Pablo Royo Chic

Date: 30 de marzo de 2009

Overview

Nowadays the use of autonomous unmanned vehicles is being more and more extended both at industrial and commercial levels. These vehicles are intended to make tasks that for his dangerousness, duration, routine character, accomplishment in extreme conditions or its economic cost, cannot be realized by humans or maned vehicles. The low cost and great versatility of these vehicles make them very attractive for the accomplishment of multiple tasks: military missions, space and submarine exploration, rescue and recognition missions, perimeters monitoring, forest control, etc.

One of the most important advantages of autonomous unmanned vehicles is their great reconfiguration flexibility for the accomplishment of multiple missions. To minimize the production cost of these vehicles it's important to make sure that the same vehicle will be capable to realize the maximum number of missions with a few minimal adjustments. To obtain this objective it's necessary to standardize the communications between the systems on board the vehicle, in order to establish new functionalities in a faster and effective way.

The control of the missions made by the autonomous unmanned vehicles is an important issue. They are controlled by a human team who, either in real time or after the mission, analyzes or studies the information generated by the vehicle. To do that it's necessary to have a communication link between the vehicle and the control station, being a wired link or in case of a great radius of action missions by wireless links.

This Final Project, Intelligent Communication Gateway for Unmanned Airborne Systems, is the design and implementation of a communications module inside a middleware called MAREA specifically designed for its use in autonomous unmanned aircraft.

INDICE

INTRODUCCIÓN

En la actualidad los sistemas autónomos no tripulados realizan misiones militares, de reconocimiento, de vigilancia perimetral, control del territorio, control forestal, etc. cuya dificultad, riesgo o coste son excesivamente elevados y no pueden ser asumidos por personas. Estos sistemas, cuyo uso es cada vez más común, presentan un diseño cerrado sin estandarizar que dificulta la integración de nuevos elementos y crea una aguda dependencia tecnológica.

Un grupo de profesores de la Universidad Politécnica de Catalunya (UPC), detectó la necesidad de crear un sistema de vuelo no tripulado. Para ello creo un grupo de trabajo denominado ICARUS (Intelligent Communications and Avionics for Robust Unmanned). La principal mejora del este nuevo sistema diseñado frente a los ya existentes es la implementación de un estándar de comunicación (Middleware Architecture for Remote Embedded Applications - MAREA), diseñado específicamente para vehículos no tripulados.
MAREA permite el funcionamiento de servicios distribuidos dentro del vehículo y representa un paso adelante en la evolución hacía una estandarización de los sistemas no tripulados.

El trabajo de ICARUS se centró, en una primera fase, en el diseño y desarrollo del estándar de comunicaciones en un escenario acotado físicamente por una red de ámbito local. Una vez implementado el estándar de comunicación de MAREA se hace preciso extenderlo a fin de hacerlo compatible con la utilización de más de una red.

El presente Trabajo Final de Carrera**, Intelligent Communications Gateway for Unmanned Airborne Systems (ICGUAS)**, es la especificación, diseño e implementación de un módulo de comunicación integrado en MAREA que permite el intercambio de información entre distintas redes. Este módulo permite la transmisión de datos entre el UAV (Unmanned Aerial Vehicle) y la estación base en tierra.

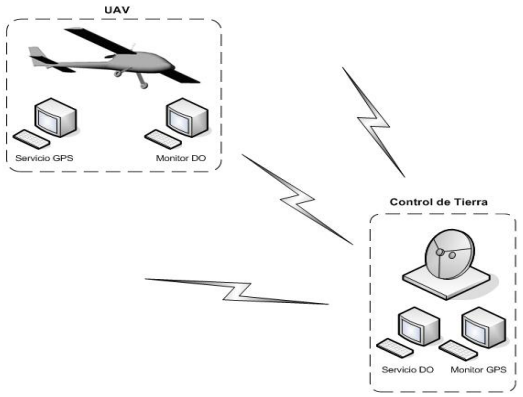


Fig. 0.1 Escenario del proyecto

Para el desarrollo de éste proyecto se formulan los siguientes objetivos concretos:

- Documentar MAREA
- Diseñar el Gateway de comunicaciones
- Implementar una primera versión del Gateway de comunicaciones

Se partirá del trabajo previo realizado por el grupo de investigación del proyecto ICARUS: MAREA y un conjunto de servicios, programados en .NET mediante Microsoft Visual Studio 2008 y Microsoft .NET Framework Version 3.5.

ICGUAS será capaz de controlar en todo momento el estado de los distintos enlaces existentes entre el vehículo y la estación base, escoger el mejor enlace para la transmisión de datos en función de la calidad de los enlaces disponibles, el tipo de datos y el destinatario.

Con el fin de trabajar de una manera cómoda, ordenada y eficiente, se ha dividido el trabajo en los siguientes capítulos: El Middleware: MAREA, Diseño, Implementación, Pruebas y un capítulo final de Conclusiones.

En el primer capítulo del proyecto se analiza exhaustivamente el protocolo de MAREA. Se ha entendido este trabajo previo como la base de todo el trabajo que posteriormente se ha realizado, por lo que se ha prestado especial atención en entender, analizar y formalizar toda la estructura, funcionalidades y recursos que ofrece MAREA, así como sus carencias y posibles mejoras.

Los capítulos dos y tres se centran en el diseño lógico e implementación del Gateway de comunicaciones respectivamente.

El cuarto capítulo del proyecto se ha dedicado a un estudio del funcionamiento y del rendimiento del código implementado así como la validación de su correcto funcionamiento conjuntamente con el protocolo de MAREA.

El último capítulo del proyecto es el de Conclusiones donde se dará una visión global de los objetivos cubiertos por el proyecto así como sus implicaciones medioambientales a corto y largo plazo.

Finalmente se adjunta una bibliografía con las referencias utilizadas en el proyecto y un apartado de anexos donde podrán encontrar todo el código del proyecto así como esquemas y diagramas detallados que, por su extensión, no se han incluido en la memoria del proyecto.

CAPÍTULO 1. EL MIDDLEWARE: MAREA

1.1. Introducción

Un *Middleware* se puede definir como una capa de software que ofrece una conectividad y servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas; una capa de software existente entre el sistema operativo y las aplicaciones que facilita su programación.

El principal beneficio que se puede obtener de un Middleware es proporcionar una API (Application Programming Interface) para realizar la comunicación entre las aplicaciones y el sistema operativo, y así no deber estudiar para que sistema operativo se está creando la aplicación, cuáles son los métodos de funcionamiento, ejecución, procesado, etc.

Existen muchos tipos de arquitecturas para el diseño de un Middleware, que serán meros patrones a la hora de desarrollar un middleware específico, por ejemplo: Remote Procedure Call, Publish/subscribe, Message Oriented Middleware, Object Request Broker, SQL-oriented Data Access, etc.

El Middleware sobre el que se desarrolla este proyecto se denomina MAREA (Middleware Architecture for Remote Embedded Aplications) y esta basado en la arquitectura Publish/Subscribe y ofrece otras funcionalidades: RPC (Remote Procedure Call), en este último el cliente realiza llamadas a procedimientos que están en máquinas remotas o MOM (Message Oriented Middleware) donde el clientes continua con sus tareas y los mensajes son guardados hasta que el cliente los solicita (véase [6]).

La arquitectura Publish/Suscribe se basa en el paso de mensajes entre un elemento que actúa como “publicador” de la información y un elemento que actúa como “suscriptor” de la misma. Se utilizan estos nombres porque la arquitectura se asemeja a los miles de sistema de suscripción existentes, por ejemplo, el Pay-per-View. En este cuando hay recursos disponibles se anuncian a la red y todos los usuarios interesados se Suscriben a dicho recurso y cuando existe nueva información el recurso la envía a todos y cada uno de los suscriptores registrados.

El objetivo principal de MAREA es dar soporte a un UAV (Unmanned Aircraft Avionics), un avión capaz de operar sin tripulación que necesita integrar una serie de servicios que puedan aportar operatividad y autosuficiencia a la aeronave además de permitir obtener diferentes datos: altura, meteorología, visión de una superficie desde el cielo, planes de vuelo, etc (véase [6], [7] y [8]). Con el fin de poder proporcionar todos estos servicios, el UAV necesita que todos sus componentes estén conectados entre si y puedan intercambiarse datos. En el **ANEXO 3** se puede observar la arquitectura interna de un UAV.

Otro de los objetivos del diseño del middleware del UAV es la comunicación entre la aeronave y la estación en tierra. Dicho objetivo es el que se persigue a lo largo de este proyecto.

1.2. Primitivas

Marea ofrece diversas primitivas de comunicación que permiten a los servicios comunicarse de diferentes formas según sus necesidades especificas. Hasta la fecha, se han diseñado e implementado cuatro tipos distintos de primitivas de servicio en MAREA: Eventos, Variables, Funciones y Ficheros.

La clase Event define las primitivas del tipo Evento que se utilizan para programar envíos de información ligados a incidentes no periódicos y por lo general impredecibles, como por ejemplo: detección de un obstáculo, desviación del rumbo, perdida de presión, error crítico del sistema, etc.

Por otro lado, la clase Variable define las primitivas del tipo Variables que se utilizan para programar envíos de infamación, deterministas y generalmente periódicos. Por ejemplo, en el caso del receptor GPS del UAV se enviará periódicamente la posición de la aeronave.

Las primitivas Funciones y Ficheros se implementaron después de la finalización de este proyecto y por eso no se incluyo su estudio en el mismo.

1.3. Arquitectura de clases

Siguiendo la arquitectura Publish/Subscribe, en la arquitectura del MAREA se define un publicador (servicio), y un suscriptor (monitor), que a través de diversos mensajes se intercambiaran información de manera eficiente a través de la red.

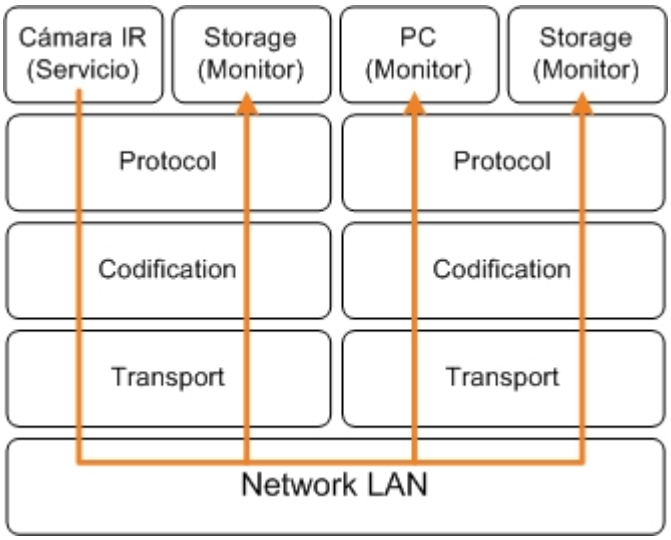


Fig.. 1.1 Arquitectura de clases.

Tal y como se observa en la Figura anterior (**Fig..** 1.1), bajo los servicios se encuentra el middleware dividido en 3 capas: Protocolo, Codificación y Transporte. Finalmente, debajo se encuentra la capa física que en el caso del ejemplo sería una red local o LAN (Local Area Network). La distribución de la información se realiza mediante los protocolos UDP y TCP, en función del tipo de mensajes que se desee enviar.

Algunos de los servicios que dispondrá el avión no tripulado serán: cámara infrarrojos, unidad de almacenamiento, unidad GPS, módulo de control, piloto automático, sensores de temperatura, inclinación, etc. Todos ellos se comunican entre si utilizando las primitivas de comunicación (Eventos, Variables, Funciones y Ficheros) ofrecidas por MAREA.

Como ya se ha mencionado anteriormente, la estructura del MAREA está dividida en tres capas o bloques funcionales: Protocolo, Codificación y Transporte. La división del protocolo en capas supone un conjunto de ventajas considerables: estructuración del contenido, facilidad de programación, la comprensión del código, etc., además esta división permite incluir soporte a varios tipos de codificaciones y de tipos de transporte en el mismo protocolo.

En el **ANEXO 3** se puede ver el diagrama de clases de MAREA, resumiendo cómo interactúan entre si todas las clases del middleware. En los siguientes puntos se detallan las características y funcionalidades de cada una de las capas del middleware y de cada una de sus clases:

1.3.1. Capa de Protocolo

Esta capa es la capa principal de las tres que conforman MAREA ya que es la que define como se intercambia toda la información dentro del protocolo. Esta capa posee la inteligencia para saber que hacer con los mensajes recibidos, modificar las listas de publicadores y suscriptores, etc. Además, esta capa se encarga de analizar los mensajes y manipularlos dependiendo de su contenido.

A continuación se describen brevemente las clases que conforman la primera capa del middleware MAREA:

- *ServiceContainer*: es la clase principal del middleware, en ella se contienen los demás objetos. Esta clase permite el paso de mensajes entre servicios y middleware, contiene las listas de servicios, variables y eventos disponibles, y dependiendo el mensaje que le llegue, realiza una función u otra.
  También se encarga de llamar a otras funciones de otras clases, como en el caso de arrancar servidores. Por último, incluye también el menú de operaciones disponibles en el modo consola.

- *ServiceManager*: esta clase es la encargada de iniciar y detener los servicios que se ejecutan sobre el middleware, conformar la lista de servicios que utilizará el ServiceContainer y cargar el fichero de configuración startup.xml.

- *Variable y Event*: estas clases definen los atributos y funciones que disponen las variables y los eventos. Se incluyen funciones para añadir, borrar y modificar la información de publicadores y suscriptores.

- *VariableManager y EventManager*: estas clases contienen las listas de publicadores y consumidores, y se encargan de modificarlas si es necesario. Además, proporcionan la información necesaria de cada variable y evento registrado en el sistema.

- *Scheduler*: esta clase trabaja como una FIFO de threads, donde cada vez que se quiere enviar variables/eventos por la red pasa a esta cola y así logramos evitar errores por buffers limitados.

- *DispatchObject*: Esta clase simplemente nos ofrece dos atributos necesarios para pasárselos a la clase anterior.

1.3.2. Capa de Codificación

La capa de Codificación tiene dos cometidos principales: preparar el mensaje para su envío y elegir el protocolo de transporte con el que se enviará.

Para preparar el mensaje para su posterior envío sobre la red se transforma la clase Message en una cadena de bytes mediante una clase de C# llamada BinaryFormater. Para ello, la clase Message se define como [Seriarizable]. Este atributo nos permite serializar la clase en un extremo de la comunicación y deserializarla en el otro. De esta manera se puede recuperar el mismo mensaje idéntico.

Por otra parte, la capa de Codificación se encarga de escoger el protocolo de transporte con el cual se enviará el mensaje así como la dirección de red y los puertos a utilizar. Esta elección se realiza en función del tipo de mensaje. Por

regla general los Eventos, Funciones y Ficheros se envían mediante TCP mientras que las Variables y mensajes de protocolo se envían mediante UDP.

A continuación se describen brevemente las clases que conforman la segunda capa del middleware MAREA:

- *NetSerializationCoder*: contiene el algoritmo necesario para pasar objetos o clases a un vector de bytes o viceversa de manera óptima. Es el punto intermedio entre la capa superior y la de transporte.

- *Message*: se trata de una clase genérica que contiene la definición de los diferentes tipos de mensajes que se utilizará en el middleware. En el siguiente punto se verá qué tipo de mensajes hay, en qué se diferencia cada uno y para qué se utilizan.

1.3.3. Capa de Transporte

La tercera sección en la que se divide el middleware es la capa de transporte. Como bien indica su nombre es la encargada de transportar los datos a través de la red. Mediante esta capa se enviará por red la secuencia de bytes resultante de la serialización. En función del tipo de mensaje a transmitir se enviará mediante el protocolo UDP o TCP

El protocolo UDP (User Datagram Protocol) es un protocolo del nivel de transporte basado en el intercambio de detagramas. Este protocolo permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera. Tampoco tiene confirmación, ni control de flujo, por lo que es posible que los paquetes se adelanten unos a otros; del mismo modo tampoco se sabe si ha llegado correctamente, ya que no hay confirmación de entrega o de recepción (véase [1] y [2]).
Aun así, su utilización es interesante en muchos de los escenarios planteados en este proyecto donde las funciones de retransmisión o control de flujo no son necesarios. Además, la posibilidad de hacer llegar la misma copia del mensaje a múltiples destinatarios mediante direcciones de broadcast o multicast permiten un ahorro de ancho de banda muy importante en contraposición a TCP. Por regla general se utilizará UDP para el envío de los mensajes de protocolo y de las Variables.

TCP (Transmisión Control Protocol) es otro protocolo de nivel de transporte utilizado ampliamente en redes de comunicación. Este se basa en el intercambio de segmentos y proporciona fiabilidad en la comunicación mediante la confirmación de recepción, control de flujo, control de secuencia, etc. (véase [1] y [2]). Estas características, al contrario que UDP, hacen al protocolo TCP muy interesante para el envío de mensajes con una criticidad elevada dentro del protocolo de MAREA como puedan ser los Eventos, Funciones, Ficheros y alguno de los mensaje de protocolo.

A continuación se describen brevemente las clases que conforman la tercera capa del middleware MAREA:

- *TCPTransport*: clase donde se define las funciones y threads que sirven para arrancar un servidor continuo y un cliente para la hora de tener que enviar mensajes por la red. Estas conexiones se realizan siempre por TCP.

- *UDPTransport*: clase identica a la anterior pero con la diferencia que se usan clientes y servidores utilizando UDP. Actualmente en el middleware se usa el modo Broadcast, con lo que se necesitan especificar esa dirección o construir una clase que la obtenga

- *TransportAddress*: esta clase se utiliza en las tres capas de MAREA, aunque es en la de transporte donde desarrolla su cometido principal: indicar que tipo de transporte se va a utilizar para enviar el mensaje, la dirección IP y puerto del destino.
- *TransportMultiplexor*: se trata de una clase esencial para la comunicación dentro del protocolo de MAREA ya que es la encargada de decidir el tipo de transporte que se utilizará, TCP o UDP. Esta elección se basa en el TransportAddress que viene de la clase NetSerializationCoder.

- *Transport*: permite calcular la dirección de transporte Broadcast para una conexión UDP.

1.3.4. Clases genéricas

Además de las 3 capas anteriormente detalladas, MAREA dispone también de una serie de clases de funcionalidad general utilizadas por todas las capas del middleware. A continuación se describen brevemente dichas clases:

- *Message*: contiene la definición de los diferentes tipos de mensajes que se utilizará en el middleware. En los siguientes apartados se detallan dichos mensajes.

- *ServiceDescription*: esta clase proporciona información acerca de los servicios como por ejemplo: el nombre o las variables que utiliza.

- *Service*: esta clase contiene la definición servicio. Esta clase define cuatro funciones principales: Run(), Start(), Stop(), VariableChanged(), EventFired() y GetDescription().
  Las dos funciones más importantes son VariableChanged() y EventFired(), que son las encargadas de mandar el mensaje hacia el middleware para su posterior tramitación.

- *RemoteService*: esta clase es la encargada de definir como interactúa el middleware con un servicio remoto, es de decir, que no se está ejecutando en la misma máquina. Cuando se recibe un mensaje de

dicho servicio, MAREA copia la información y la guarda en una lista como un RemoteService. De manera que cuando se quiere intercambiarse información, se llamará a esta clase mediante las funciones VariableChanged() y EventFired() y a partir de este punto el middleware envía por la red el mensaje necesario.

Por otro lado, existen también una serie de clases definidas como interfaces, donde se determinan las funciones y atributos obligatorios que deben llevar las clases utilizadas en el middleware. Estas interfaces han sido creadas con el fin de facilitar el desarrollo de todo el software y definir unas pautas generales a seguir.

Por ejemplo, existe la clase interfaz ICoder, que está asociada a la clase NetSerializationCoder. Y en ella se específica que obligatoriamente ésta última clase debe llevar las funciones Start(), Stop(), Send() y DataReceived(). Esto no limita el número de funciones que llevará la clase, ya que además se pueden programar otras.
Aparte de ICoder, entre otras también existen IServiceContainer (asociada a ServiceContainer), IService (asociada a Service), ITransport (asociada a TransportMultiplexor), etc.

1.4. Mensajes

Con el fin de intercambiar información entre los distintos sistemas que conforman el middleware el protocolo, de MAREA define una serie de mensajes. Estos mensajes están definidos dentro de la clase Message del MareaInterface. A continuación, se describen los mensajes definidos actualmente en el middleware:

1.4.1. Publish

Este mensaje es el utilizado por los servicios para dar a conocer las variables y eventos de que dispone. Cuando cualquier servicio quiere darse a conocer a los demás elementos del Middleware, hace un RegisterService() a la clase ServiceContainer y éste crea el mensaje Publish, para ser emitido por la red y de este modo informar a todos los posibles suscriptores. La siguiente Figura (**Fig. 1**) representa los campos del mensaje Publish:

Fig. 1.2 Mensaje Publish

El campo “name” está formado por un string y representa el identificativo de la información publicada (p.e.: Altura). El campo “primitive” es un objeto del tipo

"Primitive" definido como una enumeración que indica el tipo de información publicada. Puede tomar los siguientes valores: Variable, Event, Invocation, File. El campo "control" es del tipo "TransportAddress". Este campo transporta la dirección de control del servicio que publica la información. Esta dirección es única para cada servicio del contenedor y lo identifica inequívocamente. Como se verá, esta dirección es la utilizada por el servicio para enviar y recibir información de control.

1.4.2. Suscribe

Este mensaje es utilizado por los monitores del sistema para informar a un servicio que desean recibir la información que suministra. Cuando un monitor quiere suscribirse a un publicador ya conocido, llama la función Suscribe() de la clase ServiceContainer y éste último crea el mensaje Suscribe, se envía por la red. La siguiente Figura (**Fig. 1.**) representa los campos del mensaje Suscribe:

Fig. 1.3 Mensaje Suscribe

De la misma manera que el mensaje Publish, el campo "name" está formado por un string y representa el identificativo de la información a la cual se desea suscribir el monitor (p.e.: Altura). El campo "primitive" es un objeto del tipo "Primitive" definido como una enumeración que indica el tipo de información a la cual se desea suscribir el monitor. Puede tomar los siguientes valores: Variable, Event, Invocation, File. El campo "control" es del tipo "TransportAddress". Este campo transporta la dirección de control del monitor que Suscribe la información. Como se verá, esta dirección es la utilizada por el monitor para enviar y recibir información de control.
El campo "data" se define como una lista de "TransportAddress". Cada elemento de esta lista es una dirección de transporte por la cual el monitor puede recibir la información suscrita. Como se verá, es el publicador quien decide por cual de estas direcciones se envía finalmente la información.

1.4.3. SuscribeACK

Este mensaje es enviado por el publicador como respuesta a un mensaje de Suscribe. La siguiente Figura (**Fig. 1.**) representa los campos del mensaje SuscribeACK:

Fig. 1.4 Mensaje SuscribeACK

De la misma manera que el mensaje Publish, el campo “name” está formado por un string y representa el identificativo de la información a la cual se desea suscribir el monitor (p.e.: Altura). El campo “primitive” es un objeto del tipo “Primitive” definido como una enumeración que indica el tipo de información a la cual se desea suscribir el monitor. Puede tomar los siguientes valores: Variable, Event, Invocation, File. El campo “data” es del tipo “TransportAddress”. Este campo transporta la dirección de datos por la cual el monitor (o suscriptor) recibirá la información del publicador.
El mensaje SuscribeACK es un mensaje enviado por el publicador en respuesta de una solicitud de suscripción de un monitor. Por tanto, es siempre el publicador quien decide por que dirección enviará los datos.

1.4.4. Discover

Este mensaje es utilizado por los monitores cuando quieren recibir información de una primitiva en concreto de la cual no tienen constancia que esté publicada. Cuando el publicador recibe un mensaje de Discover de una de sus variables, esté vuelve a enviar el Publish correspondiente y se inicia el proceso de suscripción que posteriormente se detallará. La siguiente Figura (**Fig. 1.**) representa los campos del mensaje Discover:

Fig. 1.5 Mensaje Discover

Como se puede observar, este mensaje solo incluye el string “name” y la enumeración “primitive” que, como en el resto de los mensajes, puede tomar los valores: Variable, Event, Invocation, File. Del mismo modo, el campo “name” representa el identificativo de la información a la cual se desea suscribir el monitor. Si algún publicador tiene dispone de la información solicitada se iniciará el proceso de suscripción que se detallará más adelante.

1.4.5. UnPublish

Este mensaje es el utilizado por los servicios para indicar que dejan de subministrar información de una primitiva en concreto. La siguiente Figura (**Fig. 1.**) representa los campos del mensaje UnPublish:

Fig. 1.6 Mensaje UnPublish

Este mensaje contiene los mismos campos que el mensaje Publish y es tratado a la inversa. En caso de recibir un UnPublish, el Contenedor eliminará la primitiva en cuestión de su lista de publicadores. El campo “control” es igualmente necesario ya que es posible que distintos publicadores ofrezcan la misma información simultáneamente. Dado que la dirección de control es única para cada servicio se evitan incoherencias con este campo.

1.4.6. UnSuscribe

Este mensaje es el utilizado por los monitores para indicar que ya no desean seguir recibiendo información de una primitiva en concreto. La siguiente figura (**Fig. 1.**) representa los campos del mensaje UnSuscribe:

Fig. 1.7 Mensaje UnSuscribe

Este mensaje contiene los mismos campos que el mensaje Suscribe y es tratado a la inversa. En caso de recibir un UnSuscribe, el Contenedor eliminará la primitiva en cuestión de su lista de suscriptores. El campo “control” es igualmente necesario ya que es posible que distintos monitores estén suscritos a la misma información simultáneamente. Dado que la dirección de control es única para cada servicio se evitan incoherencias con este campo. Además, el campo “data” especifica la dirección de datos por la cual ya no se desea recibir información ya que es posible que un mismo servicio se suscriba varias veces a la misma información.

1.4.7. SlowData

Este mensaje es el utilizado para transportar los datos de las variables. La siguiente figura (**Fig. 1.**) representa los campos del mensaje SlowData:

Fig. 1.8 Mensaje SlowData

Identicamente al resto de mensajes, el SlowData contiene el campo “name” y el campo “primitive”. El campo “data” es un objeto genérico donde se podrá transportar cualquier tipo de información: integer, long, booleano, string, imágenes, etc.

1.4.8. Otros mensajes

Así mismo, la arquitectura de MAREA define más mensajes dentro de su protocolo. Muchos de ellos proceden de versiones anteriores y han pasado a un segundo plano o se especificaron para futuros usos. Su explicación sobrepasa los límites de este proyecto y por eso tan solo los enumeraremos: DialogueError, Register, Unregister, Notify, CallFunction, ReturnFunction, Data, Do, ProtocolCommand, Return, AskForService, ReturnAskForService.

1.4.9. Nuevos mensajes

En un futuro se prevé la creación de más tipos de mensaje destinados al transporte de datos. La inclusión de nuevos tipos de mensaje permitirá una mejora en la gestión de los mensajes y la calidad de servicio ofrecido.

En el siguiente capítulo del proyecto se detalla el mensaje GatewayMessage que, como se verá, es utilizado por el Gateway para encapsular información.

1.5. Protocolo

Como ya se ha visto en los apartados anteriores, el protocolo de MAREA está dividido en tres capas distintas y utiliza mensajes especiales para transportar la información entre ellas. Estos mensajes fluyen a través de las clases del middleware y son enviadas por la red hasta su destinatario. En el siguiente apartado se explica detalladamente como interactúan todas las clases en el paso de los mensajes.

Con el fin de facilitar la comprensión del documento, el siguiente apartado se estructura siguiendo el flujo lógico de la información dentro del middleware. Tomando como punto de inicio de la comunicación el servicio que genera la información. Se recorren todas las capas por las cuales transcurren los datos hasta llegar al monitor de destino. En la siguiente figura (**Fig. 1.**) se puede observar la pila de protocolo del MAREA que se utiliza en este apartado:

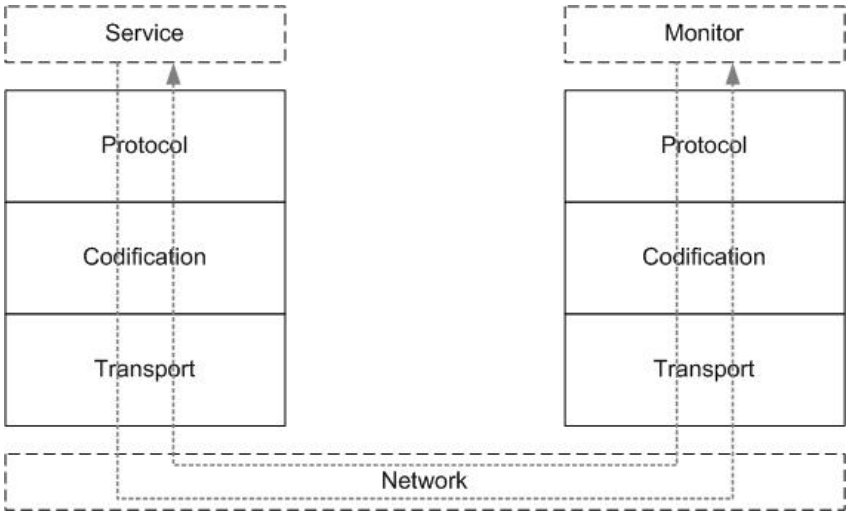


Fig. 1.9 Protocolo a nivel de capas

Como es obvio, la aplicación diseñada trabaja en la capa de aplicación del modelo OSI. En la figura anterior (**Fig. 1.**) el cuadro inferior etiquetado como "Network" representa las capas 1-6 del modelo OSI. Esta relación podemos observarla en la siguiente figura (**Fig. 1.10**):

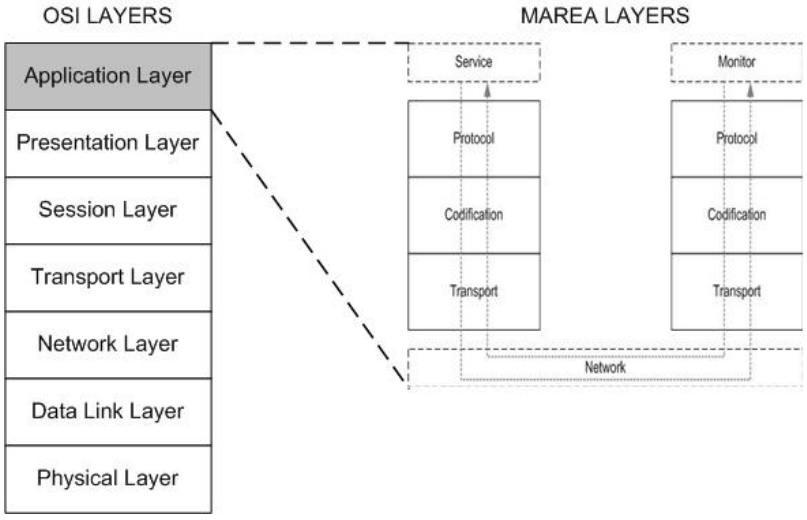


Fig. 1.10 Modelo OSI – Network

1.5.1. Puertos de comunicación

Como ya se ha explicado en varios puntos anteriores, una de las funciones principales de un middleware es la de facilitar la comunicación entre servicios distribuidos en un mismo sistema. Para ello, todos los contenedores que se ejecutan en MAREA tienen una serie de puertos de comunicación establecidos por defecto. Estos puertos permiten la comunicación entre todos los contenedores del sistema sin necesidad de establecer previamente una negociación. Dado que en el protocolo de comunicación de MAREA se utiliza tanto UDP como TCP, todos los contenedores tienen un puerto disponible para recibir mensajes mediante estos dos protocolos. Los puertos en cuestión son

los 11811 y 11000 respectivamente. Por norma general, los puertos UDP se utilizan para el envío de Variables y los TCP para el envío de Eventos, Funciones y Ficheros.

La arquitectura de MAREA define que cada servicio/monitor está identificado mediante una dirección de control. Esta dirección permite identificar todos los servicios activos del sistema y al mismo tiempo, es la dirección utilizada por esos mismos servicios para recibir y enviar información de control.

Análogamente, cada servicio define una o varias direcciones de datos. Estas direcciones definen con que dirección el servicio enviará/recibirá información por la red. Por ejemplo, un servicio que publica una variable, podría establecer dos tipos distinto de direcciones de datos: UDP Broadcast y TCP. De este modo, cada monitor que decide recibir esa información, puede escoger de qué modo desea hacerlo.

Así pues, podemos definir cada servicio/monitor del sistema mediante sus correspondientes direcciones de control y de datos. Toda esta información se comparte entre los servicios mediante los mensajes de Publish y Suscribe durante el proceso de publicación y suscripción de información que se explica a continuación.

1.5.2. Publicar información

Como ya se ha explicado anteriormente, una de los objetivos del middleware es facilitar la comunicación entre los programas que se ejecutan en las capas superiores. Cuando un servicio dispone de información y quiere anunciarla a la red, utiliza las funciones de las que dispone MAREA para hacerlo. Para ello, el servicio únicamente debe indicar a su Contenedor que está disponible para subministrar información mediante la función RegisterService(Service).

Cuando el ServiceContainer trata esta función, procede a registrar la variable (o evento) que el servicio quiere publicar en la red. Para ello utiliza las funciones PublishVariable(Service) y AddPublisher(Service) de las clases VariableManager (EventManager) y Variable (Event) respectivamente. Una vez finalizado este proceso el Servicio y las variables/eventos que puede suministrar ya están registrados localmente en el middleware y están listos para utilizarlos. Como veremos más adelante, el registro de publicadores y suscriptores difiere en función de si se trata de un publicador/suscriptor local o remoto. El siguiente paso en el protocolo es anunciar la disponibilidad del Servicio al resto del sistema. Este proceso se inicia en la clase Variable.

Una vez añadido el nuevo servicio en la lista de publicadores (función AddPublisher), se indica al ServiceContainer que se debe enviar un mensaje indicando la presencia de un nuevo servicio. Como ya se ha explicado en los apartados anteriores, este mensaje es el Publish y contiene el nombre de la variable publicada y la dirección de control del servicio que la genera. Una vez creado el mensaje, se pasará de la capa de Protocolo a la de Codificación mediante la función SendMessage(Message).

En esta capa, se transforma el mensaje en una serie de bytes mediante la función Seriealize(Message). Esta función transforma los datos estructurados de la clase Message en un string de bytes dispuestos para su envío por red. Además de convertir la clase Message en bytes, esta clase define que tipo de transporte se utilizará para la transmisión de los datos. Como ya se ha visto anteriormente, se han definido dos tipos posibles de transporte, UDP Broadcast y TCP. En el transcurso del proyecto se define una nueva clase de transporte, SerialTransport, que nos permite enviar información a través de la interfície del radio módem. La información conformada por Protocolo, dirección y puerto se define en la clase TransportAddress. El objeto TransportAddress se subministrará a la capa de transporte juntamente con los datos serializados.

Los mensajes del tipo Publish se envían siempre por Broadcast. Esto garantiza que todos los contenedores de la red están correctamente informados de todos los posibles publicadores del sistema. Los mensajes UnPublish se envían del mismo modo.

Cuando los datos pasan de la capa de Codificación a la de Transporte mediante la función Send() se pasa el TransportAddress y los Datos a la clase TransportMultiplexor. Como su nombre indica, esta clase escoge el tipo de transporte en función de los datos del objeto TransportAddress asociado a la información. Una vez elegido el tipo de transporte a utilizar, se envían los datos por red mediante la función Send(Datos) que tienen todos los objetos definidos como ITransport.

La siguiente figura (**Fig. 1.2**) esquematiza el flujo de la información que se acaba de explicar:

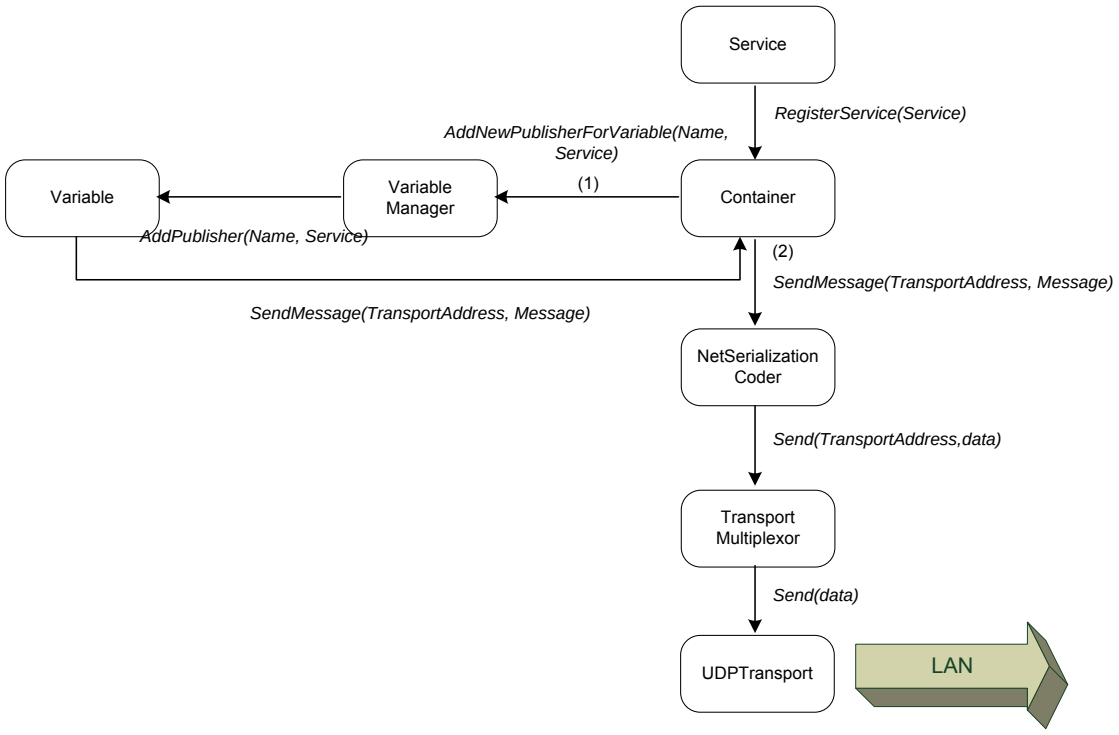


Fig. 1.2 Envío de un paquete Publish

Cuando un mensaje Publish es recibido por otro contenedor del sistema, se realiza el proceso inverso.

En primer lugar, el mensaje recibido es procesado por la clase UDPTransport (ya que el Publish se ha enviado por UDP Broadcast) dentro de la capa de Transporte. Esta clase desempaqueta el mensaje recibido mediante el protocolo UDP y extrae los datos en forma de secuencia de bytes. Esta secuencia, juntamente con su longitud, se pasará a la capa de Codificación que realiza el proceso inverso de la Serialización. Es decir, convierte la secuencia de bytes recibidos en un objeto del tipo Message.

Una vez obtenido de nuevo el mensaje original, se pasa a la capa de Protocolo que lo atenderá. Esta primera parte del procesado es análoga para todos los mensajes del middleware. En la siguiente figura (**Fig. 1.3**) se puede observar el flujo de datos recibidos que se acaba de explicar:

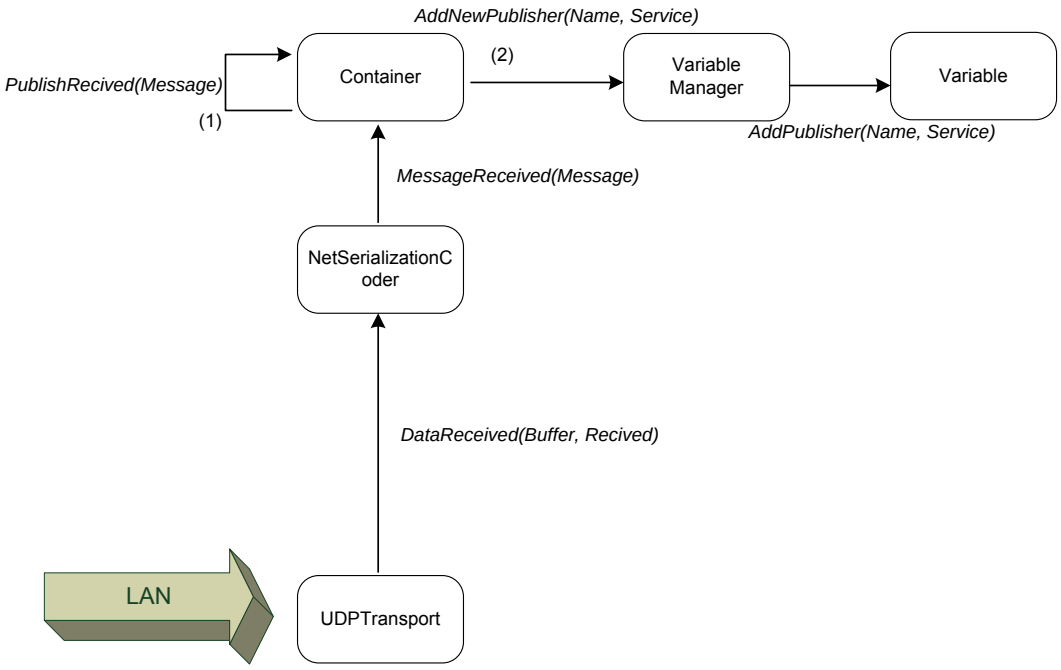


Fig. 1.3 Recepción de un paquete Publish

Una vez en la capa de Protocolo, el mensaje es procesado en función de su tipo. En el caso que nos ocupa, los mensajes Publish, en una primera versión del protocolo eran ignorados salvo que algún Monitor hubiera solicitado información de dicha variable. En una segunda versión del protocolo, se modificó esta actuación por la de almacenar toda la información de los Publish, se hubiera solicitado o no, eliminando así la repetición de envío de mensajes por la red y el correspondiente ahorro de ancho de banda y recursos.

Así pues, cuando un mensaje Publish llega a la capa de transporte proveniente de la red, se inicia un proceso de registro similar al explicado al principio del

apartado. Como ya se ha comentado, el sistema hace distinción entre los publicadores/suscriptores locales y remotos, entendiendo local a los servicios que se ejecutan en el mismo contenedor y remoto en todos aquellos que se ejecutan en otros contenedores. Se hace evidente, que la comunicación con un publicador/suscriptor local se realiza mediante funciones de Sistema Operativo, mientras que la comunicación con un publicador/suscriptor remoto se hace por red.

Como vemos en la figura anterior (**Fig. 1.12**), el registro de un mensaje Publish remoto es similar al de un Publish local. La diferencia reside a la hora de añadir la variable a la lista de publicadores. Cuando se ejecuta la función AddPublisher(Service), el sistema detecta que dicho servicio no se ejecuta en el contenedor local y que por tanto es remoto. Así pues, anota la dirección de control de dicho servicio en la lista para poder establecer comunicación vía red con ese servicio. En el punto (1) de la siguiente figura (**Fig. 1.13**) se puede observar el proceso de registro del mediante el mensaje Publish en ambos lados del sistema (local y remoto):

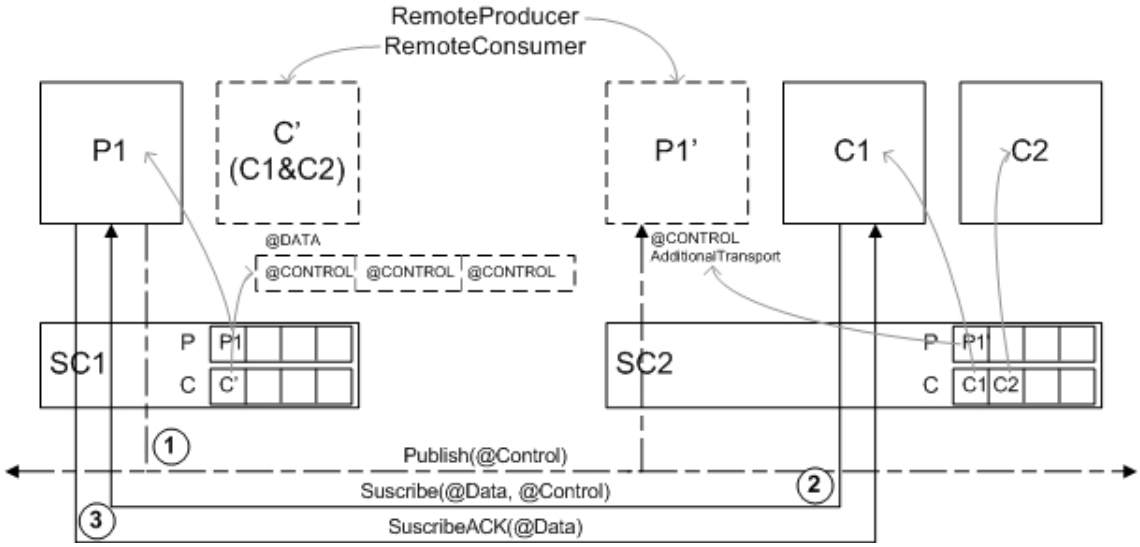


Fig. 1.4 Esquema Publicación/Suscripción

La figura anterior (**Fig. 1.4**) representa todo el proceso de publicación y suscripción de una variable. En lo que se refiere a la publicación de variables, se puede observar que cuando un publicador (Producer), en este caso P1, envía un Publish, en local se crea un registro con la información de P1 mientras que en el otro extremo se crea un registro con la dirección de control de P1 al que llamamos P1'. Este nuevo registro es lo que se denomina un RemoteProducer o Publicador Remoto.

1.5.3. Suscribir información

La suscripción es el proceso por el cual un contenedor informa que desea recibir una cierta información. Este proceso se realiza a nivel del protocolo

MAREA y es transparente para los servicios y monitores implicados en el proceso.

El proceso de suscripción consiste en el envío de un mensaje Suscribe por el contenedor interesado hacia el contenedor del Publicador. A su vez, este responde con un mensaje SuscribeACK validando el proceso. En función de si el contenedor monitor ha recibido previamente un Publish de la información o no, el proceso de suscripción constará de una o dos fases: fase de descubrimiento y fase de suscripción.

Fase de suscripción:

En esta fase el contenedor ya dispone de la información necesaria para suscribir la información. Después de la recepción de un Publish se almacena el nombre y dirección de control del servicio que suministra los datos.
Como se observa en la siguiente figura (**Fig. 1.14**), el flujo de información en la suscripción es muy similar a la que se realiza en un mensaje Publish. En primer lugar el monitor llama la función RegisterService(service) del ServiceContainer. En este punto es donde el contenedor consulta sus registros para conocer la información de control del Servicio que se desea suscribir. Se pueden dar tres casos claramente diferenciados:

a. No se dispone de la información solicitada. Se deberá iniciar la secuencia de descubrimiento.
b. Se dispone de la información solicitada pero el servicio se ejecuta en el contenedor local. La petición se conmuta hacia el thread que Correspondiente.
c. Se dispone de la información solicitada y se trata de un servicio remoto. En este caso se continúa con la secuencia de suscripción.

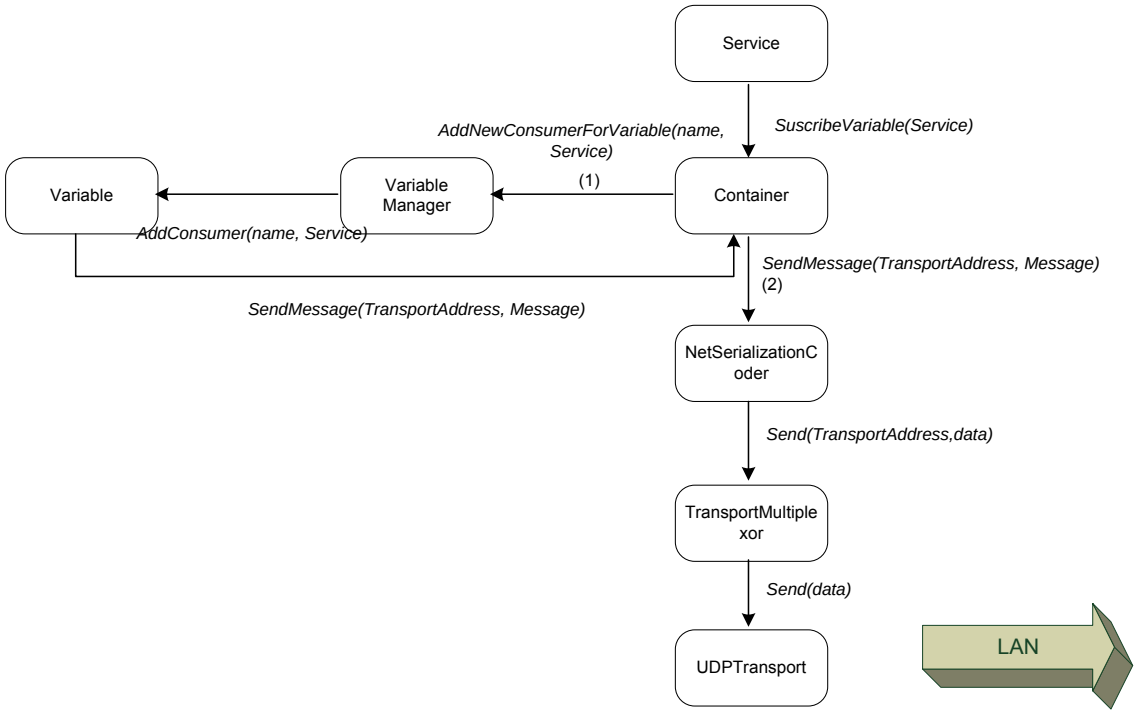


Fig. 1.5 Envío de un paquete Suscribe

Una vez se confirma que la variable/evento a la cual se quiere suscribir el monitor es remota, se procede al envío del mensaje Suscribe a la dirección de control del Servicio publicador. El mensaje Suscribe incluye una lista de todas las direcciones de transporte mediante las cuales el Suscriptor puede recibir la información. Es el servicio publicador quien decide cuál de ellas utilizar.

Como en el resto de mensajes, el Suscribe se envía por la capa de Protocolo a la de Serialización mediante la función SendMessage(Message), y ésta a su vez a la de Transporte mediante la función Send(TransportAddress, data). A diferencia del resto de mensajes, los mensajes Suscribe y SuscribeACK se envían mediante el protocolo de transporte TCP debido a que van destinados a un único receptor. Además, de este modo se garantiza su correcta recepción y procesado.

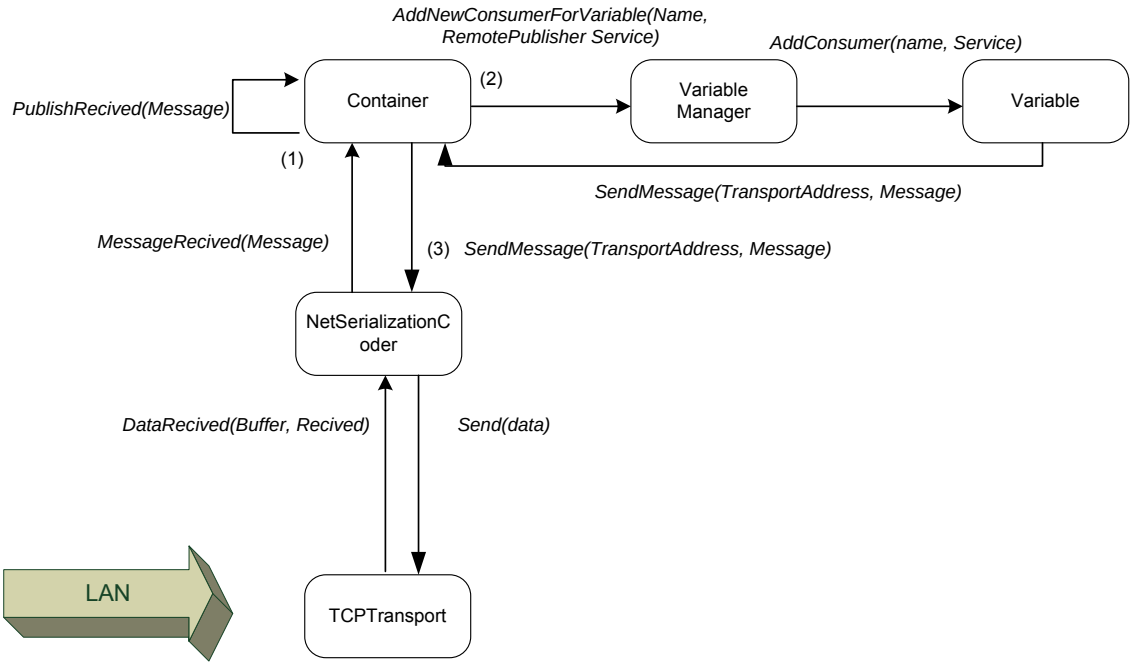


Fig. 1.6 Esquema de recepción de un Suscribe

El flujo de datos al recepcionar un mensaje Suscribe es idéntico al resto de mensajes (Fig. 1.6). Los datos pasan de la capa de Transporte a la de Serialización y de ésta a la de Protocolo. En esta última capa se añade el Suscriptor al registro local del contenedor y se decide la dirección de transporte que se utilizan para el envío de los datos. Esta elección puede depender de diversos factores: tipo de datos, número de suscriptores, criticidad de la información, etc. Acto seguido se envía un mensaje SuscribeACK a la dirección de control del suscriptor indicando la dirección de datos escogida para el envío.

En la figura mostrada más arriba (**Fig. 1.13**) se observa el proceso simplificado de suscripción. Cuando el Contenedor 1 (C1) envía el Suscribe al Productor 1 (P1) éste añade los datos en su registro local y se crea un Consumidor Remoto (C'). En el caso que el Contenedor 2 (C2) desee suscribirse a la misma información, no se crea un segundo Contenedor Remoto si no que se añade la información de C2 en el C'.

El mensaje SuscribeACK finaliza el proceso de suscripción y desencadena la apertura de un puerto de escucha destinado a la recepción de los mensajes de datos.
Fase de descubrimiento:

En caso que no se tenga constancia de ningún publicador con la información que se desea suscribir (no se ha recibido Publish alguno), previamente a la fase de suscripción se realiza una fase de desabrimiento. En esta fase el Monitor intenta descubrir algún publicador que le suministre la información que

desea. Para ello envía un mensaje de Broadcast que inunda toda la red. Este mensaje incluye el descriptor y tipo de información que se desea.

Cuando se recibe un mensaje de Discover los contenedores comprueban si disponen de algún Servicio publicador que encaje con la suscripción. En caso afirmativo vuelven a enviar un mensaje de Publish a toda la red. En cualquier otro caso ignoran el mensaje. Una vez recibido el Publish, se puede iniciar el proceso de suscripción explicado anteriormente.

1.5.4. Ejemplo comunicación

Con el siguiente ejemplo se pretende resumir todo el proceso de publicado y suscripción de información. Para ello se expone el caso más sencillo, un sistema formado por un único Publicador y un Monitor.

En la siguiente figura (**Fig. 1.17**) se observa todo el proceso detallado en los puntos anteriores, desde la publicación de la información hasta la recepción de los datos:

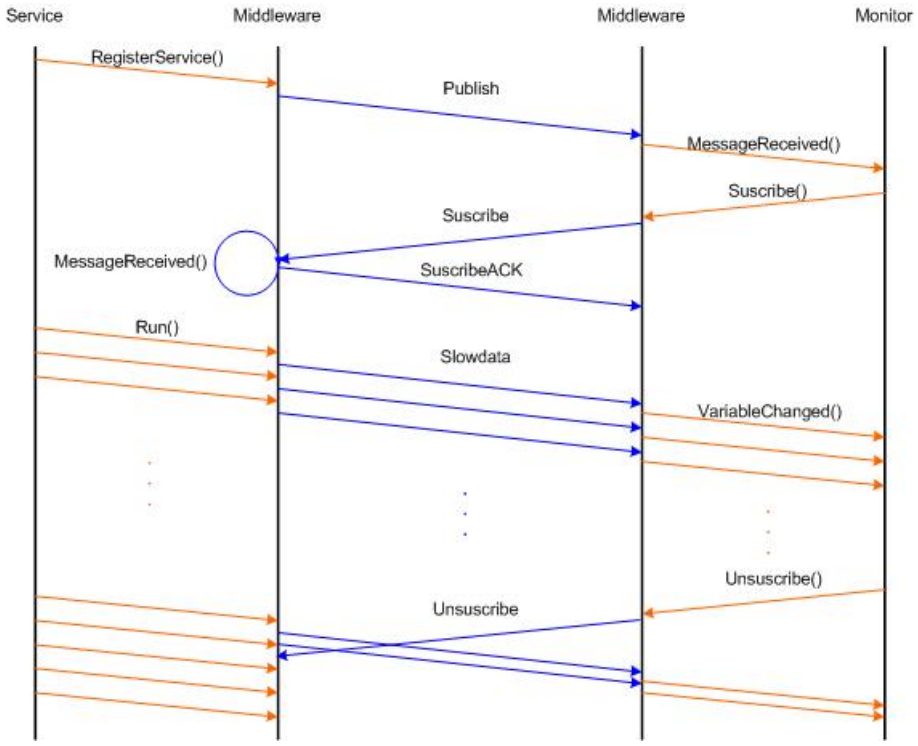


Fig. 1.7 Esquema de comunicación, caso 1

El servicio efectúa un RegisterService() al middleware y éste crea el mensaje Publish hacia otro dispositivo de la red. El middleware que recibe el Publish añade el servicio a una lista. Luego el Monitor llama a Suscribe() y su middleware crea el mensaje Suscribe y lo envía por la red hacia el middleware del servicio al cual se ha suscrito.

El middleware añade el monitor a la lista de suscritos y una vez el servicio envíadatos periódicos, el middleware los envía en forma de Data o SlowData. El middleware que los recibe llama a la función VariableChanged() del monitor y le suministra los datos.

Por último el monitor decide no recibir más datos del servicio con lo que avisa a su middleware y éste manda un UnSuscribe, el middleware que lo recibe borra de la lista el subscritor y por lo tanto si no tiene a nadie en la lista no envía más Data o SlowData.

En el siguiente ejemplo (**Fig. 1.8**) se ilustra una característica del protocolo de MAREA denominada "Desconexión Temporal". El proceso de Publicación y Suscripción son independientes. Si el monitor desea suscribirse a un servicio que no sabe si está activo o no, el middleware hace un Discover para saber si hay algún servicio asociado.

Si lo hay recibe un Publish, sino tiene que esperar a que el servicio se active. Una vez recibe el Publish el middleware envía inmediatamente un Suscribe, con lo que los dos middleware ya tienen en las listas de publicadores y subscritores a servicio y monitor respectivamente.

A partir de este momento el proceso es el mismo que en el ejemplo anterior, se envian los datos de manera periódica a través de Data o SlowData, pero en lugar de cortar el flujo el monitor ahora lo corta el publicador, diciendo que no va a suministrar más datos vía UnRegisterService() y el middleware crea el mensaje UnPublish para avisar al middleware del monitor que lo borre de su lista.

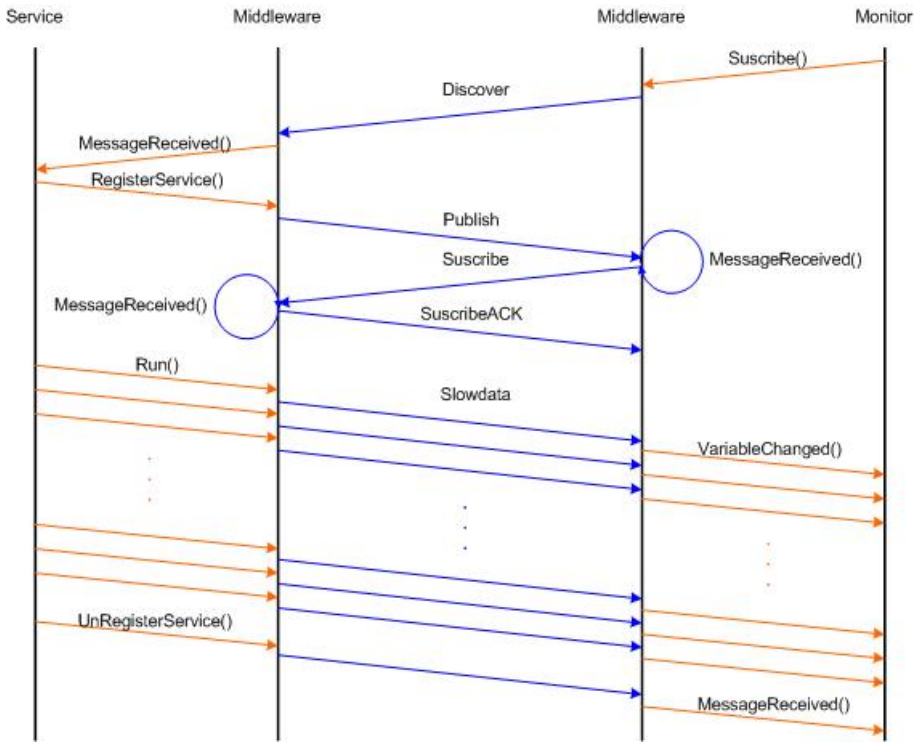


Fig. 1.8 Esquema de comunicación, caso 2

CAPÍTULO 2. DISEÑO

2.1. Introducción

El siguiente capítulo formaliza el proceso de diseño del ICGUAS. La definición formal de Gateway, o puerta de enlace, es: un sistema informático que permite interconectar redes con protocolos y arquitecturas diferentes. Su principal objetivo es la traducción del protocolo de la red de origen al protocolo de la red de destino, y viceversa, permitiendo así la comunicación entre ambas. El Gateway diseñado cumple esta definición además de aportar otras características interesantes para su utilización a bordo de un UAV.

Siguiendo el proceso lógico de diseño se ha estructurado este capítulo en seis apartados diferentes donde se detallaran los aspectos más importantes del diseño del módulo Gateway diseñado para MAREA. El primer apartado describe el escenario de partida a partir del cual se ha diseñado el ICGUAS. El segundo apartado hace referencia a las funciones propuestas que debe implementar el Gateway como punto de partida del proyecto. El siguiente apartado describe los diferentes modos de operación en los que trabajará el software diseñado y los dos últimos se centran en aspectos más técnicos referentes al paso de mensajes y redireccionamiento dentro de la estructura de MAREA.

2.2. Escenario

El escenario principal del proyecto se basa en una aeronave no tripulada la cual transporta diferentes módulos (incluido el ICGUAS) y una estación base de control (véase **Fig. 0.1**). El módulo de comunicación dispone de un enlace 802.11a con un alcance de 50Km y un enlace radio frecuencia de respaldo con un alcance superior a los 30Km.

El escenario más sencillo formado por un UAV y su estación base (**Fig. 0.1**), es el formado por un Servicio que publique información a bordo del UAV y un monitor que Suscriba dicha información en la estación base. En este caso entrará en juego el Gateway de comunicaciones diseñado que, como ya se ha avanzado, será el encargado de decidir por que enlace de los disponibles se enviarán los datos.

2.3. Funciones propuestas

En este apartado se desglosan las principales funciones que el módulo Gateway diseñado debe ser capaz de realizar. Algunas de ellas parten de la

definición inicial del proyecto y otras han surgido a lo largo de su diseño y desarrollo como mejoras a la propuesta inicial.

2.3.1. Funciones principales

Las funciones principales que debe ejercer el ICGUAS son las básicas de debe cumplir cualquier Gateway: permitir interconectar redes distintas entre si. En el caso de este proyecto, debe efectuar la comunicación entre el medio cableado del UAV y el medio cableado de la estación base a través de enlaces inalámbricos, ya sea mediante un MODEM radio frecuencia o mediante un enlace 802.11a.

Funciones principales:

a) Función Gateway radio frecuencia
b) Función Gateway 802.11a

2.3.2. Funciones secundarias

Debido a la naturaleza inalámbrica de las comunicaciones entre los Gateways, los niveles de ruido y que los errores de canal son elevados, es importante controlar los errores que pueda haber y garantizar que todos los mensajes llegan correctamente a su destino.

En el caso de la utilización de enlaces 802.11a, se utilizará TCP como protocolo de transporte. Este protocolo ya incluye funciones de control de errores y retransmisión de paquetes, así como control de flujo y congestión. Por lo tanto se delegaran estas funciones en el protocolo de transporte.
En el caso de los enlaces radio frecuencia, se deben contemplar todos estos aspectos. Se ha creído conveniente implementar un protocolo que nos proporcionara control de errores y de flujo para la transmisión vía radio MODEM. Este nuevo protocolo se ha denominado SerialProtocol y se explica a lo largo de este documento.

Por otro lado, el módulo Gateway debe ser capaz de controlar los enlaces disponibles (ancho de banda, control de errores, potencia, etc) y escoger el más adecuado para el envió de cada mensaje. En general esta elección se realizará en función del ancho de banda disponible, pero se puede dar el caso que algún tipo de mensaje sea enviado por un enlace más lento pero más seguro para garantizar su correcta recepción.

Por último, el Gateway ha de ser capaz de comunicarse con varios Gateways a la vez, distinguiéndolos y sabiendo en todo momento a cual se ha de dirigir para hacer más efectiva la comunicación, tanto en enlaces radio frecuencia como en 802.11a.

Funciones secundarias:

c) Funciones de control de errores en transmisión inalámbrica
d) Maximización de la eficiencia en la transmisión radio frecuencia
e) Funciones de control de los enlaces disponibles
f) Elección del mejor enlace
g) Soporte para interoperabilidad con múltiples Gateways

2.3.3. Otras características

Una característica importante en el diseño del Gateway es que sea modular e independiente. Es decir, que pueda agregarse al resto de código como un módulo independiente del resto, alterando lo más mínimo la programación existente.

Por otro lado, el diseño que se ha realizado debe funcionar independientemente de los servicios o aplicaciones que funcionen en capas superiores, dando así servicio a todos los futuros servicios además de los ya programados. Para ello, las entradas y salidas del sistema de comunicación deben seguir el estándar marcado por MAREA.

La última de las características adicionales del diseño, es que se puede configurar mediante un archivo de configuración en formato XML, de una manera externa e independiente del código, los parámetros de funcionamiento del Gateway. Esta característica permite una mejor gestión de los recursos y una manejabilidad que no se podría conseguir realizando la configuración mediante el código software.

Gracias a estas características se asegura que la introducción del módulo Gateway en la plataforma MAREA no interfiera en el trabajo previo ya realizado y se pueda utilizar independientemente de los servicios que se quieran integrar en el avión.

Otras funciones:

h) Programación modular
i) Integración con el protocolo de MAREA
j) Configuración mediante fichero XML

2.4. Modos de operación

Como ya se ha apuntado en la introducción de este capítulo, la principal función de un Gateway es la de realizar una traducción entre protocolos de redes distintas para permitir la comunicación final entre ambas.

Como se ilustra en la **Fig. 0.1**, el escenario del proyecto está formado por un avión no tripulado y una estación de control en tierra. Estos dos elementos

están separados una distancia considerable y variable, que puede ir desde los pocos metros a muchos kilómetros de distancia. Por ello, y para garantizar la total autonomía de la aeronave, se opta por la utilización de medios inalámbricos para la comunicación entre ambos.

Tanto en tierra como a bordo de la aeronave, el medio de comunicación utilizado es un bus de comunicaciones mediante protocolo IEEE 802.3 (Ethernet). En cambio, entre el avión y tierra se utilizan protocolos de comunicación inalámbricos, bien radiofrecuencia o bien 802.11a. Así pues, para permitir la comunicación, el sistema debe enviar los datos desde un medio cableado a otro a través del medio aéreo, para ello se realiza una doble traducción de protocolos mediante dos módulos Gateway. La siguiente figura (**Fig. 2.1**) ejemplariza el escenario descrito:

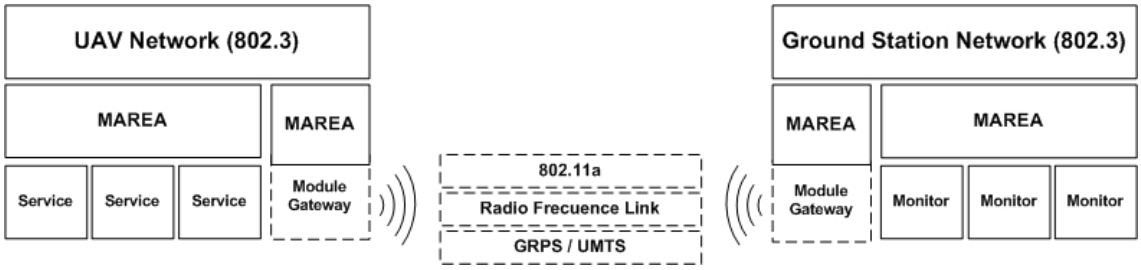


Fig. 2.1 Esquema Proyecto con módulo Gateway

Como se ve en la figura anterior (**Fig. 2.**), el sistema consta de dos módulos Gateway simétricos que establecen un canal de comunicación para enlazar ambos extremos. Así pues, el funcionamiento general de los módulos es el siguiente:

En emisión:

1. Se recibe una trama por la interfaz local Ethernet
2. Se extraen los datos de la trama Ethernet
3. Se encapsulan los datos en una nueva trama dependiendo del protocolo que se utilice para la transmisión (Radio frecuencia o 802.11a).
4. Se envía la nueva trama a través de la interfaz inalámbrica.

En recepción:

5. Se recibe una trama por la interfaz inalámbrica
6. Se extraen los datos de la trama utilizada
7. Se encapsulan los datos en una nueva trama Ethernet
8. Se envía la trama Ethernet por la interfaz local

Como se puede observar, la secuencia de emisión y recepción son simétricas. Esto facilita el trabajo de implementación además de permitir a un mismo módulo realizar ambas funciones simultáneamente. En la siguiente figura (**Fig.** 2) se puede observar el camino realizado por los datos desde que son enviados hasta que se reciben en el otro extremo y viceversa:

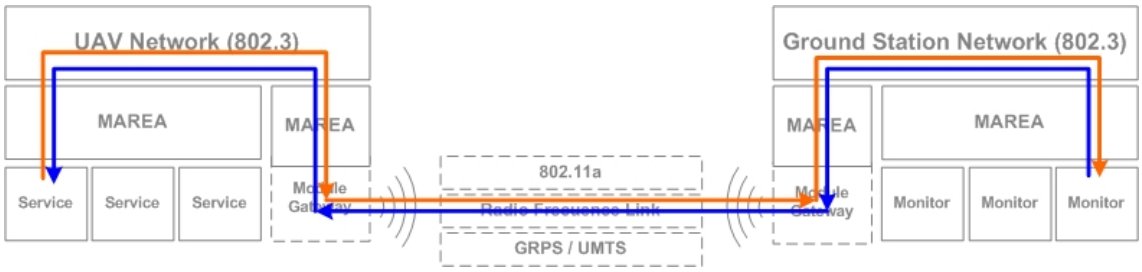


Fig. 2.2 Camino de los datos

Como ya se ha comentado anteriormente, el Gateway diseñado, además de cumplir con las funciones básicas enunciadas, cumple una serie de requisitos que facilita su integración dentro del MAREA: transparencia, modularidad, independencia y configuración externa. Estos requisitos se han impuesto des del inicio del proyecto de cara a permitir una interoperabilidad entre los servicios ya diseñados, el sistema de comunicación del MAREA y el nuevo sistema de comunicación.

Un punto importante para que el funcionamiento del Gateway, no interfiera en el funcionamiento normal del sistema, es que estos resulten “transparentes” para el resto de servicios. Es decir, que estos no perciban de modo alguno la existencia del Gateway.
Así pues, el funcionamiento de un servicio no debe verse afectado por la introducción del Gateway en el sistema. Servicios que no se encuentren físicamente en la misma red, podrán comunicarse entre si sin necesidad de conocer la ubicación real del interlocutor. De cara a los servicios, siempre se estarán realizando comunicaciones en un ámbito de red local cuando en realidad el Gateway estará haciendo de enlace entre redes a través de un enlace radio o 802.11a. De este modo, se podrá interactuar con servicios que no se encuentren físicamente en el avión (o en tierra) como si estuvieran. De alguna manera, puede decirse que gracias al sistema formado por el Gateway se puede simular la existencia de servicios remotos en la red local.
En el siguiente apartado se describen los diferentes modos o escenarios para el que se ha diseñado el módulo Gateway. El proyecto descrito en este documento funciona en todos estos escenarios y en sus variantes.

2.4.1. Modo Gateway – Gateway

Este es el modo de funcionamiento básico. Se compone de un sistema con dos únicos Gateways en equipos que no ejecutan ningún servicio o monitor. Es decir, se utiliza un módulo adicional en el sistema para ejecutar las tareas de Gateway. Debido a que el contenedor no dispone de ningún Servicio o Monitor los mensajes recibidos no son procesados a nivel de MAREA, el Gateway los reencamina hacia su destino. (**Fig. 2.**)

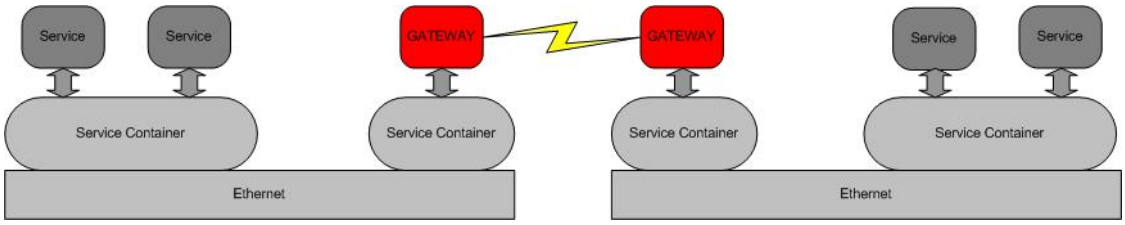


Fig. 2.3 Modo Gateway - Gateway

2.4.2. Gateway – N-Gateway

Este caso es idéntico al anterior, pero cada Gateway ha de tener la suficiente inteligencia para encaminar los paquetes hacia un Gateway determinado, de manera eficiente y evitando bucles de mensajes en la red. (**Fig. 2.**)

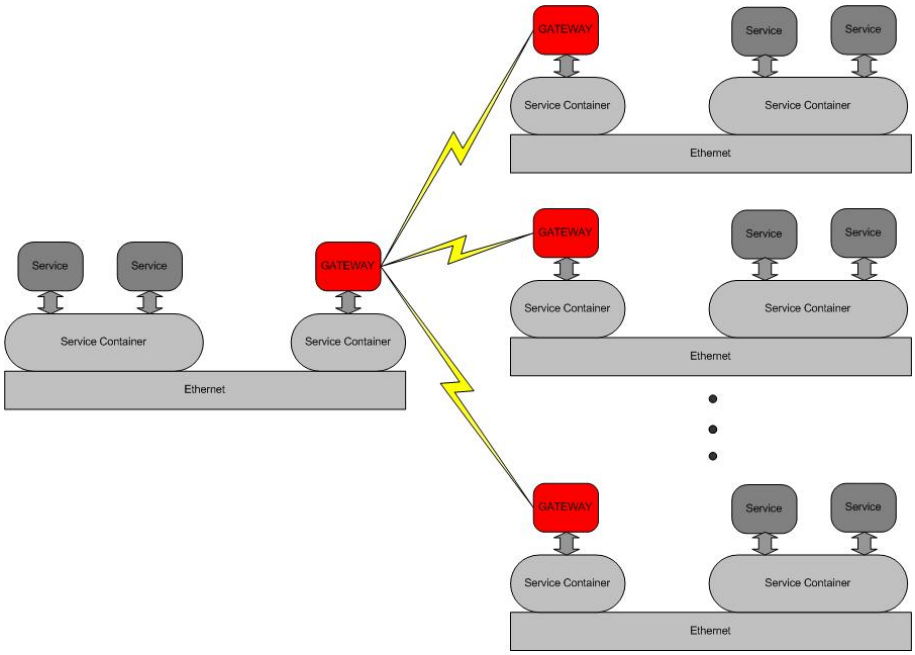


Fig. 2.4 Modo Gateway – N-Gateways

2.4.3. Gateway+Servicio – N- Gateway+Servicio

Este es el caso más complejo. El Terminal además de disponer de la función de Gateway, tiene uno o varios servicios y monitores por encima que solicitan y dan la información necesaria para otros terminales. En este caso, el módulo Gateway ha de discernir entre los mensajes que debe procesare a nivel de MAREA y los mensajes que ha de encaminar.(**Fig. 2.**)

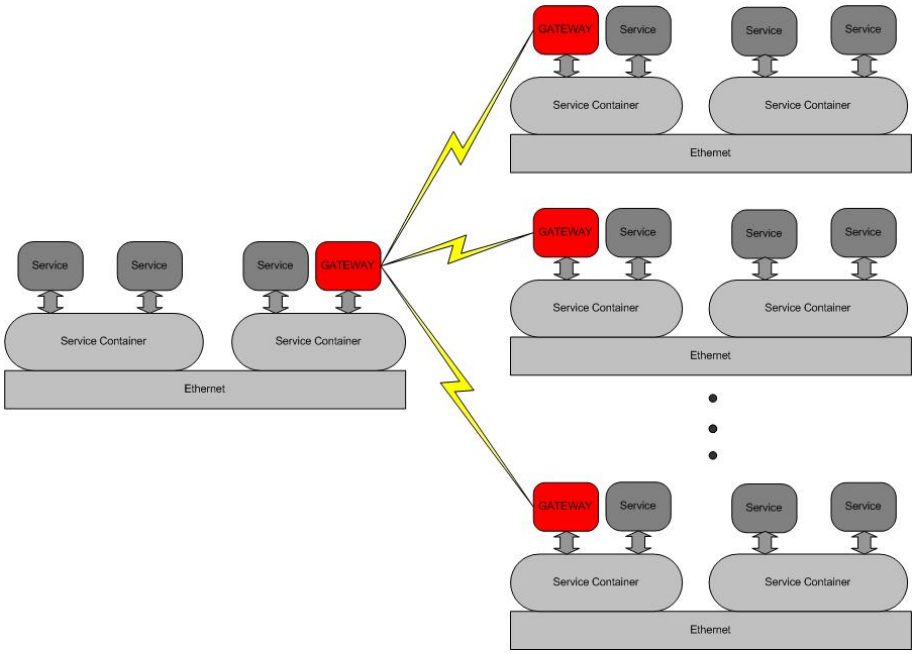


Fig. 2.5 Modo Gateway + Servicio – N-Gateway + Servicio

Todos estos modos de operación deben ir acompañados de la configuración adecuada del documento XML de cada Gateway. Estas posibles configuraciones no son excluyentes y pueden funcionar conjuntamente incrementando las posibilidades de funcionamiento del sistema.

2.5. Carga de la configuración

Esta es una parte muy importante, ya que al igual que el middleware necesita una configuración para el correcto arranque de servicios, en este caso se necesita una configuración para la parte de interfaces y Gateways de destino.

El motivo por el cual se realiza la configuración en un fichero .xml y no dentro del propio código del Gateway es para diferencial la capa de programación de la de configuración a nivel de usuario final. Se elige un fichero XML para asemejarlo lo más posible a la parte del middleware. El fichero consta de varios campos obligatorios y algunos opcionales.

Los obligatorios son:

- La dirección de Broadcast utilizada para la red local del avión o control de tierra.
- La información relativa a un Gateway remoto.

Como campos opcionales se tienen la información relativa a otros Gateways para el caso de que existan más de 1.

La información relativa a los Gateways se compone de nombre de Gateway, tipo de interfaz (serial o ip), dirección local de la interfaz por la que sale el

paquete y dirección de destino que es la dirección del Gateway al cual hace referencia.

2.6. Mensajes y servicios virtuales

Para el correcto funcionamiento del envío de las variables/eventos entran en escena los servicios virtuales descritos en el **Capitulo 1 – Estudio Preeliminar**.
Además esta explicación está realizada según el escenario descrito anteriormente.

Cuando el Gateway del Avión recibe el Publisher de la variable GPS, realiza su proceso y debido a que el servicio se encuentra en otra máquina añade a la lista de variables un publicador remoto de la variable GPS.

Una vez lo procesa, se encapsula y se envía hacia los demás Gateways. En este caso al haber únicamente uno más, se envía al Gateway del centro de control cambiando la dirección de control previa por la suya.

El Gateway recibe el mensaje, lo desencapsula y lo procesa, añadiendo otro publicador remoto para la variable GPS. Finalmente, envía ese mismo Publish a los demás dispositivos de su red local.

En el diagrama mostrado en la siguiente figura (**Fig. 2.**) se describe el flujo de mensajes respecto al tiempo, y como el Gateway encapsula o no estos mensajes y los dirige hacia el otro extremo.

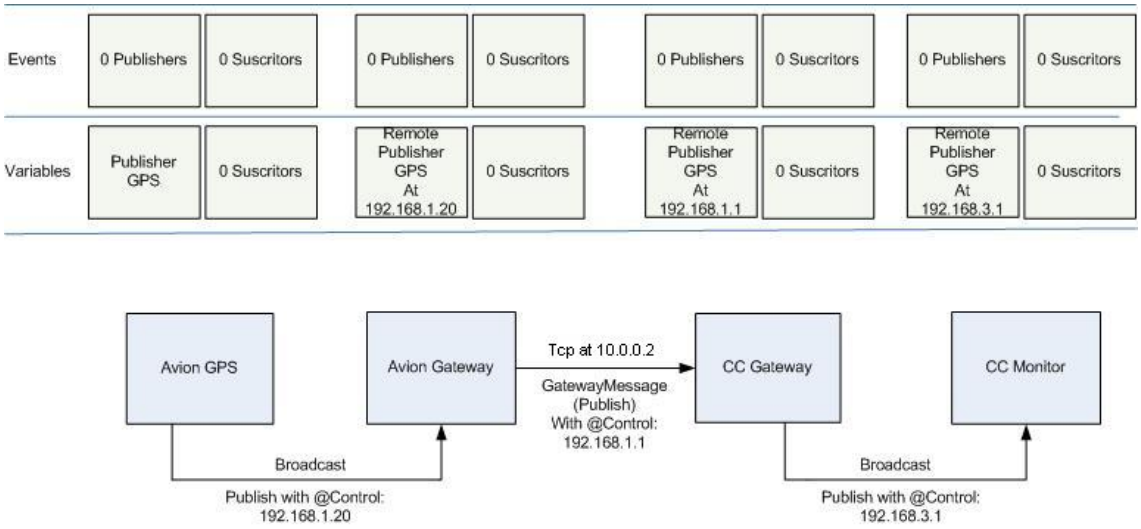


Fig. 2.6 Mensajes y Servicios Virtuales I

A la hora de suscribirse, el monitor envía el mensaje Suscribe a la dirección de control anotada en la lista de variables, esa dirección corresponde a la del Gateway del centro de control, además le pasa la dirección de datos donde ha

de responder el publicador y genera un Suscritor en la tabla de suscritores para la variable GPS.

El Gateway la recibe y el procesado interno del middleware ve que se trata de una suscripción a un servicio que el tiene anotada que está en otra máquina remota, por lo que automáticamente crea otro mensaje de Suscribe con su dirección de control y datos (que entre Gateways va a ser la misma) y lo envía hacia donde tiene anotada la dirección del publicador que será la del Gateway del avión. Además añade a la lista de suscritores un suscritor remoto con una dirección de control y de datos especificados en el mensaje recibido.

El Gateway del avión recibe el Suscribe y realiza el mismo proceso que el otro Gateway. Finalmente el terminal lo recibe, lo procesa y añade el suscritor a la lista de suscritores, con la dirección de control del Gateway.

Cabe mencionar que cuando se recibe un Suscribe se genera el SuscribeACK que comunica la dirección de datos se tomará para el envío de inforación.

La figura (**Fig. 2.**) ilustra como es el resultado del proceso:

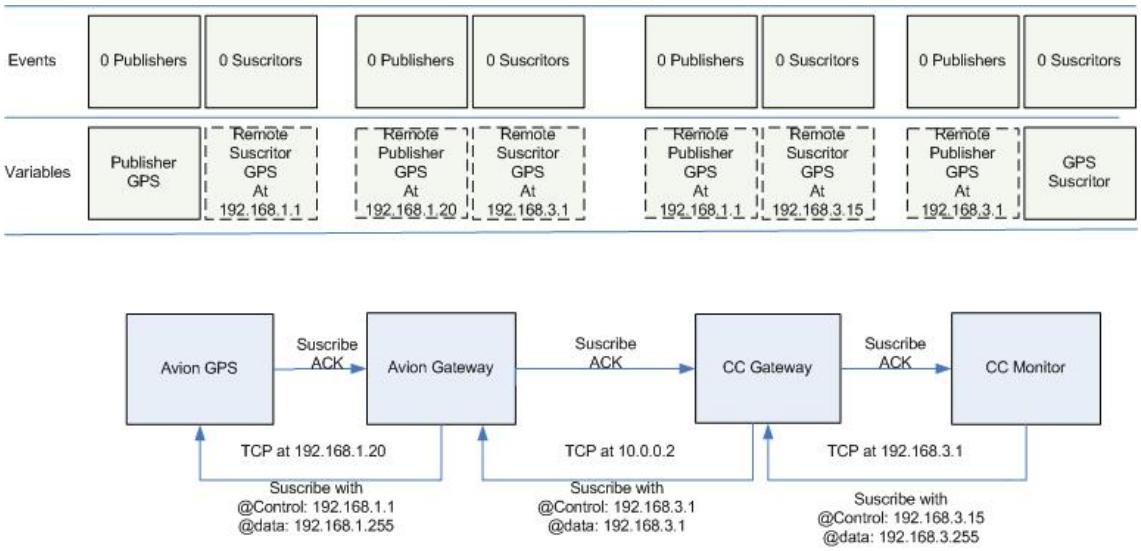


Fig. 2.7 Mensajes y Servicios Virtuales II

Finalmente se suministran los datos de terminal a terminal hacia la dirección de datos anotada en la lista de variables y el resultado es el que se muestra en la figura (**Fig. 2.**):

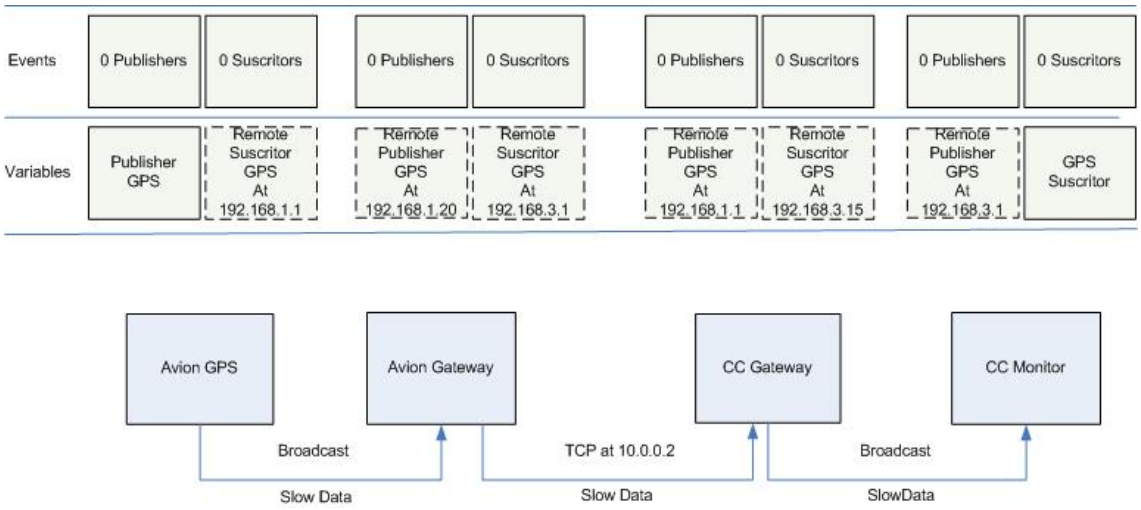


Fig. 2.8 Mensajes y Servicios Virtuales III

A la hora de la publicación de eventos, el sistema es exactamente el mismo. Únicamente varía que los publicadores y suscritores se añaden a la lista de eventos y no a la de variables. El resultado final de todo el proceso es el de la siguiente figura (**Fig. 2.**).

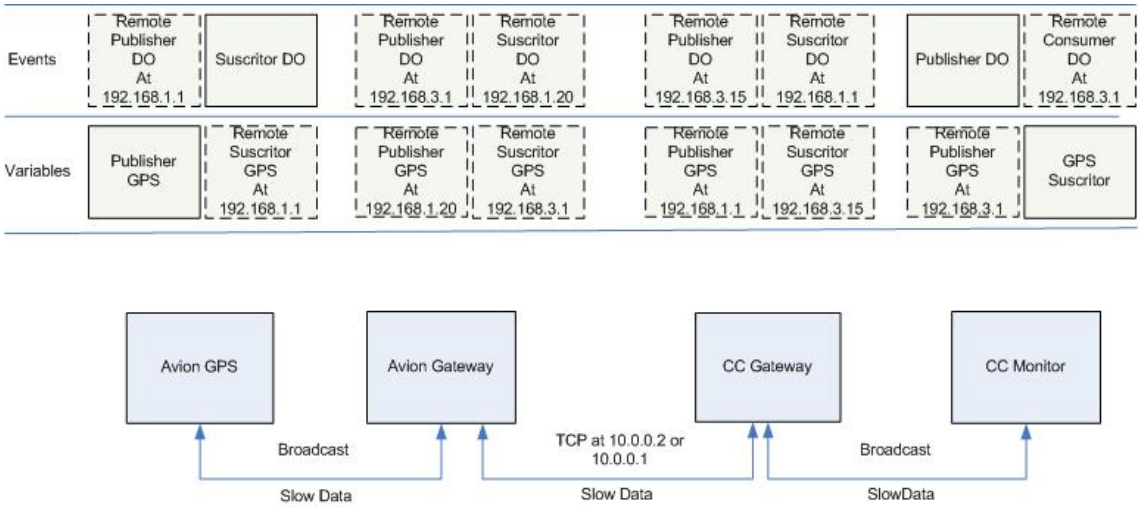


Fig. 2.9 Mensajes y Servicios Virtuales Final

Si el avión se aleja de la distancia disponible para 802.11 no nos afecta en los servicios virtuales, tan solo afecta al modo de transmisión entre Gateways, usando el protocolo Serie descrito en el capítulo anterior.

2.7. Redireccionamiento de mensajes

Para que los Gateways puedan mantener una correcta transmisión de datos sin alterar la parte de comunicaciones de la red local, es necesario diseñar un

protocolo que capacite el Gateway para decidir donde ha de enviar el mensaje, si ha de modificar algún campo, etc.

Para el correcto redireccionamiento entra en juego el fichero de configuración antes mencionado. La configuración cargada se utiliza en una tabla que se emplea únicamente para extraer la dirección de destino del paquete y la interfaz por donde ha de salir.

Dado que la transmisión sin cable tiene ciertas limitaciones en cuanto a velocidad y distancia se refiere, se necesita de un algoritmo que extraiga el estado de las interfaces y ayude a decidir por que interfaz se trasmite cada mensaje.

Para realizar este proceso, además de la tabla de configuración del Gateway se crea otra donde aparece el estado de cada interfaz reconocida por el terminal. Esta información se extrae gracias a la librería NetworkInformation para el caso de interfaces IP, para el caso de interfaces serie, se tiene que ver si el dispositivo tiene algún control de conexión, de velocidad, etc.

Cada vez que se quiera enviar un mensaje hacia otra red, se actualiza la tabla y se escoge la interfaz que ha resultado tener mayor velocidad. Todo ello se realiza en la capa de transporte justo antes de que el multiplexor elija el método de transmisión, ya que previamente se necesita la extracción de la dirección de transporte de destino que resulta del proceso antes descrito.

CAPÍTULO 3. IMPLEMENTACIÓN

3.1. Introducción

En el siguiente apartado se formaliza el proceso de implementación del Gateway, detallando los aspectos más importantes en cuanto a la programación y el diseño de algoritmo y funciones.

Para permitir la comunicación se dispone de un equipo de radio frecuencia "24XStream™ (2.4 GHz)", con un alcance de hasta 16 km a una velocidad de 9600 bps y un módulo de comunicación 802.11a Ubiquity SR5 con un alcance de hasta de 50 Km a 9600bps. El sistema diseñado es capaz de realizar la comunicación mediante ambos equipos utilizando cualquiera de los dos componentes. Se contempla, también, la utilización de medio cableado para la comunicación entre avión y estación de control. Este medio puede ser de especial interés para recopilar datos una vez el avión ha aterrizado ya que se reducen las pedidas ganando velocidad y ancho de banda de transmisión.

Para facilitar la comprensión del proceso seguido se ha estructurado el apartado siguiendo los pasos lógicos del módulo de Gateway, desde que se pone en funcionamiento hasta que inicia el envío de tramas, con toda la casuística que esto conlleva.

3.2. Arquitectura de clases

A continuación detallaremos la estructura de datos usada en la implementación del módulo Gateway.

3.2.1. ServiceContainer()

Como ya se ha mencionado anteriormente, uno de los objetivos del proyecto es adaptar el Gateway a la programación ya existente. Para ello, dentro de la clase ServiceContainer(), explicada en el *Capítulo 1. Estudio preliminar*, se añade una variable del tipo booleano llamada imGW que nos indica si el sistema debe realizar o no las funciones de Gateway.

La variable imGW, inicializada originariamente con valor *false*, cambia en función de los parámetros del fichero de configuración de MAREA startup.xml. Para ello, se ha modificado la función LoadConfiguratio() de la clase ServiceManager para detectar si el sistema debe realizar las funciones de Gateway, y, en dicho caso, realizar la carga de datos que más adelante explicaremos.

Por otro lado, se ha creado la clase GatewayContainer homónima al ServiceContainer. Esta nueva clase se encarga de gestionar el flujo de información enviada a través del Gateway así como la carga de la información de configuración al inicio del programa. Esta clase se inicializará desde el ServiceContainer en función del valor de la variable imGW. Para ello se ha modificado la función Initialize() para que, en función del valor de la variable imGW, se inicialice un objeto del tipo GatewayContainer así como otros objetos que más adelante explicaremos.

```
if (imGW)
{
      Gateway = new GatewayContainer(this);
}
```

También se han incorporado unas opciones en el menú para mostrar:

- La tabla TransportList que recoge la lista de direcciones de transporte y prioridad que utilizan los Gateways entre sí para la transmisión de la información. (**ANEXO 7: Fig. 4**). El código que imprime la tabla se encuentra en el CD anexo (**ANEXO 11**) a este documento.

- Una opción para ver o esconder la información que contiene cada mensaje procesado por el Gateway. Cada vez que llega un mensaje al módulo Gateway nos imprime toda la información que contienen los mensajes: dirección de control, tipo de mensaje, si es una variable o evento y sus direcciones de datos y control si la contienen (**ANEXO 7: Fig 14**). El código que realiza esta función se encuentra en el CD anexo (**ANEXO 11**) a este documento.

3.2.2. GatewayContainer()

Como ya se ha mencionado, la clase GatewayContainer es la encargada de gestionar el flujo de información a través del Gateway además de realizar la carga inicial de la configuración del mismo. Para ello, es necesario disponer de una estructura de datos donde almacenar la información de todas las interfaces existentes en el sistema que permita realizar el envío de datos.

En general (depende de la configuración del UAV), habrá tres interfaces básicas en el sistema diseñado: interfaz local (ethernet), interfaz wifi (802.11a) e interfaz radio (puerto serie). La configuración de cada una de estas interfaces se realiza manualmente en el fichero Gateway_Configuration.xml por el administrador del sistema. Los datos de este fichero se cargaran al inicializar el objeto de tipo GatewayContainer durante el proceso de carga de datos que se verá más adelante. Para almacenar toda esta información se declara una estructura del tipo Dictionary llamada TransportList. En dicha estructura se almacenan los datos asociados a cada Gateway remoto y a su conjunto de enlaces, más adelante se verá en detalle dicha tabla.

```
public Dictionary<String, RemoteGateway> TransportList = new
Dictionary<string, RemoteGateway>();
```

También se define un TransportAddress broadcast para el caso de envío de mensaje hacia la LAN en modo broadcast

```
private TransportAddress broadcast;
```

La clase GatewayContainer contiene las funciones encargadas de la carga de la información de configuración y de la gestión de mensajes recibidos. Estas funciones son: LoadInterfazs() y MessageReceived(Message Message). Más adelante se explicaran en detalle ambas funciones.

3.2.3. Selector()

Para poder realizar un redireccionamiento entre varios Gateways, se añade una subcapa que trabaja entre la capa de serialización y de transporte, de manera que cuando el mensaje es tratado por un Gateway la capa de serialización le pasa el mensaje con su dirección y éste determina por que interfaz ha de salir y hacia que Gateway ha de ser dirigido.

Esta capa es la que aporta operatividad al Gateway y dispone de un mecanismo para detectar enlaces caídos y el ancho de banda de cada enlace, permitiendo optimizar el uso de cada conexión.

Para hacerlo se guía por la tabla TransportList creada en el GatewayContainer. Más adelante se explica cómo se realiza el algoritmo.

3.2.4. SerialTransport()

La placa radio frecuencia 24XStream™ (2.4 GHz) de la que se dispone para la transmisión de datos, posee una entrada del tipo serial port (puerto COM) por la que recibe el flujo de información a transmitir mediante un puerto serie. De la misma manera que las clases UDPTransport y TCPTransport controlan la comunicación mediante los protocolos UDP y TCP respectivamente, se define la clase SerialTransport que controla la comunicación mediante el puerto serie.

La clase SerialTransport cumple, también, con lo establecido por la interfaz ITransport y una vez inicializada se puede agregar a la clase TransportMultiplexor del MAREA con el fin de poder ser utilizada en el envío y recepción de mensajes. A continuación se puede ver un fragmento de código donde se inicializa el GatewayContainer y el SerialTransport:

```
if (imGW)
{
      Gateway = new GatewayContainer(this);
      serial = new SerialTransport(new
      TransportAddress(Gateway.TransportList["serial"].serialport));
      transport = new TransportMultiplexor(udp, tcp, serial);
 }
else
{
```

```
        transport = new TransportMultiplexor(udp, tcp);
}
```

Cuando se dispone de enlace por 802.11a se pueden utilizar los protocolos UDP o TCP en función del tipo de mensaje que se desee enviar. En caso de utilizar TCP, el protocolo ofrece servicios de control de flujo, confirmación de recepción de la información, reenvío de paquetes, etc. Estos servicios no están disponibles en las clases y funciones incluidas dentro del API de C# para Visual Studio 2005 referentes a la gestión de puertos serie. Para suplir esta carencia se ha diseñado un protocolo que permite emular el comportamiento del protocolo TCP sobre Ethernet que se detallará en el siguiente apartado.

3.2.5. SerialMonitor()

Con el fin de controlar los mensajes pendientes de reconocimiento enviados a través del puerto serie, se ha creado la clase SerialMonitor. Cada enlace por puerto serie tiene un objeto del tipo SerialMonitor vinculado.

Esta clase contiene una estructura del tipo Dictionary llamada PendingPackages donde se almacenan temporalmente los mensajes enviados pendientes de reconocimiento. Estos mensajes se guardan juntamente con su número de secuencia. Como se verá más detalladamente en el siguiente apartado, esta clase se encarga de: actualizar la tabla de mensajes pendientes de confirmación, proporcionar el parámetro type en función del tipo de mensaje, devolver el siguiente valor Sequence válido y reenviar cada cierto tiempo los mensajes de los cuales no se ha recibido la confirmación de recepción. Estas funciones se realizan mediante los métodos ActPendingPackeges(ushort seq, byte[] data), GetType(byte[] m), GetSequence() y SendPending() respectivamente.

3.2.6. Variable()

En esta clase se tratan las variables publicadas por separado, se realiza el envío de Publishes, Suscribes, Discovers, UnPublishes y UnSuscribes.

Es necesario hacer una modificación en algunas funciones para que, además de realizar el tratamiento normal del mensaje, añadirlo a la tabla y enviar o reenviar los mensajes hacia otro destino, se envíen por la interfaz del Gateway además de la interface de la red LAN. De esta manera, se puede trabajar en cualquiera de los modos descritos anteriormente.

Así tenemos que para el caso en que el Terminal sea un Gateway, en las funciones AddPublisher(), RemovePublisher(), AddConsumer(), RemoveConsumer(), DiscoverPublisher() se añaden las siguientes sentencias:

- AddPublisher: Se añaden dos comparaciones, una para el caso del Suscribe para indicar cual es la dirección de datos a utilizar:

```
if (container.ImGW)
```

```
    tmp.Add(Transport.Control);
else
    tmp.Add(Transport.Broadcast);

container.SendMessage(((RemotePublisher)s).control, new
Subscribe(name, Primitive.Variable, Transport.Control, tmp));
```

Y otra para que el Publish se envíe tanto para la red local como para los distintos Gateways de la red:

```
if (container.ImGW)
                {
                    container.SendMessage(Transport.Broadcast, new
Publish(name, Primitive.Variable, Transport.Control));
                    container.Gateway.SendRemoteMessage(new
GatewayMessage(new Publish(name, Primitive.Variable,
Transport.Control)));
                }
                else container.SendMessage(Transport.Broadcast, new
Publish(name, Primitive.Variable, Transport.Control));
```

- RemovePublisher: Para el envío de los UnPublish hacia los Gateways y la red local:

```
if (container.ImGW)
                {
                    container.SendMessage(Transport.Broadcast, new
UnPublish(name, Primitive.Variable, Transport.Control));
                    container.Gateway.SendRemoteMessage(new
GatewayMessage(new UnPublish(name, Primitive.Variable,
Transport.Control)));
                }
                else container.SendMessage(Transport.Broadcast, new
UnPublish(name, Primitive.Variable, Transport.Control));
```

- AddConsumer: Es el caso parecido al AddPublisher, primero para el envío del Suscribe se especifica la dirección de datos a tomar:

```
if(!((RemotePublisher)Publishers[0]).remote || (container.ImGW))
                        {
                            tmp.Add(Transport.Control);
                        }
                        else
                        {
                            tmp.Add(Transport.Broadcast);
                        }

container.SendMessage(((RemotePublisher)Publishers[0]).control, new
Subscribe(name, Primitive.Variable, Transport.Control, tmp));
```

Y luego se envía el Discover por la red local y todos los Gateways:

```
if (container.ImGW)
{
   container.Gateway.SendLocalMessage(new Discover(name,
Primitive.Variable));
```

```
container.Gateway.SendRemoteMessage(new GatewayMessage(new
Discover(name, Primitive.Variable)));
}
else container.SendMessage(Transport.Broadcast, new Discover(name,
Primitive.Variable));
```

- RemoveConsumer: Este caso es el más complejo, se añade una comparación para enviar el UnSuscribe y lo que hará es ver si el mensaje se ha de enviar o no a otro de los Gateways, gracias a la variable remote. Si esta variable es falsa significa que el mensaje solo se transmite por la red local, si es *true* entonces se tiene que enviarse al Gateway correcto

```
if (container.ImGW)
{
   bool remote=false;
   foreach (string g in container.Gateway.TransportList.Keys)
   {
    if(
container.Gateway.TransportList[g].control.ipEndPoint.Address.Address
== rp.control.ipEndPoint.Address.Address)
    {
        container.Gateway.MessageReceived(new Unsubscribe(name,
Primitive.Variable, Transport.Control, rp.data));
        remote = true;
     }
   }
   if (remote == false)
   {
      container.SendMessage(rp.control, new Unsubscribe(name,
Primitive.Variable, Transport.Control, rp.data));
   }
}
else
   container.SendMessage(rp.control, new Unsubscribe(name,
Primitive.Variable, Transport.Control, rp.data));
```

- DiscoverPublisher: Finalmente aquí se realiza lo mismo que en el caso del RemotePublisher(), en esta función se envían el Publish si la máquina dispone del servicio localmente, el código añadido queda así:

```
if (container.ImGW)
{
   container.SendMessage(Transport.Broadcast, new Publish(name,
Primitive.Variable, Transport.Control));
   container.Gateway.SendRemoteMessage(new GatewayMessage(new
Publish(name, Primitive.Variable, Transport.Control)));
}
else
   container.SendMessage(Transport.Broadcast, new Publish(name,
Primitive.Variable, Transport.Control));
```

3.2.7. Event()

En esta clase pasa exactamente lo mismo que en la anterior, el problema es el mismo y lo que varía es que cuando se envía un mensaje, este mensaje tiene como primitiva Evento, lo cual hace posteriormente que toda la comunicación se haga mediante TCP y no da opción a utilizar Broadcast, Multicast o UDP en general.

Las funciones modificadas y el código es prácticamente el mismo, en el **ANEXO 11** donde está el código completo del programa se puede observar.

3.2.8. Esquema UML de clases

En el **ANEXO 5** se presenta el gráfico UML de clases del módulo Gateway. Como ya se ha ido explicando a lo largo del apartado, se puede observar como para cada instancia ServiceContainer() tenemos un objeto del tipo GatewayContainer(), este objeto a su vez, define un objeto UDPTransport, un objeto TCPTransport y otro SerialTransport para la transmisión de mensajes a través de sus interfaces. Además, por cada SerialTransport se crea un objeto del tipo SerialMonitor para gestionarlo.

3.3. Carga de la configuración

Como ya se ha comentado en los puntos anteriores, el Gateway diseñado puede configurarse mediante un archivo de configuración externo al código de de la aplicación. Esta característica permite diferenciar claramente la programación de la configuración de los parámetros del módulo Gateway. El archivo de configuración es un documento codificado en texto plano en formato XML.

La carga de configuración se realiza una única vez al inicio del programa principal. Esta carga la realiza la clase ServiceManager mediante la función LoadConfiguration(). Esta función recorre el fichero startup.xml y carga su información en las variables del MAREA. Se ha modificado dicha función para que detecte una línea que indique si se deben realizar las funciones de Gateway. Esto se indica mediante la siguiente línea del fichero statup.xml de MAREA:

```
<Gateway set = "yes" conf_file = "Gateway_Configuration.xml">
```

En caso de encontrar dicha línea en el fichero de arranque del MAREA se procede a cargar la configuración del módulo Gateway desde el archivo indicado en el parámetro conf_file. El aspecto general del archivo de configuración es el siguiente:

```
<local address="192.168.0.255"  port="11811" />

<control name="gw2" localaddress="10.0.0.2" port="11811" />
```

```
   <interface name="gw2" type="ip" localaddress="10.0.0.1"
   remoteaddress="10.0.0.2"  port="11811" />

   <interface name="gw2" type="serial" localaddress="COM3"
   remoteaddress="COM3"      port=""/>

<control name="gw3" localaddress="10.0.0.3" port="11811" />

   <interface name="gw3" type="ip" localaddress="10.0.0.2"
   remoteaddress="10.0.0.3"  port="11811" />

   <interface name="gw3" type="serial" localaddress="COM1"
   remoteaddress="COM1"      port=""/>

<RTO value="3000"/>
```

Como se puede observar, existe una línea de control por cada Gateway Remoto, y luego existen varias interfaces que estarán asociadas a esa dirección de control.

Además en el fichero de configuración se especifica también cual es la dirección Broadcast de la LAN que ocupa el Gateway.

A continuación se enumeran los parámetros configurables y su utilidad:

- Name: es la asociación de una interfaz hacia un Gateway remoto y por lo tanto a su dirección de control. Este nombre debe estar relacionado con el nombre de una dirección de control, con lo cual si el nombre de la interfaz no se corresponde con ninguna de control se ignora la interfaz y sus direcciones como si no existiesen.

- Type: se especifica el tipo de protocolo de transporte por el que viaja los mensajes mediante esa interfaz. Existen dos: IP, nos indica que utilizará la pila de protocolos TCP/IP. Y la Serial que se utiliza el protocolo Serie descrito anteriormente.

- Localaddress: IP o nombre de la interfaz local.

- Remoteaddress: IP o nombre de un Gateway remoto que es utilizado como EndPoint.

- Port: Número de puerto remoto hacia donde viaja la información. En el caso de las interfaces serial, este campo no es necesario y puede dejarse en blanco.

La lectura del fichero de configuración se realiza al inicializar un objeto del tipo GatewayContainer. Como vemos en el siguiente código, cuando se inicializa el objeto se lanza la función LoadInterfaces(). Esta función es la encargada de cargar la configuración del fichero xml a las variables del GatewayContainer.

```
public GatewayContainer(string conf_file)
{
     container = ServiceContainer.GetInstance();
    LoadInterfaces(conf_file);
    transportstatus = new TransportStatus(TransportList);
    transportstatus.addLinks();
}
```

La función LoadInterfaces() recorre secuencialmente y línea a línea el fichero de configuración. Por cada línea del fichero que contenga el atributo interfaz, la función crea un TransportAddress con los atributos que indique la configuración. Una vez creado, el objeto se añade a la lista TransportList en función de su identificador. Cuando finaliza la función se han añadido tantos TransportAddress como interfaces definidas en el fichero y cada una de ellas puede ser utilizada para el envío de datos. El código de la función LoadInterfaces() se encuentra en el CD anexo (**ANEXO 11**) a este documento.

Cada interfaz se guarda como un objeto Link. Este objeto contiene el TransporAddress remoto, la IP local y un entero que da la prioridad que tiene ese enlace instantáneamente. Más adelante se explica cómo se adquiere este valor y para qué sirve.

Para cada etiqueta de control se crea una entrada en la tabla TransportList, la clave es el nombre extraído del atributo Name y como Valor se crea un objeto RemoteGateway.

Este objeto contiene toda la información necesaria para el redireccionamiento de paquetes y información de cada Gateway. En ella contiene la dirección de control que utiliza el terminal y una lista con las diferentes interfaces que están asociadas a esa dirección de control.

La siguiente figura (**Fig. 2.1**) nos muestra visualmente como es esta tabla y la información que contiene.

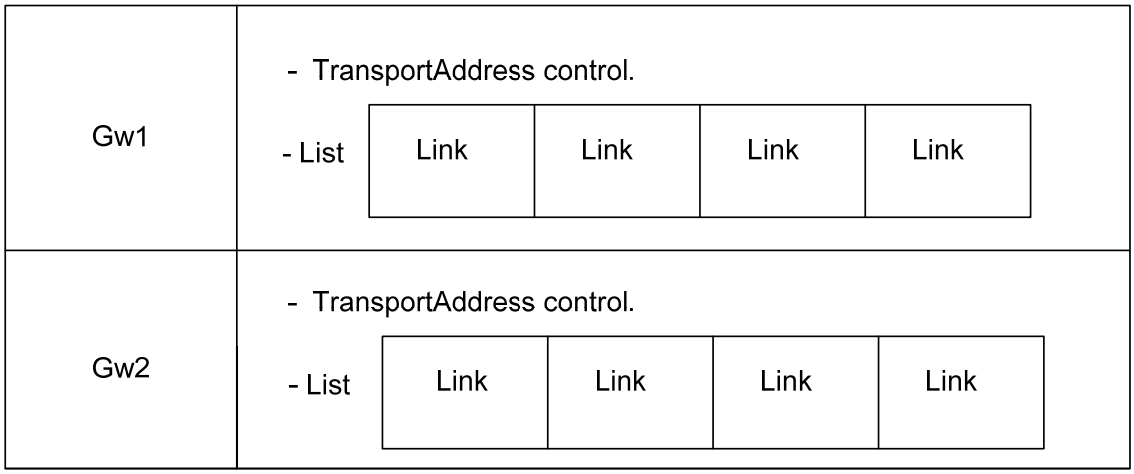


Fig. 2.10 Tabla de configuración Gateway

3.4. Flujo de datos

3.4.1. Mensajes y servicios virtuales

La relación existente entre los mensajes y los servicios virtuales implementados en MAREA no necesitan de ninguna implementación extra para funcionar ya que el protocolo del middleware funciona de manera transparente a través del módulo Gateway.

Para el caso que se expuso en el diseño, el middleware cada vez que le llegan los mensajes de publicación o suscripción correspondientes lo interpreta como si estuviesen en la red local, y los anotará con las direcciones impresas en el campo TransportAddress del mensaje.

La única complejidad que aparece en este punto es la necesidad de hacer un correcto redireccionamiento y apuntar hacia una dirección pública, debido a que en la lista se apunta la dirección correspondiente a la red privada del Gateway remoto.

3.4.2. Mensajes entre clases

Como se ha visto en el Capítulo 1, el flujo de datos dentro del MAREA sigue una estructura de capas: Protocolo, Serialización y Transporte. En la implementación del módulo Gateway se ha rediseñado el flujo de datos a través de dichas capas para adecuarlo a las nuevas necesidades. A continuación se describen las diferencias entre el flujo de datos previamente explicado y el flujo de datos en los nuevos módulos:

La principal diferencia en el comportamiento del protocolo en un servicio/monitor cualquiera y un Gateway, es que éste último además de procesar los mensajes recibidos, encamina el mensaje hacia el módulo Gateway, para ser enviado a uno o varios Gateways en función de su necesidad.

Para lograr esto se ha modificado ligeramente la función MessageReceived(Message m) de la clase ServiceContainer para que, en caso de tratarse de un módulo Gateway procese el mensaje y lo derive hacia la clase GatewayContainer. Esta distinción se ha logrado, una vez más, mediante la variable imGW explicada anteriormente.

Así pues, la clase GatewayContainer se ha programado dentro de la capa de Protocolo con el objetivo de recepcionar los mensajes y volverlos a enviar por la interfaz adecuada. Se podría decir que se ha realizado un “by pass” entre el ServiceContainer y la clase VariableManager derivando los mensajes recibidos al GatewayContainer. Se realica en la función MessageReceived incluida en el CD anexo (**ANEXO 11**) a este documento. De la misma manera que la clase ServiceContainer, la clase GatewayContainer recibe los mensajes mediante

una función llamada MessageReceived(Message Message). En dicha función se realiza el procesado del mensaje que se detalla a continuación.

Con el fin de distinguir los mensajes normales y los enviados por un Gateway, se ha creado un nuevo tipo de mensaje llamado GatewayMessage. Como el resto de tipos de mensaje, el GatewayMessage hereda de la clase Message y se diferencia del resto en que en su definición incluye un objeto 'm' donde se almacena el mensaje original a transportar.

```
[Serializable]
public class GatewayMessage : Message
{
    public object m;
    public GatewayMessage(object Message)
    {
        this.m = Message;
    }
}
```

El tipo de mensaje GatewayMessage se utiliza para realizar una encapsulación del resto de mensajes y enviarlos a través de las interfaces del Gateway. Este encapsulado a nivel de aplicación permite una gestión rápida y simple de los mensajes en el módulo Gateway. En la siguiente figura (**Fig. 2.**) se ejemplifica el encapsulado de la secuencia de los mensajes.

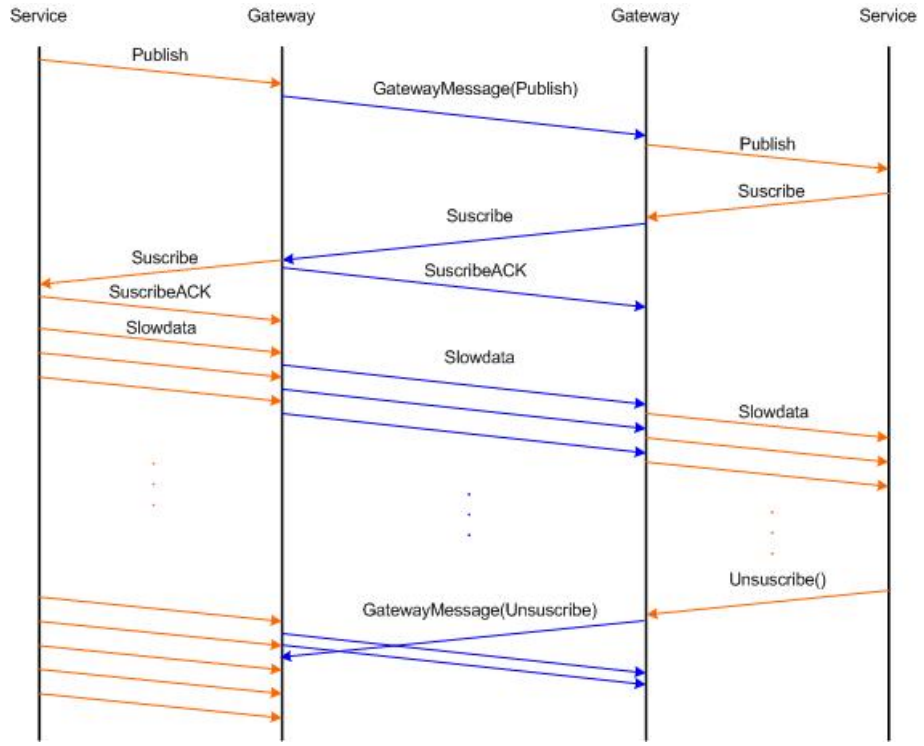


Fig. 2.11 Encapsulado de mensajes

Este encapsulamiento actualmente solo se realiza con los mensajes Publish, Discover, UnPublish y UnSuscribe, debido a que los mensajes de Data y Suscribe los realiza el middleware de manera automática cuando son necesarios, y gracias al selector el propio MAREA encamina el mensaje

mediante el ICGUAS hacia el Gateway remoto determinado, en el siguiente punto se muestra un ejemplo de cómo es este funcionamiento.

Cuando el GatewayContainer recibe un mensaje mediante la función MessageReceived(Message Message), cuyo código se puede encontrar en el CD anexo (**ANEXO 11**) al documento, se comprueba de que tipo de mensaje se trata y se procesa en función del resultado:

- GatewayMessage: en este caso la procedencia del mensaje es implícita, ha llegado a través de una de las interfaces de enlace procedente del Gateway homónimo en el otro extremo de la conexión. Se extrae el mensaje original que transporta el GatewayMessage y se envía por la interfaz local. Gracias a este encapsulado y posterior procesado de los mensajes se consigue una total transparencia entre los servicios/monitores y el módulo Gateway.

- Si se recibe cualquier otro tipo de mensaje, se supone que ha sido enviado por alguno de los servicios/monitores de la red local. En cuyo caso, se encapsula el mensaje original dentro de un mensaje del tipo GatewayMessage y se envía a través de las interfaces del módulo Gateway.

Antes del envío es necesario manipular la información de control del mensaje, más concretamente la dirección de control. Este campo notifica al destinatario de cual es la dirección que tienen que tomar para enviar cualquier dato o mensaje de control (Suscribe, UnSuscribe…), así los servicios o monitores situados en una red local ver todos los servicios como si fueran de la misma red, y no saben que existe otra red local remota que es realmente la que suministra la información, es decir, los monitores o suscriptores ven que el destino final o punto de partida de datos es el terminal encargado del Gateway. En el siguiente apartado se explica un ejemplo donde se puede ver con mayor claridad.

Como se observa en el código, el envío de mensajes se sigue realizando mediante la función del ServiceContainer SendMessage (TransportAddress t, object m). Esta función es la encargada de pasar el mensaje a la capa de Serialización y posteriormente a la capa de transporte para su envío.

```
public void SendMessage(TransportAddress t, object m) {
      coder.Send(t,(Message)m);
}
```

Luego, una vez pasa a la capa de Serialización, se serializa el mensaje de la misma manera que lo hacía el middleware antes de programarse el Gateway, pero después antes de enviar a la capa de transporte, si se trata de un Gateway, el mensaje junto con la dirección de control pasa al selector, donde se realizará un redireccionamiento del mensaje, si es necesario. Este redireccionamiento está descrito en el siguiente punto.

```
    if (servicecontainer.ImGW)
        {
            selector.SendMessage(t, outputBuff.ToArray());
        }
```

```
else
   {
       transport.Send(t, outputBuff.ToArray());
   }
```

3.4.3. Redireccionamiento de mensajes

En este apartado se explica cómo se ha implementado el sistema que hace que los mensajes viajen de un Gateway determinado a otro de la manera más óptima.

Como se ha dicho anteriormente, el redireccionamiento se hace a cargo del Selector, situado entre las capas de serialización y transporte.

Si el mensaje circula por un Gateway antes de ser enviado a la capa de transporte se realiza una función para obtener el TransportAddress de una interfaz a partir de la dirección de control remota que aparece en la tabla de publicadores/suscriptores.

Cuando el Selector es construido se le pasa la tabla TransportList creada en el GatewayContainer y gracias a esta tabla podremos hacer la conversión.

Es necesaria esta conversión ya que si el protocolo de transporte intenta enviar un mensaje a la dirección de control contenida en el mensaje no lo conseguiría, ya que al estar en redes separadas no sería posible su envío.

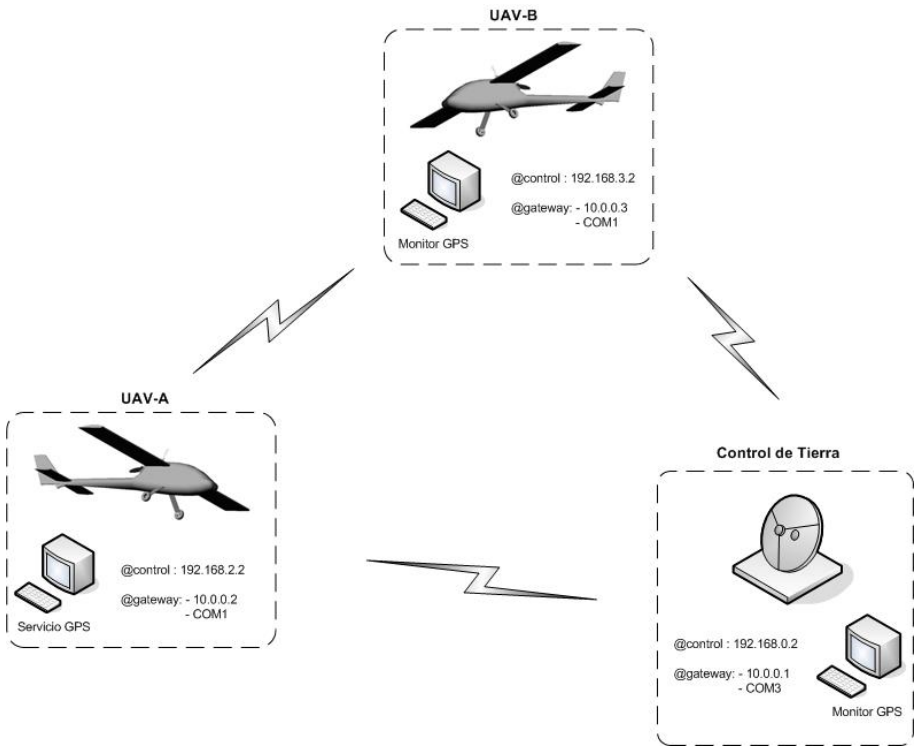


Fig. 2.12 Escenario

Para explicar este proceso de una forma clara, partimos de un ejemplo con el escenario de la **Fig. 2.1**. En el escenario tenemos dos aviones circulando por el

aire y un puesto de control situado en tierra. Tenemos un publicador de variables de localización en uno de los aviones y dos monitores, uno en tierra y el otro en el avión restante.

Cuando los monitores realizan la suscripción envian un Suscribe a la dirección de control del avión que es 192.168.2.2, que al estar en otra red diferente no pueden enviarlo y da como resultado una excepción software. Pero como se trata de un Gateway, entra en acción el Selector.

Primeramente actúa una clase llamada TransportStatus que gracias a la librería System.NetworkInformation (véase [4] y [5]) extrae la información de cada enlace que posee el Terminal y determina cual es su IP, velocidad y estado. Si la interfaz no tiene link no tendrá IP, y por lo tanto su estado es OFF y su velocidad 0.

Una vez extraída la información el TransportStatus crea una tabla de hash llamada Linklist donde la clave es la IP y el valor un objeto de la clase Link, el mismo que se usa para la tabla de redireccionamiento extraída de la carga de configuración (IPs y prioridad, donde la prioridad es la velocidad que tiene el enlace).

Esta tabla se actualiza cada vez que pasa un mensaje por el selector, lo que hace que cada vez que se quiera enviar un mensaje se actualice el estado y se tenga así su estado instantáneo. Una vez está actualizada la tabla se recorre la tabla TransportList y se actualiza la variable de prioridad.

Para el caso de las interfaces que funcionan por el puerto serie, es necesario de algún algoritmo que capture el estado del enlace, para ello se necesita saber que órdenes dispone la tarjeta RF para preguntar sobre el estado de la línea. Actualmente se supone que siempre está activo y la prioridad es fija al valor de 10.
Una vez se ha actualizado esta tabla situada en la clase TransportStatus, la clase Selector antes de buscar la dirección de la interfície que va a utilizar actualiza el valor “priority” de la tabla TransportList.

Esta actualización se hace a partir de un recorrido de las dos tablas (TransportList y LinkList) y comparando las IPs que salen en cada lista. Sí éstas coinciden pasa a actualizar el campo “priority”, sino pasa al siguiente. Al final si existen algún objeto que no coincida en la tabla el valor de “priority” será 0, lo cual quiere decir que no dispone de link activo. Aquí se puede ver como se realiza esta iteración a base de recorridos de las dos tablas:

```
private void RefreshPriority()
{
     int i;
     this.ResetPriority();
     container.Gateway.transportstatus.addLinks();
foreach (string ip in container.Gateway.transportstatus.linkList.Keys)
     {
      foreach (string s in TransportList.Keys)
          {
            for (i = 0; i < TransportList[s].transportlist.Count; i++)
```

```
                {
                   if (ip == TransportList[s].transportlist[i].ip)
                     {
                          TransportList[s].transportlist[i].priority =
                          container.Gateway.transportstatus.linkList[ip].
                          priority;
                     }
                }

            }
      }
}
```

A continuación el Selector busca qué Gateway tiene como dirección de control la 192.168.2.2, ve que se trata del UAV-A y procede a recorrer la lista de links para buscar cual es el link de mayor prioridad:

```
private TransportAddress GetTransportAddress(TransportAddress t)
{
  foreach (string s in TransportList.Keys)
  {
    TransportAddress tmp = (RemoteGateway)TransportList[s]).control;
    if (t == Transport.Broadcast)
     {
      return t;
     }
    if (tmp.ipEndPoint.Address.Address ==
    t.ipEndPoint.Address.Address)
      {
         t=
         GetHighPriority(((RemoteGateway)TransportList[s]).transportli
         st)
          if (t == null)
              return null;
          else
              return t;
      }
   }
  return t;
}
```

Cuando recorre la tabla para extraer la dirección de control pueden pasar varias cosas:

- Que el TransportAddress esté en la tabla: se busca en la lista de Links el enlace que tiene la prioridad más elevada y se devuelve su dirección de control.

- Que la dirección sea de Broadcast o una Unicast que no exista en la tabla de transporte: se devuelve la misma dirección de transporte que recibió el Selector, esto es porque el destino del mensaje no es una máquina situada en la red de otro Gateway, sino una máquina de la propia red local, y no es necesaria ninguna conversión de TransportList.

Si todos tienen prioridad 0, se retorna una TransportAddress “null” con lo cual no se envía el mensaje ya que quiere decir que no hay ningún link activo.

Finalmente, una vez se encuentre el Link por donde ha de viajar el mensaje, el Selector le pasa a la capa de transporte el mensaje previo recibido de la Serialización y le pasa el TransportAddress resultante del Link con mayor prioridad encontrado con dirección de destino 10.0.0.2.

Este proceso es exactamente igual para el resto de mensajes, excepto para el Publish y Discover. En este caso se envía el mensaje por todos los Gateways, recorriendo la tabla TransporList y por cada Gateway remoto se envía el Publish por su mejor Link. De esta forma se asegura que todos los Gateways hayan recibido el mensaje.

Una vez se ha hecho la suscripción el servicio realiza el mismo proceso de envío. En este caso, se envía un mensaje por cada suscritor, el selector realiza el proceso de conversión previamente explicado y envía los mensajes a la IP resultante 10.0.0.1 (tierra) y 10.0.0.3 (UAV-B) (O por puerto COM si el enlace está caído).

3.5. Protocolos de Transporte

Como se ha explicado en los puntos anteriores, las redes locales, tanto a bordo de la UAV como en la estación de tierra, utilizan el protocolo Ethernet (802.3) a nivel de enlace. Por otra parte, el transporte de los mensajes se realiza mediante los protocoles UDP (User Datagram Protocol) y TCP (Transmisión Control Protocol) sobre el puerto 2222. Con el fin de ahorrar ancho de banda y optimizar las comunicaciones, los mensajes que deban ser entregados a más de un destinatario se envían utilizando UDP Broadcast.

En cuanto al módulo Gateway, se utilizan igualmente los protocolos UDP y TCP cuando se esté transmitiendo mediante el enlace Wifi 802.11a. Así pues, se requiere que todos estos mensajes lleguen a su destino. Por ello en las comunicaciones entre Gateways se utiliza, generalmente, el protocolo TCP. En el caso del protocolo Serie se ha diseñado para que emule las características de TCP.

Así pues, el módulo Gateway, además de realizar un encapsulado de los mensajes mediante e GatewayMessage, realiza en la mayoría de los casos un cambio del protocolo de transporte. En la siguiente figura (**Fig. 2.1**) se puede observar el flujo de los datos a través del sistema y como varía el protocolo de transporte en función del segmento que atraviesan los datos. Las flechas naranjas representar el transporte de datos mediante UDP Broadcast desde los servicios hasta el módulo de Gateway y desde éste hasta los monitores; la línea azul representa el transporte de datos mediante TCP o el protocolo serie entre el par de Gateways del sistema:

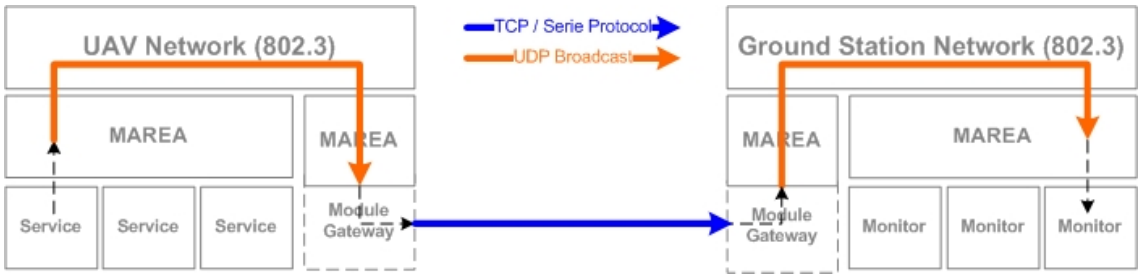


Fig. 2.13 Protocolos Transporte

3.6. Protocolo Serie

Como se ha comentado en los puntos anteriores, se ha diseñado un protocolo que permite la emulación de alguna de las características de la utilización del protocolo TCP sobre Ethernet (véase [1] y [2]). Las características que se pretenden reproducir son:

- Control de flujo
- Detección de errores
- Reconocimiento de tramas

A continuación se formalizan los detalles del protocolo diseñado:

Al igual que en la mayoría de protocolos que se utilizan hoy en día para la transmisión de datos sobre redes (véase [1], [2] y [3]), el protocolo diseñado se basa en la agregación de información extra a los datos útiles a modo de cabeceras y colas mediante un proceso de encapsulado. Dicho proceso consiste en el envío de la información útil juntamente con otros campos que se añaden al inicio y al final de los datos.

3.6.1. Formato trama

Preamble (4 bytes)	Type 1 byte	Sequence 2 bytes	Lenght 2 bytes	Payload 0 – 65535 bytes	CRC 4 bytes

Fig. 2.14 Trama protocolo Serie

Preamble:

Todas las tramas se inician con un campo de preámbulo diseñado para garantizar el correcto alineamiento del mensaje. Este correcto alineamiento garantiza la lectura de todos los campos de la cabecera además de permitir la sincronización entre emisor y receptor en caso de producirse errores en el canal.

El preámbulo está formado por cuatro bytes idénticos cuyo valor es 10101011.

Preámbulo = 10101011 10101011 10101011 10101011

Type:

Este campo indica el tipo de trama que se envía, y por tanto, el procesado que se realiza. El protocolo diseñado define tres tipos distintos de trama:

- Type = 0: tramas que no requieren reconocimiento. Se utilizan para el envío de datos que no requieran confirmación.
- Type = 1: tramas que requieren reconocimiento: Se utilizan para el envío de datos que requieren confirmación.
- Type = 2: trama ACK. Trama utilizada para el reconocimiento de datos.

Se ha estimado oportuno sobredimensionar el tamaño del campo Type, ya que, aún siendo suficientes dos bits para cubrir todos los tipos de mensajes diseñados, en futuras ampliaciones del protocolo los bits restantes pueden resultar de utilidad para definir nuevos tipos de tramas o incluso campos adicionales de opciones.

Sequence:

El campo de secuencia sirve para identificar los mensajes. Este campo se incrementa secuencialmente en una unidad a cada nuevo mensaje enviado. Así pues, existe una correlación entre el mensaje enviado y su respectivo número de secuencia. De este modo cada mensaje puede ser identificado y confirmado mediante su número de secuencia.

Lenght:

Campo que indica la longitud en bytes del campo de datos de la trama. Este campo se utiliza para determinar el tamaño del buffer de lectura en recepción.

Payload:

Este campo transporta la información útil del mensaje. Esta información es entregada por la capa de serialización como un vector de bytes y se añade directamente a la trama conformada.

CRC:

En el canal radio los niveles de ruido son generalmente elevados y por tanto existe una tasa de error considerable. Por este motivo es indispensable la utilización de funciones que indiquen si se han producido errores en la transmisión. Los CRC (Cyclic Redundancy Check) se utilizan para detectar la alteración de datos durante su transmisión ya que resulta muy efectivo en la detección de errores ocasionados por ruido.

El campo de 4 bytes, CRC, de la trama corresponde al valor de la función CRC de 32 bits calculada para la cabecera y el payload de la trama. Las funciones CRC son un tipo de función que recibe un flujo de datos de cualquier longitud

como entrada y devuelve un valor de longitud fija como salida, en este caso 32 bits. En este proyecto se utiliza el término CRC para designar tanto a la función como a su resultado.

Para el cálculo del CRC se dispone de una tabla normalizada de valores uints **(ANEXO 6**). Cuando llega un paquete de bytes para el cálculo de su CRC, coge este paquete, hace grupos de 8 bytes y mediante traslados de símbolos, sumas y multiplicaciones lógicas a nivel de bit, junto con los distintos valores de la tabla anterior, se obtiene un array de 32 bytes. Este es el resultado final de todo el proceso.

3.6.2. Control de errores

Como se ha explicado en el punto anterior, todas las tramas enviadas tienen un campo de CRC para el control de errores. Este campo se calcula en el momento del envío en función de la cabecera y de los datos útiles y se añaden al final de la trama.

En recepción se vuelve a calcular el CRC de la cabecera y los datos recibidos y se compara con el campo CRC recibido. Si ambos valores (el calculado y el recibido) no coinciden significa que se han producido errores en la transmisión y por tanto se descarta la trama (no se procesa). En caso contrario, se extra el payload de la trama y se pasa a la capa superior para su procesado. Este proceso se denomina desencapsulado.

3.6.3. Reconocimiento de tramas

De igual manera que en el protocolo TCP, en el protocolo diseñado, el reconocimiento de mensajes se realiza mediante una trama especial denominada ACK (Acknowledgement). Esta trama se caracteriza con el valor del campo Type de la cabecera igual a 2. El campo Sequence, que en el resto de tramas indica el número de secuencia del mensaje enviado, en el caso de los ACKs indica el número de secuencia del mensaje confirmado.

En las tramas del tipo ACK el campo Payload no se enviará y, por tanto, el campo Lenght será igual a 0. Aun así, se realizará el control de errores de la trama mediante el CRC para confirmar que no se han producido errores en la transmisión de la misma.

3.6.4. Funcionamiento

Como ya se ha ido detallando en los puntos anteriores, el protocolo diseñado realiza diversas funciones (control de flujo, control de errores y reconocimiento de tramas) análoga al protocolo TCP. A continuación se detalla el funcionamiento del protocolo diseñado así como sus principales funciones.

Para facilitar la comprensión del funcionamiento del protocolo se sigue la secuencia lógica de funcionamiento del mismo, desde la creación de las cabeceras y colas en función del tipo de mensaje por parte del emisor, hasta la recepción del reconocimiento de la trama por parte del receptor.

El primer paso que realiza el protocolo serie es la creación de la cabecera y cola de la trama. A partir del mensaje que se quiere enviar, se calculan los campos de la cabecera. El primer campo de la cabecera es el preámbulo. Este campo es siempre igual para todas las tramas y se construye repitiendo cuatro veces el byte 10101011 de manera similar que en el protocolo Ethernet (véase [1] y [2]).

Preámbulo = 10101011 10101011 10101011 10101011

El siguiente campo de la cabecera es el parámetro Type e indica el tipo de trama enviada y depende del tipo del tipo de mensaje que contenga la trama:

- Type = 0, datos que no requieren confirmación
- Type = 1, datos que requieren confirmación
- Type = 2, ACK.

Debido a que los datos a enviar son subministrados por la capa superior ya serializados, como un string de bytes, no hay modo de saber de qué tipo de mensaje se trata a priori. Mediante la función GetType(byte[] m) del objeto SerialMonitor se realiza un deserializado para conocer qué tipo de mensaje es el que se pretende enviar. Esta función devuelve directamente el byte del parámetro Type.

Dependiendo del tipo de mensaje se le asigna un valor u otro al parámetro Type. Como ya se ha comentado, este valor puede ser 0, 1 o 2 en función del tipo de procesado que requieren los datos. En la siguiente tabla (**Tabla 2.1 Tipo de Tramas**) se muestra la relación entre el tipo de mensaje y el valor del campo Type asignado:

Tabla 2.1 Tipo de Tramas

Tipo de mensaje	Type asignado
SlowData	0
Publish	1
Subscribe	1
Discover	1
Register	1
ACK	1

El tercer campo de la trama es el identificador de secuencia. Este campo de 2 bytes tiene un rango de valores de 0 a 65535 ya que esta codificado con el tipo de datos ushort. El encargado de asignar el número de secuencia a cada trama es la función GetSequence() del objeto SerialMonitor.

Como se observa en la figura del diagrama UML (**ANEXO 5**), el objeto SerialMonitor tiene un atributo del tipo ushort llamado sequence que indica el último número de secuencia asignado por la función GetSequence(). El valor de este objeto se inicializa al inicio del programa a 0, y en cada trama enviada se incrementa en una unidad. Antes de asignar un nuevo número de secuencia la función GetSequence() comprueba que el siguiente valor del sequence no se encuentra entre los mensajes pendientes de confirmación, ya que si así fuera, se entraría en conflicto habiendo simultáneamente dos tramas con el mismo identificador. La misma función comprueba, además, que el valor asignado no supere el rango válido del objeto. Si así fuera, se volvería a inicializar a 0 y se reiniciaría el proceso de asignación de números de secuencia. El algoritmo de diseñado es el siguiente:

```
public ushort GetSequence()
{
      bool end = false;

      while (!end){
            if (sequence < 65535)
            {
                  sequence++;
            else
            {
                  sequence = 0;
            }
      end = !PendingPackages.ContainsKey(sequence);
      }
      return sequence;
}
```

El último parámetro de la cabecera es el campo Lenght. Este campo indica la longitud en bytes del campo de datos de la trama. Esta longitud se obtiene mediante el atributo lenght asociado a un objeto del tipo byte[]. Una vez calculada la cabecera de la trama, se calcula el CRC de la cabecera más los datos. Este cálculo, explicado anteriormente, se realiza mediante la clase CRC y la función return_CRCvalues(byte[] packet) incluida en la misma clase.

Por último se envían secuencialmente la cabecera y el CRC calculado juntamente con los datos mediante la función SerialPort.Write(byte[] buffer, int Offset, int count) perteneciente a la librería System.IO.Ports. Esta función envía los datos a través del puerto serie en formato de string de bytes. Si la trama enviada requiere confirmación (type=1), se almacena dentro del Dictionary PendingPackages<ushort, byte[]> mediante la función ActPendingPackeges (ushort seq, byte[] data) de la clase SerialMonitor que más adelante explicaremos.

Cada tiempo RTO (Retransmission Time Out) se recorre la tabla PendingPackages y se vuelven a enviar los mensajes que aún están pendientes de confirmación. Este proceso se realiza mediante la función SendPending() del SerialMonitor que se encuentra junto al resto de código en el CD anexo (**ANEXO 11)** a este documento.

El tiempo RTO (Retransmission Time Out) es un parámetro ajustable por el administrador del sistema en unidades de milésimas de segundo mediante del fichero de configuración del Gateway. Por defecto se toma un valor de RTO igual al RTT (Round-Trip delay Time). El tiempo RTT se calcula como el tiempo que tarda un paquete enviado desde un emisor en volver a este mismo emisor habiendo pasado por el receptor de destino. En nuestro caso se realizan los cálculos para una trama genérica de 200 bytes a través de un enlace de 9600 bps (suponiendo que no hay errores de transmisión y despreciando el tiempo de procesado y de propagación):

$$RTO = RTT = 2 \times (200*8/9600) \approx 350 \text{ ms}$$

Tanto en el emisor como en el receptor siempre habrá un hilo de ejecución activo escuchando el puerto serie de entrada. Al detectar la recepción de byte por dicho puerto se inicia el proceso de sincronización. Este proceso consiste en la lectura byte a byte del buffer de entrada hasta detectar la secuencia del preámbulo. Dicha secuencia consiste en cuatro byte idénticos:

Preámbulo = 10101011 10101011 10101011 10101011

Una vez localizado el preámbulo dentro del buffer de lectura, y por tanto alineada la trama, el receptor lee el resto de la cabecera y almacena el campo Type, Secuence y Lenght. Gracias al campo Lenght el receptor conoce la longitud exacta en bytes de los datos que debe leer y almacenar. Una vez almacenada la información útil se leen los cuatro bytes restantes pertenecientes al CRC de la trama.

Una vez leída y almacenada la totalidad de la trama, se realiza el control de errores. Tal y como se ha explicado anteriormente, se calcula la función CRC de 32 bits de la cabecera y los datos y se compara con el valor del campo CRC recibido. En caso de no coincidir significa que han ocurrido errores durante la transmisión y que por tanto la trama no es válida y debe desecharse. En caso contrario se pasa la información útil de la trama a la capa superior.

Si la trama recibida es del tipo 1. Es decir, que contiene datos que precisan confirmación, se envía un ACK con el número de secuencia del mensaje informando de su correcta recepción. Como ya se ha explicado, la trama ACK se identifica con el campo Type igual a 2 e identifica el mensaje confirmado mediante su número de secuencia. Este tipo de tramas no transportan datos útiles.

Cuando el emisor del mensaje reciba un ACK entiende que una de las tramas que ha enviado ha sido recibida correctamente y por tanto ejecuta la función ActPendingPackeges(ushort seq, byte[] data) del Serial Monitor para eliminar la trama de la lista de paquetes pendientes de confirmación.

Esta función se utiliza indistintamente para añadir mensajes en la lista y para eliminarlos de ella. Como se ve en el código anterior, si el número de secuencia ya se encuentra en la lista se entiende que dicho número procede de un ACK y por lo tanto se eliminan los datos de la lista, en caso contrario se entiende que

se trata del número de secuencia de una trama pendiente de confirmación y se añade a la lista.

Para entender de un modo más visual cómo funciona el protocolo serie en cuanto al recibimiento y el tratado de cada mensaje, se muestra en la siguiente figura (**Fig. 2.**) el diagrama de estados del protocolo serie.

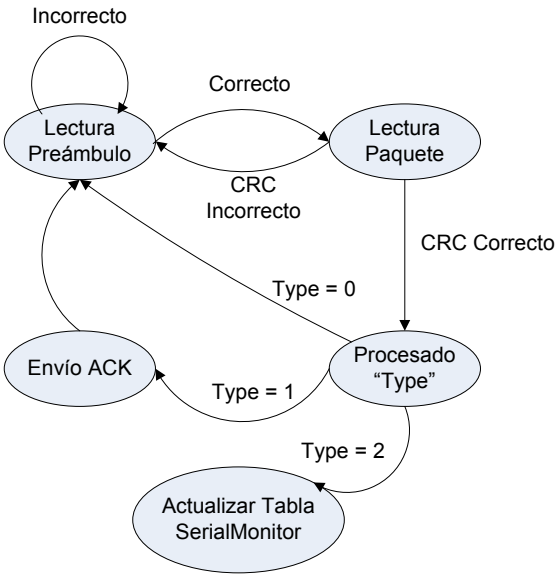


Fig. 2.15 Diagrama de estados del protocolo Serie

Una vez se procesa el paquete, si se trata de un tipo 0 y 1, se pasa el mensaje a la capa de codificación y de allí sigue su curso normal dentro de la arquitectura del middleware.

CAPÍTULO 4. PRUEBAS

4.1. Introducción

En el siguiente apartado se realizan las pruebas necesarias para confirmar el correcto funcionamiento del proyecto planteado. Para ello se han creado dos bloques de pruebas diferenciados:

a. Pruebas de funcionamiento: diseñadas para validar el correcto funcionamiento del proyecto dentro de un escenario real.
b. Pruebas de rendimiento: diseñadas para evaluar el rendimiento del proyecto en un escenario real.

Con estas pruebas se pretende validar el correcto funcionamiento del diseño e implementaciones realizadas así como su correcta inclusión en el protocolo de MAREA.

4.2. Escenario planteado

El escenario planteado para la realización de ambos bloques de pruebas consta de los siguientes componentes:

- Equipos SC1 y SC2:

    - Intel® Core™ Duo CPU E4500 @ 2.20GHz
    - 2 GB de RAM
    - Intel® 82566DM Gigabit Network Connection
    - Microsoft Windows XP Professional Version 2002 SP2
    - Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM
    - Microsoft .NET Framework Version 3.5

- Equipos GW1 y GW2:

    - Intel® Pentium® M 1.87 GHz
    - 1 GB de RAM
    - Broadcom NetXtreme Gigabit Ethernet
    - Intel® PRO/Wirekess 2200BG network Connection
    - Puerto Comunicaciones (COM1)
    - Microsoft Windows XP Professional Version 2002 SP2
    - Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM
    - Microsoft .NET Framework Version 3.5

En la siguiente figura (**Fig. 3.1**) se observa el escenario y direccionamiento utilizado para la realización de las pruebas, tanto de funcionamiento como de rendimiento.

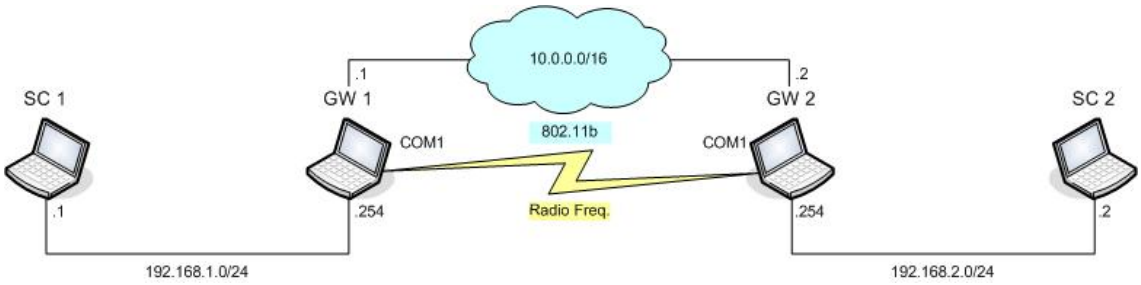


Fig 4.1 Escenario y direccionamiento

4.3. Ficheros de configuración

Dado el escenario planteado se han configurado los equipos GW1 y GW2 con los siguientes ficheros de configuración:

GW1:

```
<local address="192.168.1.255"  port="11811" />

<control name="GW2" localaddress="10.0.0.2" port="11811" />

   <interface name="GW2" type="ip" localaddress="10.0.0.1"
   remoteaddress="10.0.0.2" port="11811" />

   <interface name="GW2" type="serial" localaddress="COM1"
   remoteaddress="COM1" port=""/>

<RTO value="3000"/>
```

GW2:

```
<local address="192.168.2.255"  port="11811" />

<control name="GW2" localaddress="10.0.0.1" port="11811" />

   <interface name="GW2" type="ip" localaddress="10.0.0.2"
   remoteaddress="10.0.0.1" port="11811" />

   <interface name="GW2" type="serial" localaddress="COM1"
   remoteaddress="COM1" port=""/>

<RTO value="3000"/>
```

4.4. Pruebas de Funcionamiento

Uno de los objetivos que se perseguían en este proyecto es integrar el módulo Gateway con el protocolo de comunicaciones de MAREA y que ello no afectara al funcionamiento global del sistema. Con el fin de validar la integración entre Gateway y MAREA se ha diseñado el siguiente experimento práctico donde se testeará en un escenario real el funcionamiento del diseño realizado.

Para plasmar en la medida de lo posible todas las casuísticas que puedan darse en un entorno real, se han realizado las siguientes pruebas:

1. Publicación y despublicación de eventos y variables
2. Suscripción y desuscripción de eventos y variables
3. Pedida de enlace wifi y transmisión por enlace Radio
4. Recuperación de enlace wifi

En el **ANEXO 7**: Capturas pruebas funcionamiento se muestran los resultados obtenidos a modo de captura de pantalla para cada una de las pruebas especificadas.

En primer lugar podemos observar la pantalla inicial que muestra el correcto arranque de MAREA en modo Gateway (**ANEXO 7: Fig. 1 y Fig. 2**). Como puede observarse, tanto en el GW1 como en el GW2 se han abierto sendos puertos de comunicación tanto en la red local (192.168.X.0/24) como en la red de interconexión (10.0.0.0/16).

Las siguientes capturas (**ANEXO 7: Fig. 3 y Fig. 4**) ilustran las tablas de direccionamiento para cada uno de los enlaces de cada Gateway así como las prioridades de cada una de ellas. Como puede verse, en ambos Gateways hay dos rutas que apuntan al otro extremo de la comunicación: un enlace a través del puerto COM con prioridad 10 ya que la velocidad definida es de 9600bps, y un enlace a través de la IP 10.0.0.2 con prioridad 54000 ya que se realiza a través de una red 802.11g con una velocidad nominal de 54Mbps.

4.4.1. Publicación y despublicación de eventos y variables

En el siguiente apartado se pretende validar el correcto funcionamiento del proceso de publicación y despublicación de Variables y Eventos. Para ello el SC1 publica una variable con el nombre de Longitud. En la figura (**ANEXO 7: Fig. 5**) correspondiente al GW1 recibe un mensaje Publish desde la dirección de control del SC1.

A continuación el GW2 reenvía en mensaje encapsulado como GatewayMessage que es recibido por el GW2. En este caso se puede apreciar en la figura (**ANEXO 7: Fig. 6**) como se ha sustituido la dirección de control del SC1 por la del GW1.

Seguidamente se pueden ver en las tablas de publicadores/suscriptores del GW1 y GW2 respectivamente (**ANEXO 7: Fig. 7 y Fig. 8**) como la publicación de una variable se ha efectuado correctamente, y a continuación se realiza la despublicación de la misma variable (**ANEXO 7: Fig. 9 y Fig. 10**).

4.4.2. Suscripción y desuscripción de eventos y variables

En el siguiente apartado se pretende validar el correcto funcionamiento del proceso de suscripción y desuscripción de Variables y Eventos. Para ello el SC2 se suscribirá a la Variable con el nombre de Longitud. En la figura (**ANEXO 7: Fig. 11**) correspondiente al GW2 se observa como recibe un mensaje Suscribe desde la dirección de control del SC2.

A continuación el GW2 reenvía el mensaje al GW1 sustituyendo la dirección de control del SC2 por la suya. En la figura (**ANEXO 7: Fig. 12**) se observa como el GW1 recibe el mensaje de Suscribe desde GW1 y posteriormente como GW2 recibe un mensaje de SuscribeACK desde GW1 (**ANEXO 7: Fig. 13**).

Como se puede observar en las figuras (**ANEXO 7: Fig. 14 y Fig. 15**) correspondientes a las tablas de publicadores/suscriptores de GW1 y GW2 respectivamente el proceso de suscripción se ha realizado correctamente.

En la (**ANEXO 7: Fig. 16**) se observa el envío del mensaje UnSuscribe con el fin de desuscribirse de la variable. A continuación en las figuras (**ANEXO 7: Fig. 17 y Fig. 18**) correspondientes a las tablas de publicadores/suscriptores de GW1 y GW2 respectivamente se observa que el proceso de desuscripción se ha realizado correctamente.

4.4.3. Pérdida de enlace

En este set de pruebas lo que se pretende es comprobar el correcto funcionamiento del protocolo en caso de perdida del enlace 802.11a. Para ello simularemos este escenario desconectando la red inalámbrica creada entre el GW1 y GW2 (**ANEXO 7: Fig. 19**) tal y como se puede ver en la siguiente figura (**Fig. 3.2**).

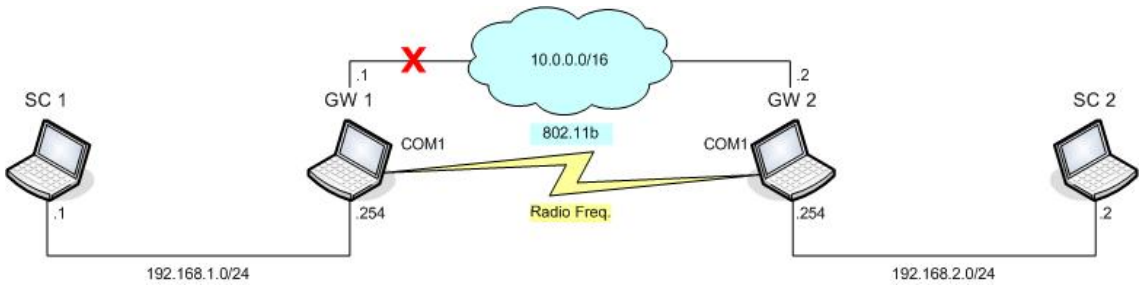


Fig. 3.2 Escenario con caida de enlace

Una vez desconectada la red wifi el enlace que unía ambos Gateways mediante 802.11a desaparece de la lista de conexiones del Gateway (**ANEXO 7: Fig. 20**). Así púes, en este escenario la comunicación entre ambos módulos se reduce únicamente al enlace radio frecuencia tal y como se ve en la anterior figura.

Cuando el ICGUAS detecta la perdida del enlace entre conmuta la transmisión de paquetes al protocolo serie. En las figuras (**ANEXO 7: Fig. 21**) se puede observar la transmisión y recepción de paquetes desde que se pierde el enlace hasta que vuelve a recuperarse.

Una vez recuperada la conexión wifi entre el GW1 y GW2 el enlace vuelve a aparecer en la tabla de enlaces disponibles de ambos Gateways y por tanto la comunicación puede restablecerse (**ANEXO 7: Fig. 22 y Fig. 23**).

4.4.4. Resultados

Como se ha podido comprobar a lo largo de las pruebas realizadas, el funcionamiento de los módulos Gateway ha sido el esperado conforme a lo establecido a lo largo de todo el proyecto, tanto a nivel de diseño como de integración dentro del protocolo de MAREA.

En todo momento el comportamiento del software ha sido el esperado y no se han reportado errores en su ejecución.

4.5. Pruebas de Rendimiento

En el siguiente ser de pruebas se pretende verificar el correcto funcionamiento del software diseñado a nivel de rendimiento así como compararlo con las prestaciones ofrecidas por el protocolo de MAREA sin el módulo Gateway en funcionamiento. Para ello se han diseñado una batería de pruebas consistentes en el envío de paquetes a diferentes longitudes y frecuencias, tanto de Eventos (TCP) como de Variables (UDP Broadcast).

4.5.1. Software test

El software que se va a utilizar para comprobar el rendimiento de este proyecto es un software diseñado e implementado por uno de los grupos de trabajo del proyecto ICARUS. El objetivo de este software es analizar la latencia en el envío de mensajes desde que se crea hasta que se recibe en el otro extremo de la comunicación. Evaluando así el rendimiento del protocolo tanto a nivel de sistema operativo como a nivel de red.

El software de test está diseñado para operar siguiendo el esquema de Publish/Suscribe que ya se ha comentado a lo largo del proyecto. Por este motivo utilizará la interface de comunicación que proporciona MAREA como si se tratara de cualquier otro servicio: publicará, se suscribirá y informará de cambios de variable, evento según esté configurado.

Los datos obtenidos por el programa se guardan en formato texto en un fichero de la raíz del disco duro de la máquina que ejecuta como Publicador el test. Una vez finalizado el test el mismo software permite generar una serie de gráficas donde se puede apreciar el retardo de cada paquete y un histograma del retardo, aportando así una herramienta muy potente para evaluar el rendimiento del protocolo. Las siguientes figuras (**Fig. 3.3 y Fig 3.4**) corresponden a las graficas tipo que genera el software:

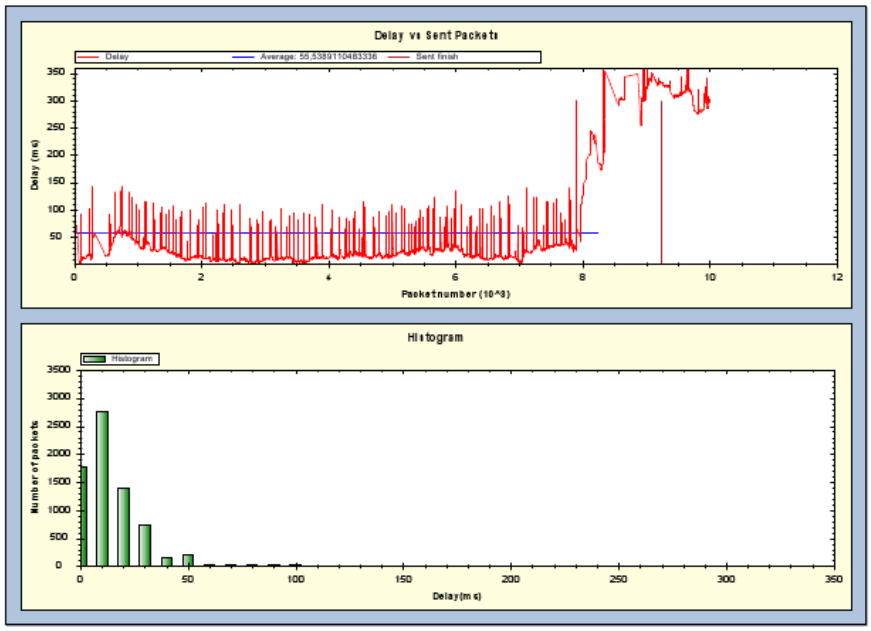


Fig. 4.3 – Gráficas software Test

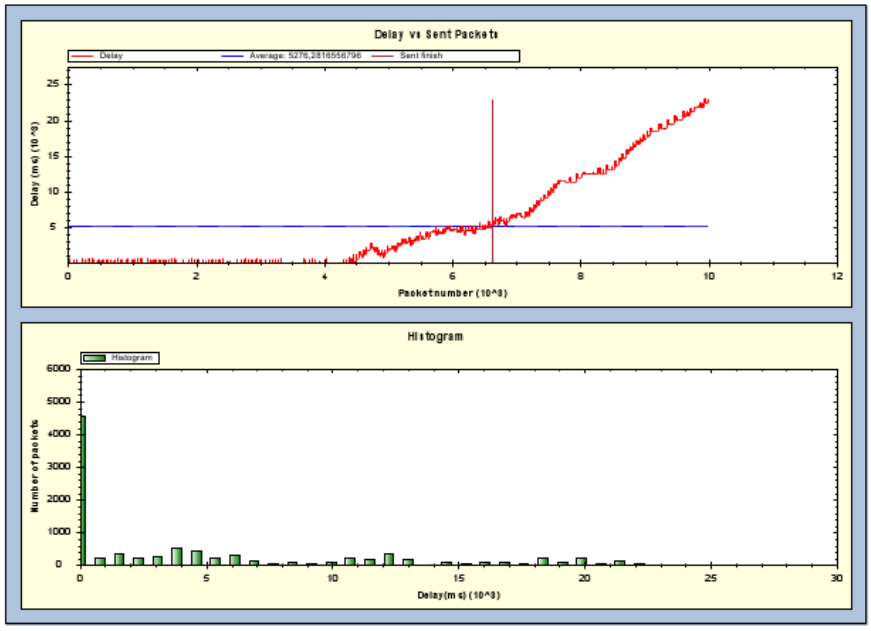


Fig. 4.4 – Gráficas software Test

El software de test requiere una parametrización previa que permite adaptar cada prueba a un determinado escenario a evaluar. Los parámetros que se permiten modificar son:

- Tipo de datos: tipo de datos con el que se rellenará el campo de datos de los paquetes enviados. Pueden ser: Byte, Int, Long o MFTP.

- Modo de test: el software permite realizar cuatro tipos de test diferentes:

- Round-Trip: modo específico para evaluar el retardo de la comunicación entre dos host. Envía paquetes esperando la confirmación del anterior antes de enviar el siguiente.

- Full: modo específico para evaluar el ancho de banda soportado por el enlace o por el sistema operativo. Intenta enviar todos los mensajes sin esperar confirmación.

- Specified Frequency: envía paquetes a una determinada frecuencia medida en Herzios.

- Automatic Frequency: igual que el modo anterior pero la frecuencia de envío la determina el programa en función del retardo medio de los paquetes previos.

- Tipo de primitiva: podemos elegir el tipo de primitivas con el cual queremos realizar el test. En función del tipo de primitivas el protocolo de transporte variará y por tanto los resultados serán diferentes.

- Longitud del paquete: permite definir la longitud en bytes de los paquetes enviados.

- Numero de paquetes: permite definir el número total de paquetes que se enviaran durante la prueba.

4.5.2. Test control

Con el fin de tener una referencia de funcionamiento en condiciones óptimas, se toma como referencia de control un test realizado con el escenario de prueba pero sustituyendo el enlace 802.11a por un enlace ethernet a 100Mbps.

En las siguientes figuras (**Fig. 3.5 y Fig. 3.6**) se puede observar la grafica del retardo en función del tamaño de paquete y frecuencia de envío tanto para Eventos como para Variables en el test de control (**ANEXO 8**)

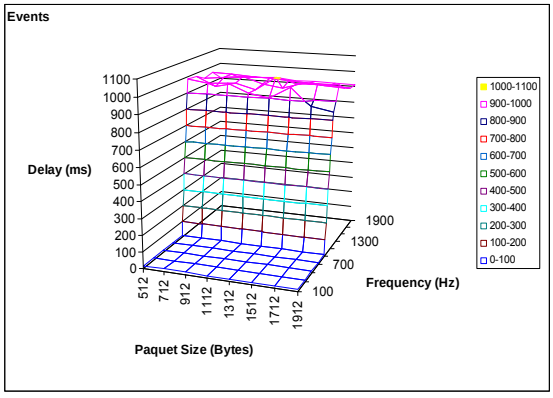


Fig 4.5 Gráfica I

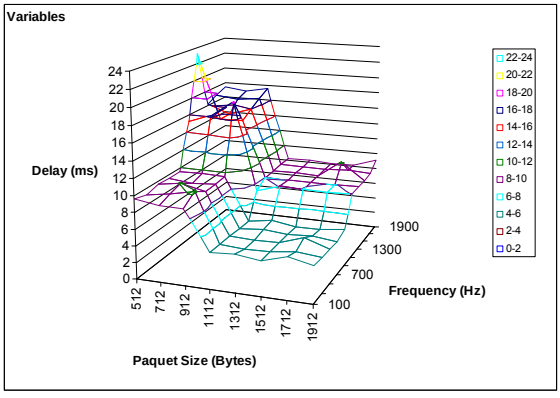


Fig 4.6 Gráfica II

4.5.3. Test wifi

En las siguientes figuras (**Fig. 3.7 y Fig. 3.8**) se puede observar la grafica del retardo en función del tamaño de paquete y frecuencia de envío tanto para Eventos como para Variables en el test de 802.11a (**ANEXO 9**).

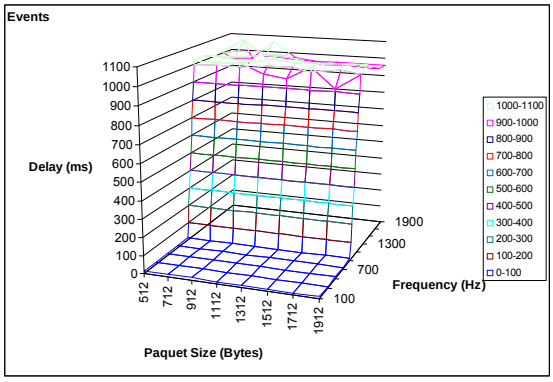


Fig 4.7 Gráfica III

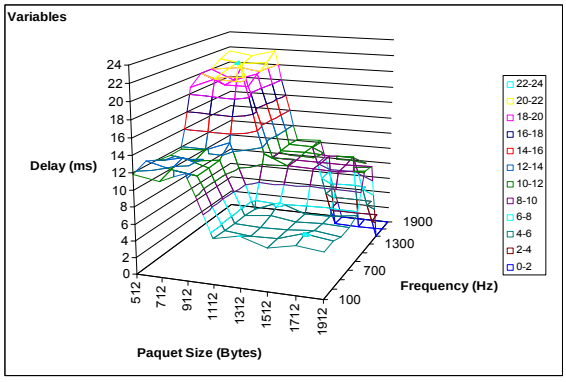


Fig 4.8 Gráfica IV

4.5.4. Test Radio Frecuencia

En las siguientes figuras (**Fig. 3.9 y Fig. 3.10**) se puede observar la grafica del retardo en función del tamaño de paquete y frecuencia de envío tanto para Eventos como para Variables en el test de radio frecuencia (**ANEXO 10**).

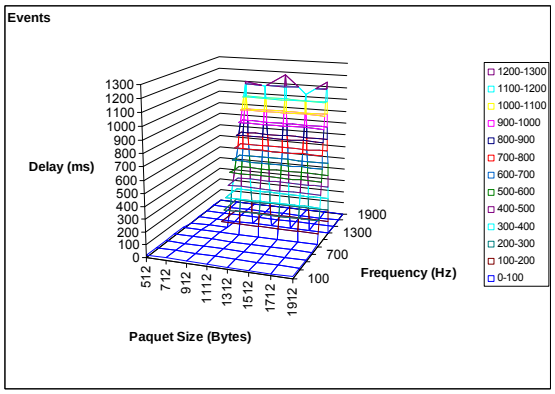


Fig 3.9 Gráfica V

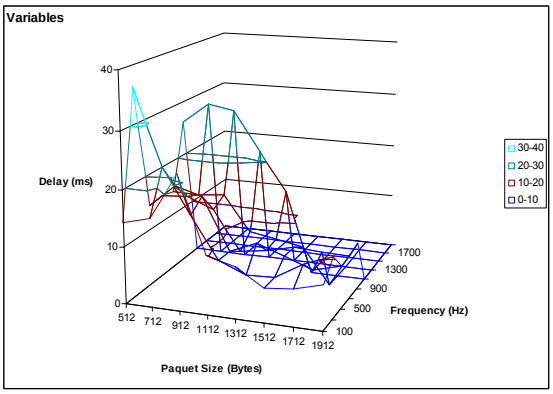


Fig. 3.10 Gráfica VI

4.5.5. Resultados

Como se puede observar en las tablas de resultados de cada test (véanse anexos) los retardos observados se pueden dividir en cuatro zonas distintas en función de la frecuencia de envío y el tamaño de los paquetes. Podemos observar este comportamiento en la siguiente figura (**Tabla 1**):

Tabla 3.1 Bytes/Frecuencia

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	Zona 1					Zona 3				
712										
912										
1112	Zona 2					Zona 4				
1312										
1512										
1712										
1912										

Aunque las pruebas se han realizado en las cuatro zonas mencionadas, la zona de trabajo del protocolo MAREA se centra en la Zona 1 y en la Zona 2 en casos puntuales. Para la Zona 1 el incremento retardos medios en las pruebas Wifi y RF en comparación con el test de control son los siguientes:

Tabla 3.2 Incremento del retardo

	Eventos	Variables
Wifi	4,7%	25,3%
RF	31,3%	85,7%

De la tabla anterior podemos concluir que el retardo introducido por el enlace wifi es aceptable para el envío de Variables y Eventos. En cambio, para el enlace RF el rendimiento se deteriora sustancialmente con el uso de Variables. Este retardo es debido al uso de UDP y envío sin confirmación, en wifi y el protocolo RF respectivamente, lo que introduce un retardo extra debido a la congestión del enlace. Así púes, en este caso debería considerarse la posibilidad de restringir el uso del enlace RF al envío de Eventos (véase [10]).

Por otra parte se ha observado que la codificación realizada por las clases de .net (véase [4] y [5]) utilizadas en la serialización de los datos no es óptima, incrementando hasta en un 200% el tamaño inicial de los datos. Mejorando este aspecto el rendimiento del Gateway de comunicaciones, así como todo el protocolo, mejoraría considerablemente.

Por otro lado, si nos fijamos en los datos de retardo obtenidos, se observa que en todos los casos el envío de variables introduce un retardo sensiblemente inferior que los eventos. Esto es debido a que el protocolo de comunicaciones de MAREA precisa reconocimiento para los Eventos y no para las Variables debido a la naturaleza de cada una de las primitivas.

CAPÍTULO 5. CONCLUSIONES

A través de los distintos capítulos del presente Trabajo Final de Carrera, se han realizado diversas observaciones, discusiones y pruebas importantes, las cuáles tienen relación directa con los objetivos expuestos en la introducción de este documento. A continuación, se hará una recopilación de todas estas consideraciones con el propósito de satisfacer tanto los objetivos principales del que parte el proyecto, como cada uno de los objetivos secundarios que han ido apareciendo a lo largo de su desarrollo.

5.1. Conclusiones generales

Tal y como se describe en la introducción de este TFC, **Intelligent Communications Gateway for Unmanned Airborne Systems**, los tres objetivos generales de los que parte el proyecto son: documentar MAREA, diseñar el Gateway de comunicaciones e implementar una primera versión del Gateway de comunicaciones. En los capítulos en que se ha estructurada el documento se recoge el desarrollo llevado a cabo para lograr dichos objetivos generales.

En el primer capítulo, El Middleware MAREA, se ha documentado exhaustivamente el protocolo de comunicaciones diseñado por el grupo de trabajo ICARUS. Hasta el momento, la documentación formal existente referente al protocolo era prácticamente nula y se limitaba a presentaciones y artículos sobre el funcionamiento general del protocolo. La documentación generada en el capítulo primero de este TFC servirá de base para todos los futuros desarrollos que se realicen sobre MAREA y como documento didáctico para todos los futuros desarrolladores que trabajen en el proyecto.

En los siguientes tres capítulos del TFC (Diseño, Implementación y Pruebas) se ha diseñado, implementado y probado el correcto funcionamiento del Gateway alcanzando así los dos últimos objetivos propuestos al inicio del proyecto. Pese a contar desde un principio con una idea clara de diseño e implementación que seguir, el proceso de diseño y desarrollo se ha ido realimentado así mismo aportando nuevas ideas y enfoques que se han ido adaptando a las necesidades del proyecto. Aun así, muchas de estas nuevas aportaciones se han dejado pendientes para nuevas implementaciones al salirse del alcance del proyecto propuesto.

Cabe destacar, que el trabajo que realiza el grupo ICARUS con MAREA es un proyecto vivo y en pleno desarrollo. En ocasiones ha resultado complicado adaptar el avance del TFC con la evolución del protocolo MAREA ya que éste ha cambiado sustancialmente durante el periodo de realización del TFC. Este proceso de avance y retroceso ha comportado en ocasiones el rediseño y planteamiento de partes importantes del proyecto, provocando un retraso considerable en su desarrollo. Aun así, se ha conseguido implementar exitosamente un módulo de Gateway y adaptarlo a la última versión de MAREA desarrollada en el momento de la finalización de este proyecto lo cual

consideramos un logro tanto para el equipo de ICARUS como para los estudiantes que han realizado el TFC.

5.2. Conclusiones específicas

El desarrollo de los objetivos principales de este Trabajo Final de Carrera conllevaba implícitamente la consecución de unos objetivos específicos tales como: estudio del protocolo de comunicación de MAREA, estudio de la implementación y clases de MAREA, diseño de la estructura del Gateway de comunicaciones, programación de las clases que forman el ICGUAS, diseño de protocolo de comunicación sobre puerto de comunicaciones serie, ensamblaje del módulo diseñado con MAREA, realización de pruebas de funcionamiento y la realización de pruebas de rendimiento del protocolo.
La realización de todos estos objetivos específicos han conllevado a la postre la correcta consecución de los objetivos principales del proyecto tal y como se ha comentado en el apartado anterior.

En primer lugar, se realizó la documentación referente al Middleware MAREA. Esta tarea, a priori sencilla, se retrasó debido a la falta de documentación existente y a los cambios realizados en el protocolo durante el desarrollo del proyecto. Por otro lado, esta documentación ha sido la base del trabajo realizado posteriormente ya que para el diseño e implementación del Gateway se ha tenido muy en cuenta el funcionamiento de MAREA. Así mismo se analizó en detalle el protocolo de comunicaciones y la estructura de capas del software conformando un documento muy interesante a nivel didáctico para los futuros trabajos basados en MAREA. Este documento se incluye como parte del TFC en el Capítulo 1.

Una vez realizado el trabajo previo de documentación se inició el proceso de diseño del ICGUAS partiendo de las directrices acordadas conjuntamente con el director y subdirector de este Trabajo Final de Carrera, Juan López Rubio y Pablo Royo Chic, respectivamente. Estas directrices contemplaban aspectos como: mantener la integridad con el protocolo MAREA, permitir la parametrización del módulo Gateway, controlar el estado de los enlaces e implementar el enlace radio mediante la placa 24XStream RF que se utilizará a bordo de un UAV de pruebas. En los capítulos dos y tres de este documento se describen los pasos seguidos para intentar satisfacer todos estos aspectos.

Cabe destacar que mantener la integridad del protocolo de MAREA conjuntamente con el módulo Gateway ha sido uno de los puntos más críticos del proyecto. Es evidente que si el software diseñado no podía funcionar conjuntamente con la última versión del protocolo su realización no hubiera resultado interesante para el desarrollo del proyecto. Este aspecto se ha visto complicado debido a la rápida progresión del desarrollo de MAREA. A modo de ejemplo, al inició del TFC se empezó a trabajar con la versión r34 y en su finalización se integró con la versión r64 de MAREA.

Una fase especial dentro del proceso de diseño e implementación del ICGUAS ha sido el protocolo realizado específicamente para el enlace radio frecuencia a

través del puerto serie de comunicación. Este protocolo se diseñó con el fin de emular las características básicas que ofrece TCP, es decir: control de flujo, control de orden de los paquetes, control de errores, confirmación y retransmisión de paquetes. Todas estas características han sido diseñadas e implementadas satisfactoriamente en la clase SerialTransport del proyecto y probadas sobre la placa 24XStream RF con éxito.

En cuanto a los resultados de las pruebas, cabe decir que no han sido los esperados. Si bien es cierto que para el enlace wifi el retardo introducido en el sistema es aceptable, para el enlace RF todo parece indicar que se debe descartar la utilización de Variables para el envío de datos. Como ya se ha comentado, parte de este problema viene debido a la congestión que produce el protocolo de transporte utilizado y al incremento del volumen de los datos después de pasar por la capa de Codificación y el proceso de serialización de .Net (véase [4] y [5]). Probablemente optimizando este proceso el rendimiento del conjunto del protocolo mejoraría sustancialmente.

5.3. Consideraciones medioambientales

El desarrollo de éste trabajo afecta de modo indirecto al medioambiente, y contribuye a la creación de tecnología que posibilita el avance hacia una sociedad sostenible y cuidadosa con su entorno. Estos avances se centran en el campo de la prevención y control de los recursos naturales.

El **Intelligent Communications Gateway for Unmanned Airborne Systems** como ya se ha comentado en la introducción, forma parte del proyecto MAREA. Éste a su vez procede de la propuesta a nivel europeo “RED EYE”. Ésta es una es una aplicación diseñada para ayudar a los equipos de prevención y extinción de incendios a seguir y analizar los frentes y posibles focos de los incendios forestales (**Fig 4.1**). La interfície de usuario es capaz de guiar al piloto del UAV (o un piloto automático) a través de un plan de vuelo previamente designado. Mientras tanto un sistema de infrarrojos es capaz de localizar los focos del incendio y posicionarlos mediante GPS. Estos datos son enviados en tiempo real a las patrullas de extinción que están trabajando sobre el terreno o a los supervisores para su evaluación y toma de decisiones.

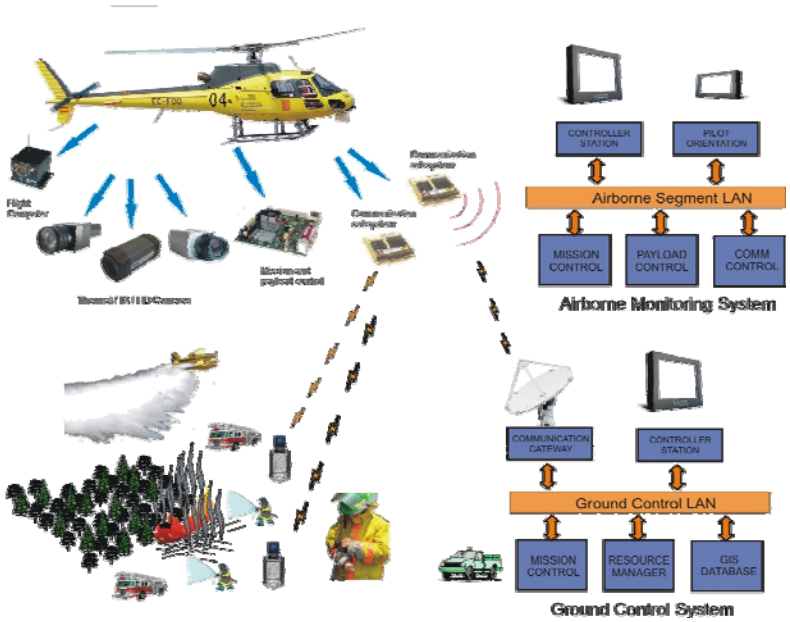


Fig. 4.1 Uso de Uvas como elementos de seguridad

De esta manera, la aplicación y utilidad del presente TFC pueden tener un impacto directo en la prevención y control de los recursos naturales.

5.4. Futuras líneas de trabajo

A lo largo de este trabajo se han comentado futuras líneas de trabajo y nuevas mejoras que sería interesante desarrollar, dando así continuidad a este proyecto. Todas estas mejoras se centran principalmente en el módulo Gateway diseñado en el proyecto aunque algunas de ellas podrían extrapolarse al conjunto del protocolo de MAREA. A continuación se resumen brevemente.

En un futuro se prevé dotar a todas las comunicaciones de MAREA, y en especial a las que llevará a cabo el Gateway, de un sistema de cifrado que garanticen la confidencialidad e integridad de los datos. Este es un punto fundamental del proyecto que no se ha realizado por salirse de la escala temporal asignada al TFC aunque se ha estructurado el diseño de la aplicación de tal manera que la inclusión de una capa de seguridad dentro del protocolo resulta fácil y rápida.

Por otro parte, la métrica escogida en primera instancia para la elección del enlace de transmisión de datos es, como ya se ha comentado en el documento, el ancho de banda de los enlaces. Existen múltiples alternativas para esta elección que deben estudiarse en un futuro. Así mismo, la configuración de red de cada uno de los enlaces del Gateway se realiza estáticamente por el administrador del sistema, en una futura implementación se diseñará un algoritmo capaz de autoconfigurar todos los enlaces disponibles del UAV así como la prioridad de cada uno de ellos.

Por último, con el fin de mejorar el rendimiento en las comunicaciones del protocolo, y en especial las que involucran los enlaces inalámbricos, se debería estudiar la posibilidad de volver implementar la capa de codificación de MAREA con el fin de optimizarla.

BIBLIOGRAFÍA

[1] Tanenbaum A.S., *Redes de computadoras* 4ª ed. México Editorial Pearson Educación, cop. 2003.

[2] Stallings W., *Comunicaciones y redes de computadores,* 7ª ed, Editorial Pearson Educación, 2004.

[3] Holzmann G.J.; Cliffs E., *Design and validation of computer protocols*, Editorial Prentice Hall, cop. 1991.

[4] Abrams, B.; Abrams, T., "Networking Library, Reflection Library, And Xml Library", Volumen 2 en .net Framework Standard Library Annotated Reference, Editorial ADDISON-WESLEY 2005.

[5] Librerías y clases Microsoft .NET, *Visual C#,* http://msdn.microsoft.com/es-es/library/kx37x362(VS.80).aspx (último acceso 05/05/09).

[6] López J.; Royo P.; Pastor E.; Barrado C.; Santamaria E., "A Middleware Architecture for Unmanned Aircraft Avionics"

[7] López J.; Royo P.; Pastor E.; Barrado C., "Service Abstraction Layer for UAV Flexible Application Development", *Computer Architecture Dept., UPC, Spain.*

[8] López J.; Royo P.; Pastor E.; Barrado C., "Applyin MAREA middleware to UAS communications".

[9] Timothy X Brown; Brian Argrow; Cory Dixon; Sheetalkumar Doshi; Roshan-George Thekkekunnel; Daniel Henkel, "Ad Hoc UAV Ground Network (AUGNet)".

[10] Warthman F.; Warthman Associates, "Delay-Tolerant Networks (DTNs), A tutorial", Version 1.1.

ANEXOS

Títol: Intelligent Communication Gateway for Unmanned Airborne Systems

Autor: Daniel Jiménez Verdugo
Joan Miquel Luque Oliver

Director: Juan López Rubio
Pablo Royo Chic

Data: 30 de marzo de 2009

ANEXO 1: Datasheet Ubiquity SR5

UBIQUITI NETWORKS

SUPER RANGE 5

Powerful Range and Throughput Performance for 5GHz Networks

CARD INFORMATION	
Chipset	Atheros, 4th Generation, AR5213
Radio Operation	IEEE 802.11a, 5GHz
Interface	32-bit mini-PCI Type III
Operation Voltage	3.3VDC
Antenna Ports	(1) u.fl, (1) MMCX
Temperature Range	-40C to +80C
Security	WPA, WPA2, AES-CCM & TKIP Encryption, 802.1x, 64/128/152bit WEP
Data Rates	6Mbps, 9Mbps, 12Mbps, 24Mbps, 36Mbps, 48Mbps, 54Mbps
TX Channel Width Support	5MHz / 10MHz / 20MHz / 40MHz
RoHS Compliance	YES

REGULATORY INFORMATION	
Wireless Modular Approvals	FCC Part 15.247, CE

RADIO OPERATING FREQUENCY 5.20-5.825GHz

	TX SPECIFICATIONS				RX SPECIFICATIONS			
	DataRate	Avg. Power	Tolerance			DataRate	Sensitivity	Tolerance
802.11a OFDM	6Mbps	26 dBm	+/-1.5dB		802.11a OFDM	6Mbps	-94 dBm	+/-1.5dB
	9Mbps	26 dBm	+/-1.5dB			9Mbps	-93 dBm	+/-1.5dB
	12Mbps	26 dBm	+/-1.5dB			12Mbps	-91 dBm	+/-1.5dB
	18Mbps	26 dBm	+/-1.5dB			18Mbps	-90 dBm	+/-1.5dB
	24Mbps	26 dBm	+/-1.5dB			24Mbps	-86 dBm	+/-1.5dB
	36Mbps	24 dBm	+/-1.5dB			36Mbps	-83 dBm	+/-1.5dB
	48Mbps	22 dBm	+/-1.5dB			48Mbps	-77 dBm	+/-1.5dB
	54Mbps	21 dBm	+/-1.5dB			54Mbps	-74 dBm	+/-1.5dB

ADJUSTABLE CHANNEL SIZE SUPPORT (Increase Channel Capacity or Increase Throughput)			
5MHz	10MHz	20MHz	40MHz (Turbo)

CURRENT CONSUMPTION INFORMATION

	TX CURRENT CONSUMPTION				RX CURRENT CONSUMPTION			
	DataRate	Current	Tolerance			DataRate	Current	Tolerance
802.11a OFDM	6Mbps	1.30 A	+/-100 mA		802.11a OFDM	6Mbps	350 mA	+/-100 mA
	9Mbps	1.30 A	+/-100 mA			9Mbps	350 mA	+/-100 mA
	12Mbps	1.30 A	+/-100 mA			12Mbps	350 mA	+/-100 mA
	18Mbps	1.30 A	+/-100 mA			18Mbps	350 mA	+/-100 mA
	24Mbps	1.30 A	+/-100 mA			24Mbps	350 mA	+/-100 mA
	36Mbps	1.00 A	+/-100 mA			36Mbps	350 mA	+/-100 mA
	48Mbps	0.90 A	+/-100 mA			48Mbps	350 mA	+/-100 mA
	54Mbps	0.80 A	+/-100 mA			54Mbps	350 mA	+/-100 mA

RANGE PERFORMANCE	
Indoor (Antenna Dependent):	Up to 150meters
Outdoor (Antenna Dependent):	Over 50km

DRIVER INFORMATION	
Operating System Support	Linux MADWIFI, WindowsXP, Windows2000
Advanced Mobility / QuickHandoff	WindowsXP/2000 Utility with Enhanced Mobility Driver from Ubiquiti
Cisco Support	CCX 4.0 Supported Driver/Utility also available from Ubiquiti

For help with MADWIFI or other Special Driver Support, Please e-mail support@ubnt.com

UBIQUITI NETWORKS

495-499 Montague Expwy. Milpitas, CA 95035
San Jose, CA 95112
T (408)-942-3085 F (408)-351-4973
http://www.ubnt.com

ANEXO 2: Datasheet 24XStream RF Modules

XStream™ OEM RF Modules

900 MHz - 2.4 GHz - Long Range OEM RF Modules by MaxStream, Inc.

Long Range Performance

Indoor/urban Range:	up to 1500' (450 m)
Outdoor line-of-sight Range:	up to 7 miles (11 km) w/ 2.1 dB dipole antenna
Outdoor line-of-sight Range:	up to 20 miles (32 km) w/ high-gain antenna
Receiver Sensitivity:	-110 dBm (@9600 bps)

RS-232/485, USB, Ethernet & Telephone interface packages available

Advanced Networking & Security

7 Frequency Hopping Channels - each with 65K available

Retries & Acknowledgements for reliable packet delivery

Several advanced networking modes supported

Easy-to-Use

No configuration is necessary for out-of-box RF operation. Simply feed data into one module, then the data is sent out the other end of the wireless link.

If more advanced functionality is needed, the modules support an extensive set of AT and binary commands.

"Instant Gratification...
The radios worked perfectly together right out of the box."
- Fred Eady
Circuit Cellar "Radio Roundup"
In describing the out-of-box experience
he gathered from the MaxStream Development Kit.

Key Features

Price-to-Performance Value.
Due to innovations stamped in its design, the XStream Module yields 2-8x the range of competing modules. This allows OEMs and integrators to cover more ground with fewer devices.

Additionally, XStream Modules are easy-to-use and therefore greatly reduce the cost of data system development.

Receiver Sensitivity.
MaxStream modules 'hear' what others cannot; therefore supplying greater range and reliability in wireless links.

For every 6 dB gained in TX power or RX sensitivity, OEMs and integrators can double the range of a wireless link. XStream Modules outperform higher costing modules due in large part to range gained through superior RX sensitivity.

Low Power Consumption.
For power-sensitive applications; Pin, Serial Port and Cyclic Sleep Modes are available. Power-down currents can reach below 26 µA.

Worldwide Acceptance.
FCC (U.S.A.), IC (Canada), ETSI (Europe)

Systems that include XStream Modules inherit MaxStream's certifications. Contact MaxStream for a complete list of government agency approvals.

Sample Applications


Monitoring of remote systems


Sensor data capture in embedded systems


Home automation & building control


SCADA (Supervisory control & data acquisition)


Fleet management & asset tracking

Call today!
- Free RF Consultation
- Volume Discounts
- Development Kit Pricing

MaxStream.
(866) 765-9885 toll-free in U.S. & Canada
(801) 765-9885 worldwide
www.maxstream.net

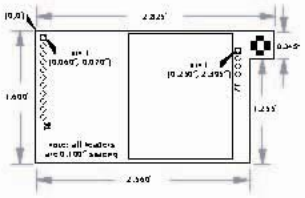
XStream™ 900 MHz & 2.4 GHz OEM RF Modules

Specifications		9XStream™ (900 MHz)		24XStream™ (2.4 GHz)	
Performance	Indoor/Urban Range (w/ 2.1 dB dipole antenna)	up to 1500 ft. (450 m)		up to 600 ft. (180 m)	
	Outdoor RF line-of-sight Range (w/ 2.1 dB dipole antenna)	up to 7 miles (11 km)		up to 3 miles (5 km)	
	Outdoor RF line-of-sight Range (w/ high-gain antenna)	up to 20 miles (32 km)		up to 10 miles (16 km)	
	Transmit Power Output	100 mW (20 dBm)		50 mW (17 dBm)	
	Interface Data Rate (software selectable)	10 - 57600 bps (including non-standard baud rates)		10 - 57600 bps (including non-standard baud rates)	
	Throughput Data Rate	9,600 bps	19,200 bps	9,600 bps	19,200 bps
	RF Data Rate	10,000 bps	20,000 bps	10,000 bps	20,000 bps
	Receiver Sensitivity	-110 dBm	-107 dBm	-105 dBm	-102 dBm
Power Requirements	Supply Voltage	5V (± 0.25V) regulated		5V (± 0.25V) regulated	
	Receive Current	50 mA		80 mA	
	Transmit Current	140 mA		150 mA	
	Power-down Current	< 26 µA		< 26 µA	
General	Dimensions	1.600" x 2.825" x 0.350" (4.06 cm x 7.18 cm x 0.89 cm)		1.600" x 2.825" x 0.350" (4.06 cm x 7.18 cm x 0.89 cm)	
	Weight	0.8 oz. (24 g)		0.8 oz. (24 g)	
	Operating Temperature	0 - 70 C° (commercial) or -40 to 85° C (industrial)		0 - 70 C° (commercial) or -40 to 85° C (industrial)	
	Antenna Options	RPSMA, MMCX, or Wire Antenna		RPSMA, MMCX, or Wire Antenna	
Networking and Security	Operating Frequency	ISM 902 - 928 MHz		ISM 2.4000 - 2.4835 GHz	
	Supported Network Topologies	Peer-to-Peer (no master/slave dependencies), Point-to-Point, Point-to-Multipoint, Multidrop		Peer-to-Peer (no master/slave dependencies), Point-to-Point, Point-to-Multipoint, Multidropp	
	Number of Channels (software selectable)	7 Frequency Hopping Channels		7 Frequency Hopping Channels	
	Network Filtration Layers	VID, Channel, Destination Address		VID, Channel, Destination Address	
Agency Approvals	FCC Part 15.247	OUR9XSTREAM		OUR-24XSTREAM	
	Industry Canada (IC)	4214A-9XSTREAM		4214A12008	
	Europe	n/a		ETSI	

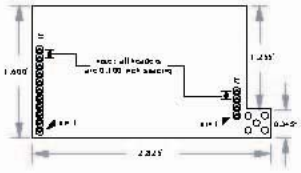
Specifications are subject to change without notice.

Mechanical Drawings

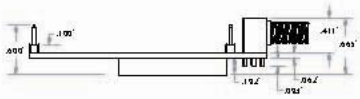
TOP VIEW



BOTTOM VIEW (pins protruding)



SIDE VIEW (with RPSMA Connector)

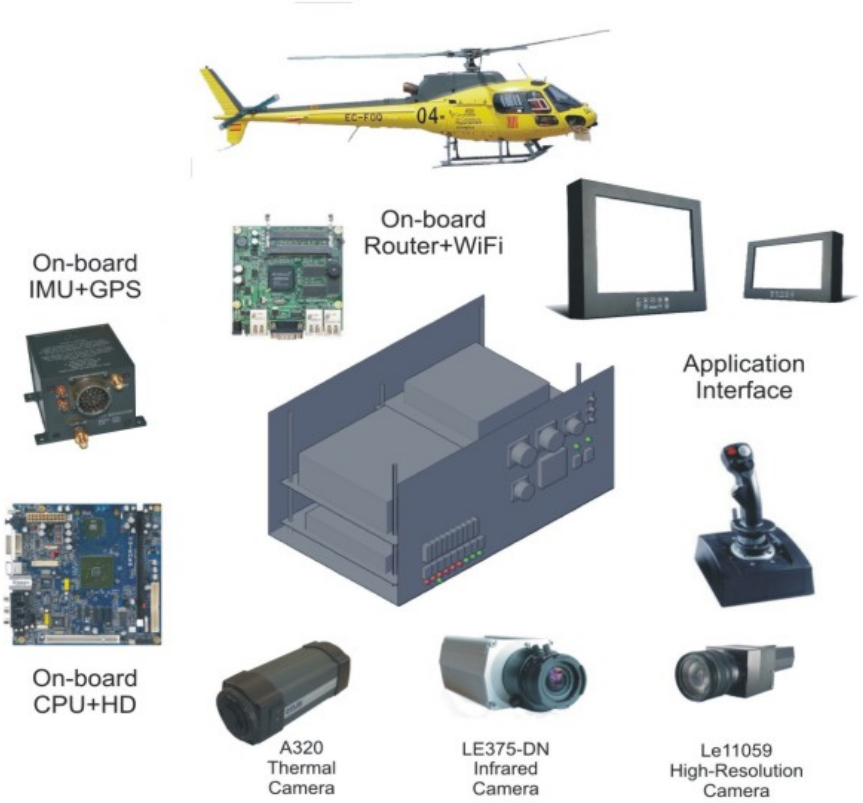


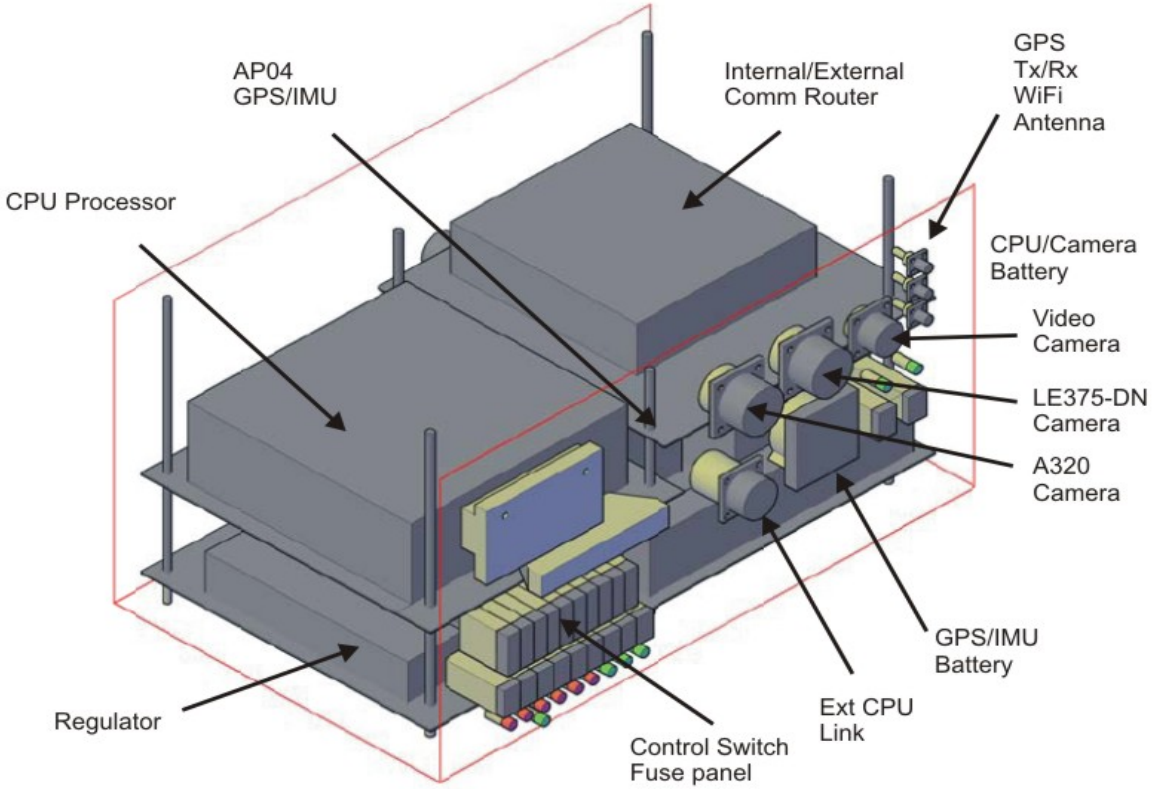
MaxStream.
355 South, 520 West, ste. 180
Lindon, UT 84042

For the best in wireless data solutions and support, contact MaxStream, Inc.

phone: (866) 765-9885 (toll-free in U.S. & Canada)
(801) 765-9885 (worldwide)
fax: (801) 765-9895
web: www.maxstream.net
(live chat & many other resources available)

ANEXO 3: Arquitectura Interna UAV





ANEXO 4: Diagrama UML de clases de MAREA

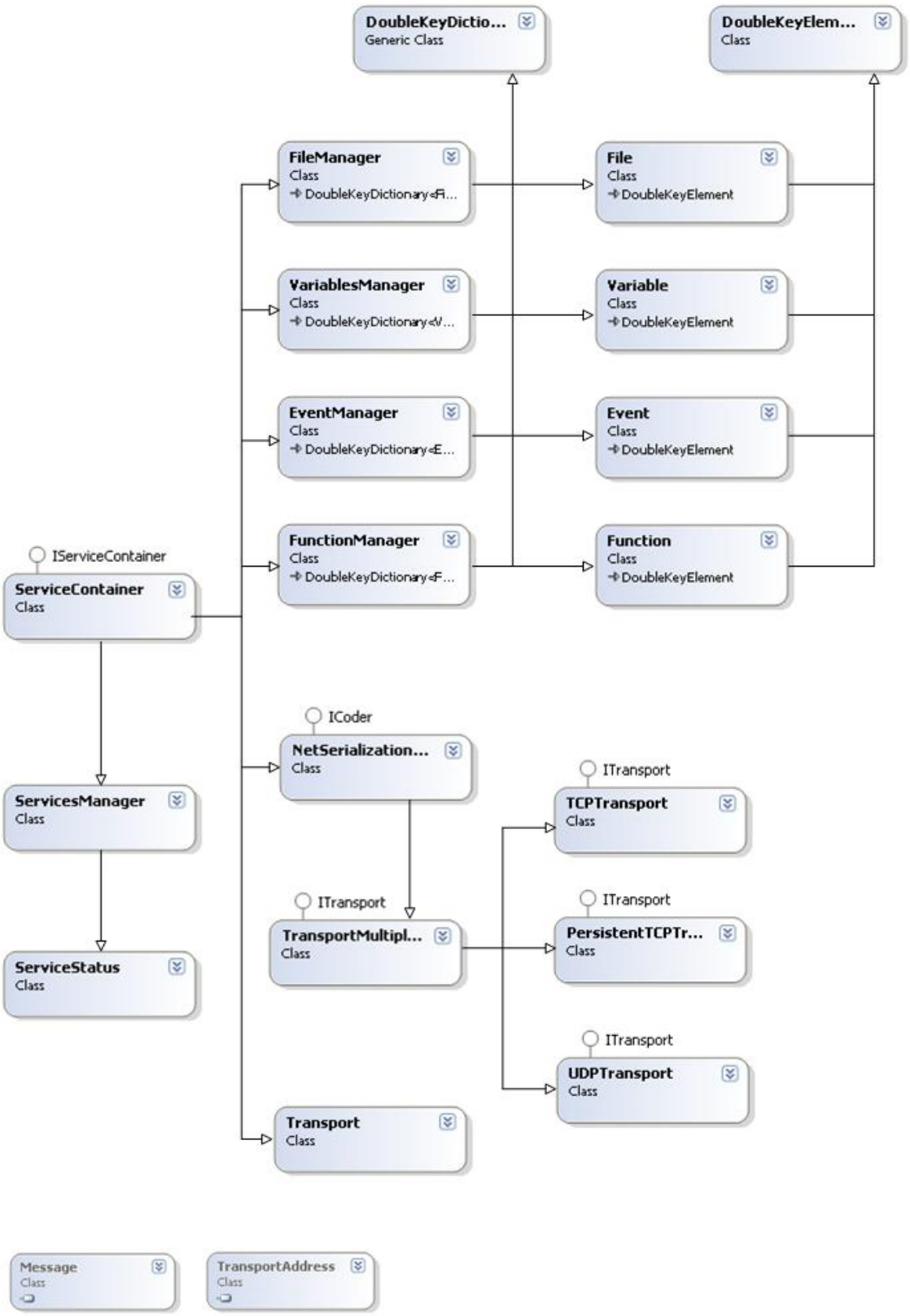


Fig. 1 Diagrama de clases MAREA

ANEXO 5: Diagrama UML de clases de ICGUAS

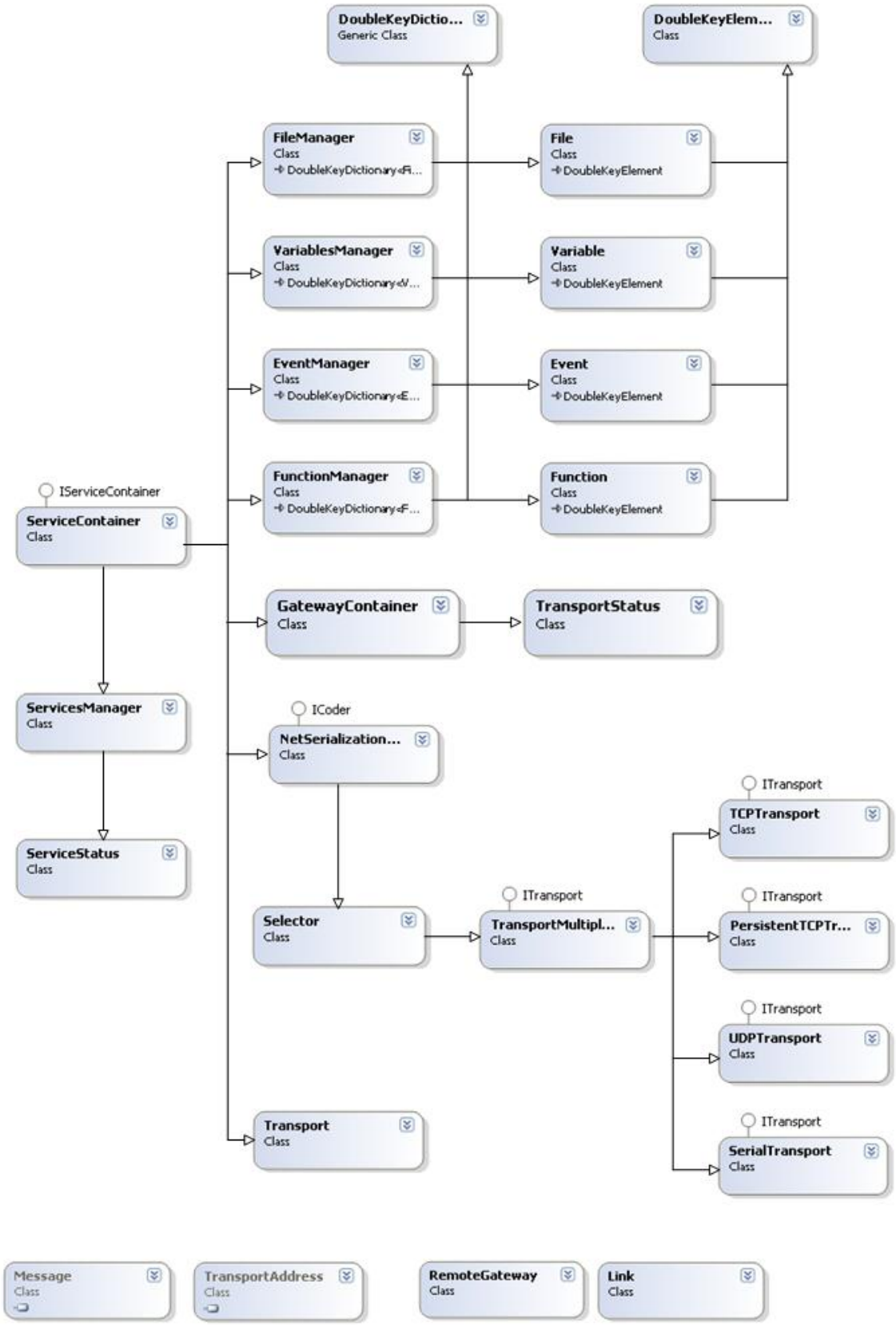


Fig. 1 Diagrama UML

ANEXO 6: Tabla normalizada CRC

```
0x00000000,0x77073096,0xEE0E612C,0x990951BA,
0x076DC419,0x706AF48F,0xE963A535,0x9E6495A3,
0x0EDB8832,0x79DCB8A4,0xE0D5E91E,0x97D2D988,
0x09B64C2B,0x7EB17CBD,0xE7B82D07,0x90BF1D91,
0x1DB71064,0x6AB020F2,0xF3B97148,0x84BE41DE,
0x1ADAD47D,0x6DDDE4EB,0xF4D4B551,0x83D385C7,
0x136C9856,0x646BA8C0,0xFD62F97A,0x8A65C9EC,
0x14015C4F,0x63066CD9,0xFA0F3D63,0x8D080DF5,
0x3B6E20C8,0x4C69105E,0xD56041E4,0xA2677172,
0x3C03E4D1,0x4B04D447,0xD20D85FD,0xA50AB56B,
0x35B5A8FA,0x42B2986C,0xDBBBC9D6,0xACBCF940,
0x32D86CE3,0x45DF5C75,0xDCD60DCF,0xABD13D59,
0x26D930AC,0x51DE003A,0xC8D75180,0xBFD06116,
0x21B4F4B5,0x56B3C423,0xCFBA9599,0xB8BDA50F,
0x2802B89E,0x5F058808,0xC60CD9B2,0xB10BE924,
0x2F6F7C87,0x58684C11,0xC1611DAB,0xB6662D3D,
0x76DC4190,0x01DB7106,0x98D220BC,0xEFD5102A,
0x71B18589,0x06B6B51F,0x9FBFE4A5,0xE8B8D433,

0x7807C9A2,0x0F00F934,0x9609A88E,0xE10E9818,
0x7F6A0DBB,0x086D3D2D,0x91646C97,0xE6635C01,
0x6B6B51F4,0x1C6C6162,0x856530D8,0xF262004E,
0x6C0695ED,0x1B01A57B,0x8208F4C1,0xF50FC457,
0x65B0D9C6,0x12B7E950,0x8BBEB8EA,0xFCB9887C,
0x62DD1DDF,0x15DA2D49,0x8CD37CF3,0xFBD44C65,
0x4DB26158,0x3AB551CE,0xA3BC0074,0xD4BB30E2,
0x4ADFA541,0x3DD895D7,0xA4D1C46D,0xD3D6F4FB,
0x4369E96A,0x346ED9FC,0xAD678846,0xDA60B8D0,
0x44042D73,0x33031DE5,0xAA0A4C5F,0xDD0D7CC9,
0x5005713C,0x270241AA,0xBE0B1010,0xC90C2086,
0x5768B525,0x206F85B3,0xB966D409,0xCE61E49F,
0x5EDEF90E,0x29D9C998,0xB0D09822,0xC7D7A8B4,
0x59B33D17,0x2EB40D81,0xB7BD5C3B,0xC0BA6CAD,
0xEDB88320,0x9ABFB3B6,0x03B6E20C,0x74B1D29A,
0xEAD54739,0x9DD277AF,0x04DB2615,0x73DC1683,
0xE3630B12,0x94643B84,0x0D6D6A3E,0x7A6A5AA8,
0xE40ECF0B,0x9309FF9D,0x0A00AE27,0x7D079EB1,
0xF00F9344,0x8708A3D2,0x1E01F268,0x6906C2FE,
0xF762575D,0x806567CB,0x196C3671,0x6E6B06E7,
0xFED41B76,0x89D32BE0,0x10DA7A5A,0x67DD4ACC,
0xF9B9DF6F,0x8EBEEFF9,0x17B7BE43,0x60B08ED5,
0xD6D6A3E8,0xA1D1937E,0x38D8C2C4,0x4FDFF252,
0xD1BB67F1,0xA6BC5767,0x3FB506DD,0x48B2364B,
0xD80D2BDA,0xAF0A1B4C,0x36034AF6,0x41047A60,
0xDF60EFC3,0xA867DF55,0x316E8EEF,0x4669BE79,
0xCB61B38C,0xBC66831A,0x256FD2A0,0x5268E236,
0xCC0C7795,0xBB0B4703,0x220216B9,0x5505262F,
0xC5BA3BBE,0xB2BD0B28,0x2BB45A92,0x5CB36A04,
0xC2D7FFA7,0xB5D0CF31,0x2CD99E8B,0x5BDEAE1D,
0x9B64C2B0,0xEC63F226,0x756AA39C,0x026D930A,
0x9C0906A9,0xEB0E363F,0x72076785,0x05005713,
0x95BF4A82,0xE2B87A14,0x7BB12BAE,0x0CB61B38,
0x92D28E9B,0xE5D5BE0D,0x7CDCEFB7,0x0BDBDF21,
0x86D3D2D4,0xF1D4E242,0x68DDB3F8,0x1FDA836E,
0x81BE16CD,0xF6B9265B,0x6FB077E1,0x18B74777,
0x88085AE6,0xFF0F6A70,0x66063BCA,0x11010B5C,
0x8F659EFF,0xF862AE69,0x616BFFD3,0x166CCF45,
```

```
0xA00AE278,0xD70DD2EE,0x4E048354,0x3903B3C2,
0xA7672661,0xD06016F7,0x4969474D,0x3E6E77DB,
0xAED16A4A,0xD9D65ADC,0x40DF0B66,0x37D83BF0,
0xA9BCAE53,0xDEBB9EC5,0x47B2CF7F,0x30B5FFE9,
0xBDBDF21C,0xCABAC28A,0x53B39330,0x24B4A3A6,
0xBAD03605,0xCDD70693,0x54DE5729,0x23D967BF,
0xB3667A2E,0xC4614AB8,0x5D681B02,0x2A6F2B94,
0xB40BBE37,0xC30C8EA1,0x5A05DF1B,0x2D02EF8D;
```

ANEXO 7: Capturas pruebas funcionamiento

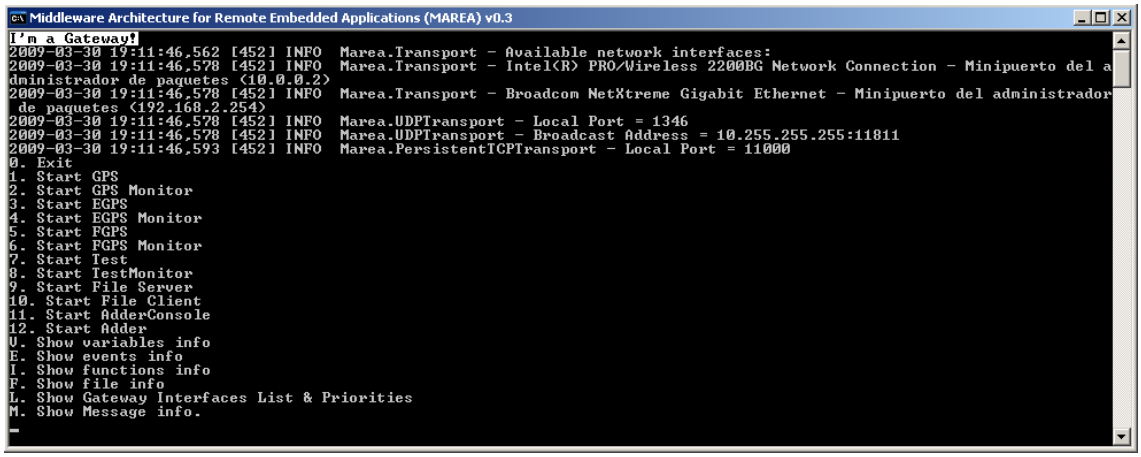


Fig .1

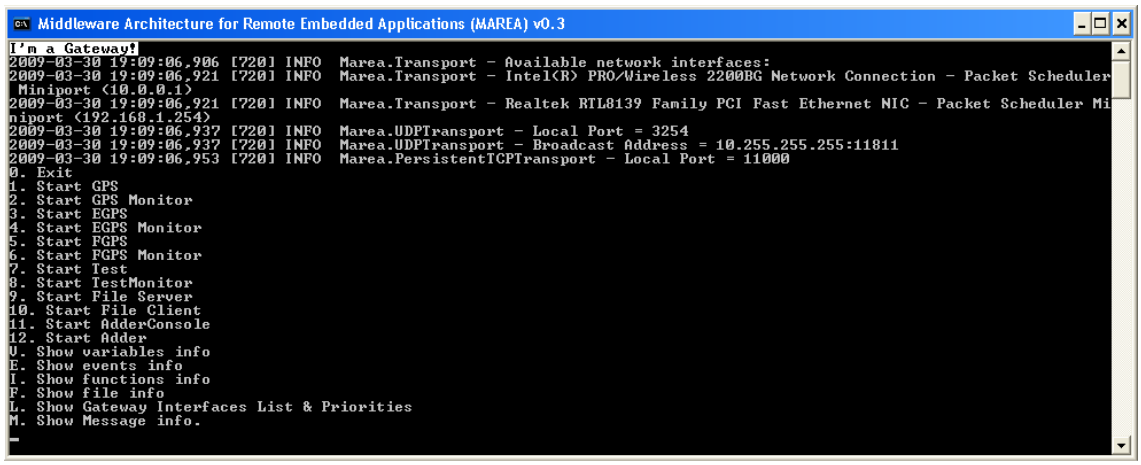


Fig. 2

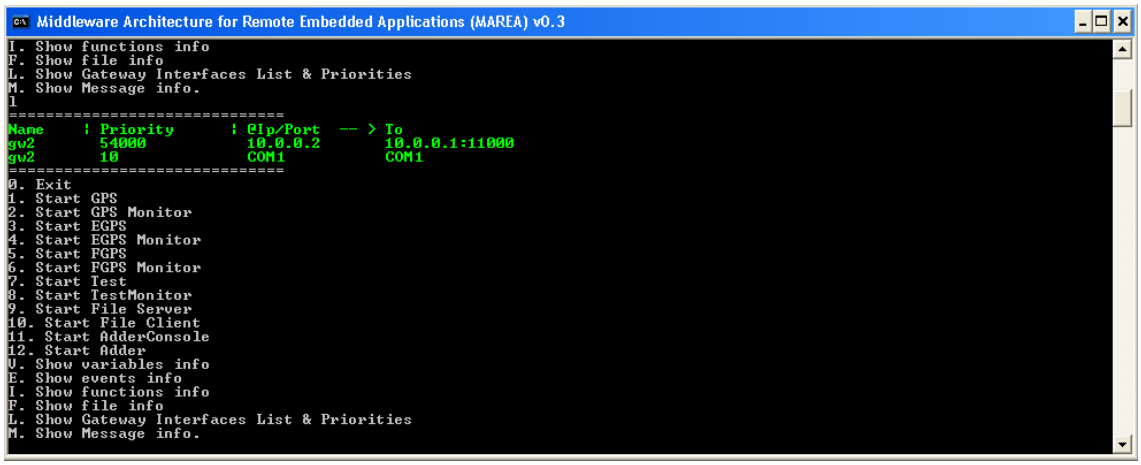


Fig. 3

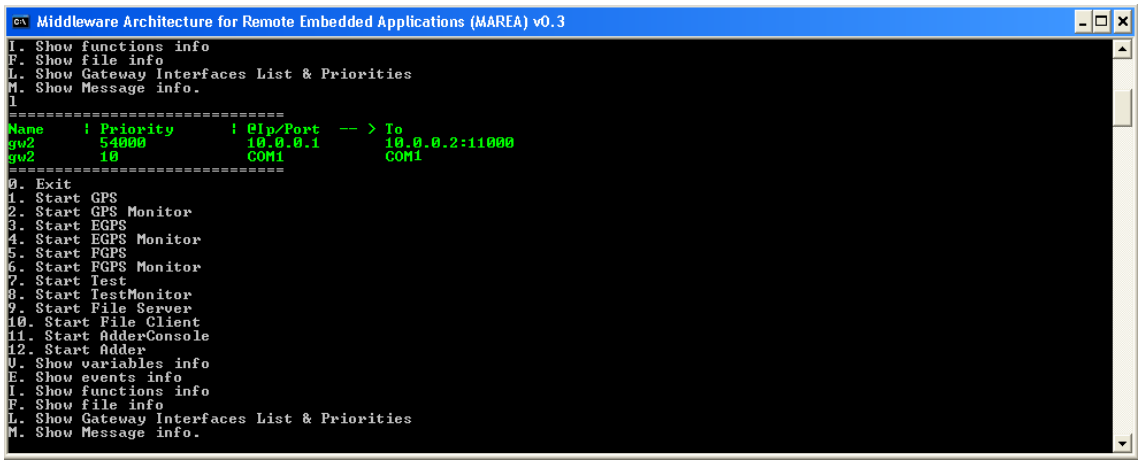


Fig. 4

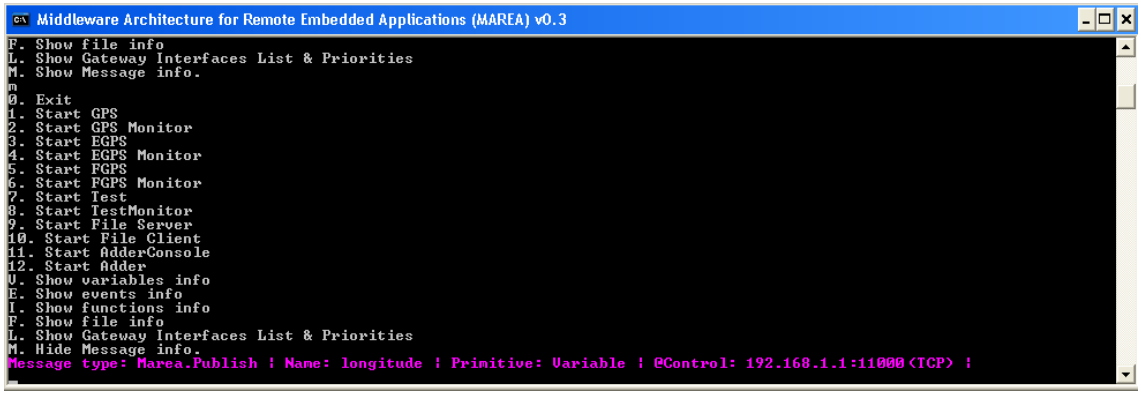


Fig. 5

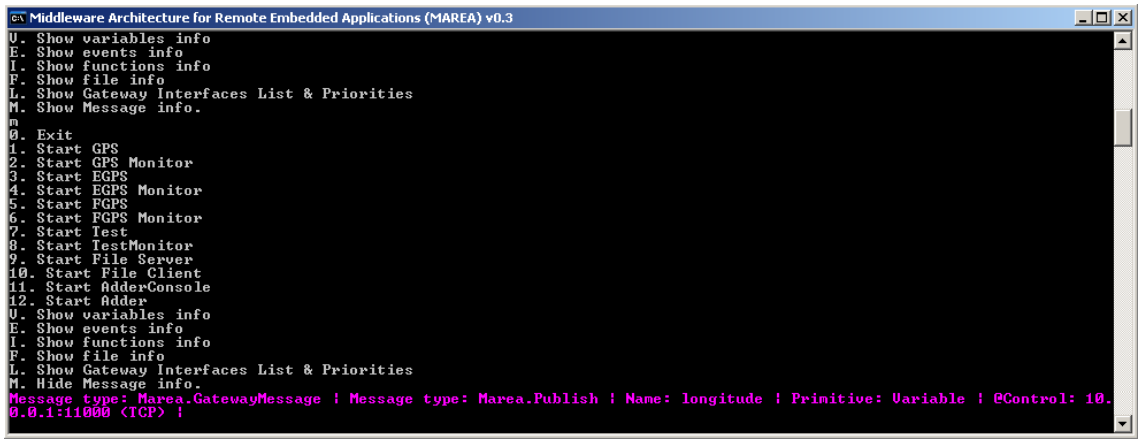


Fig. 6

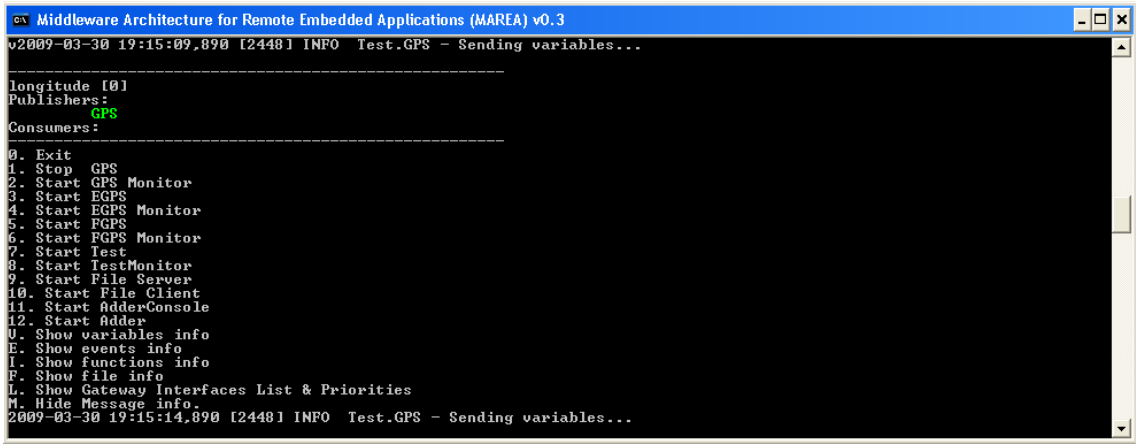


Fig. 7

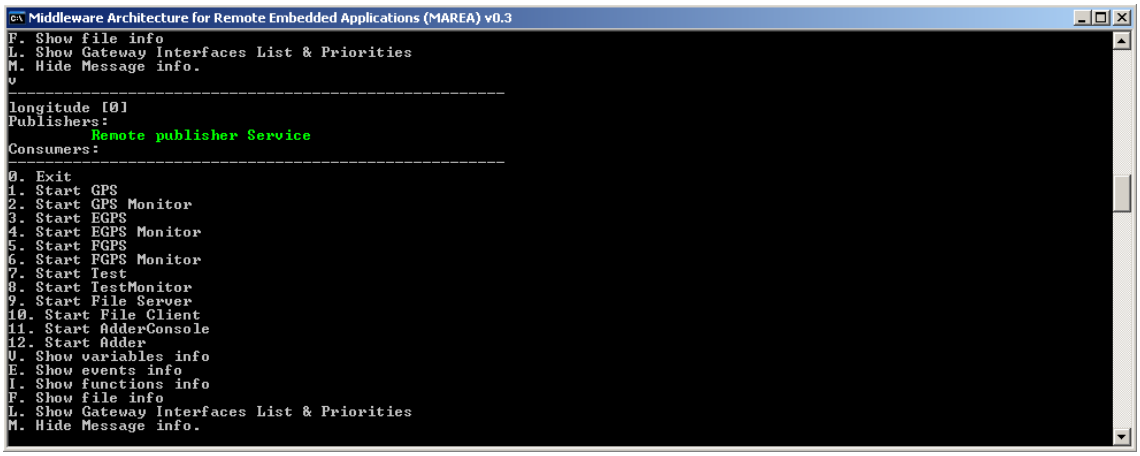


Fig. 8

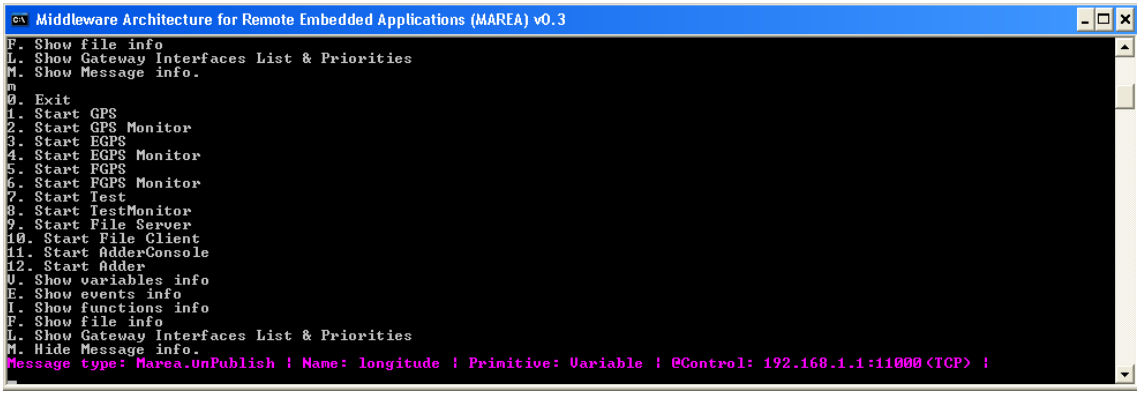


Fig. 9

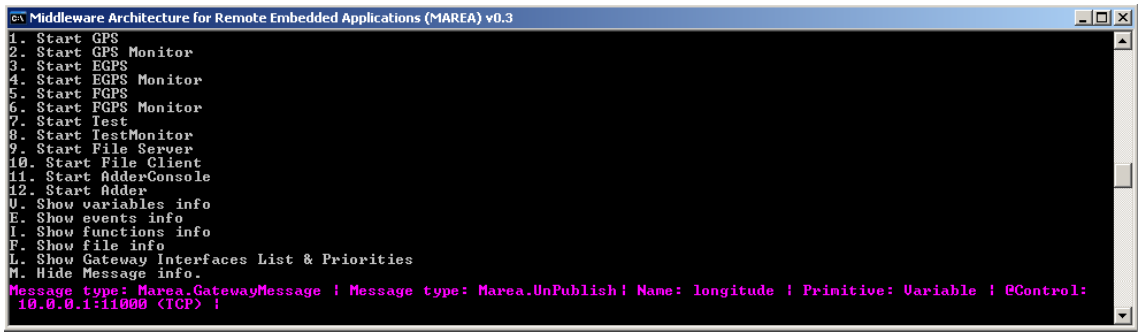


Fig. 10

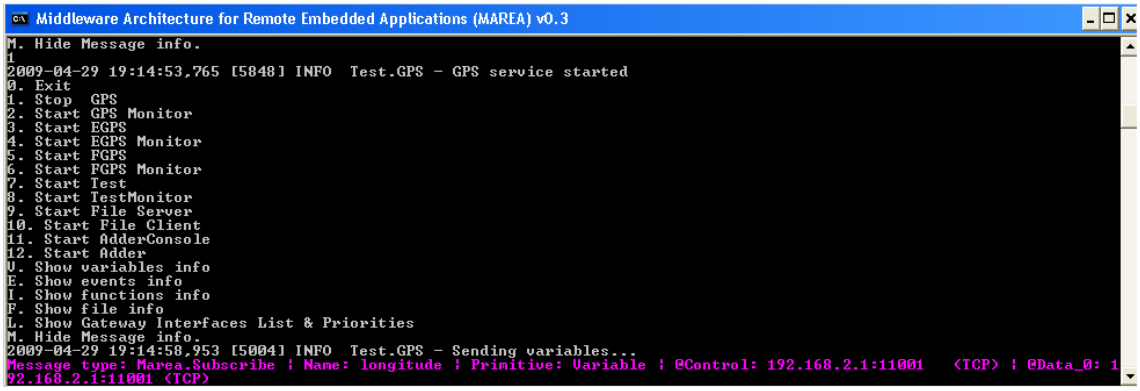


Fig. 11

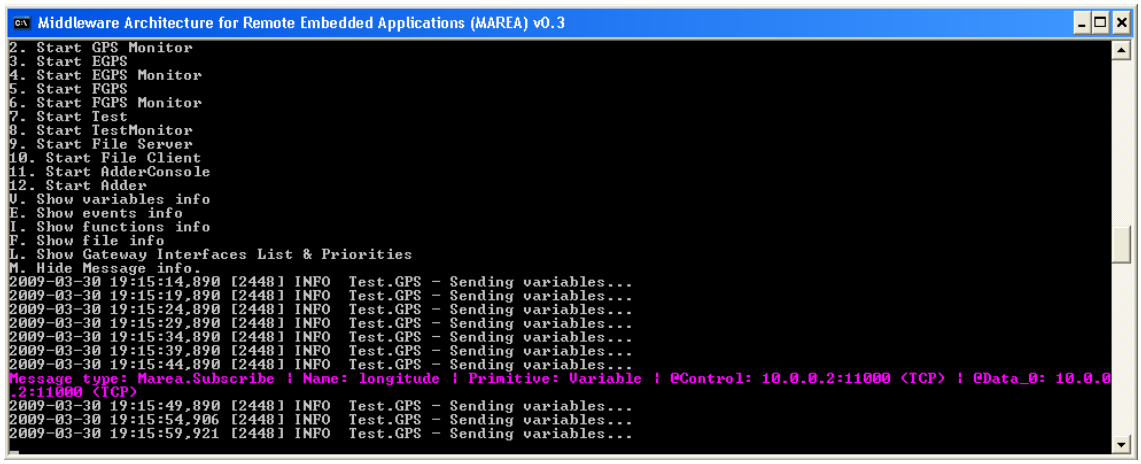


Fig. 12

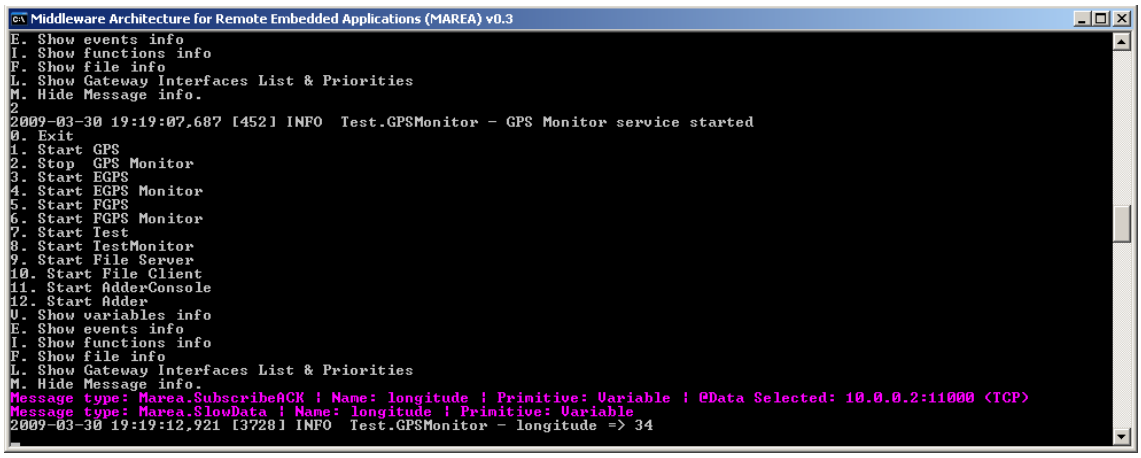


Fig. 13

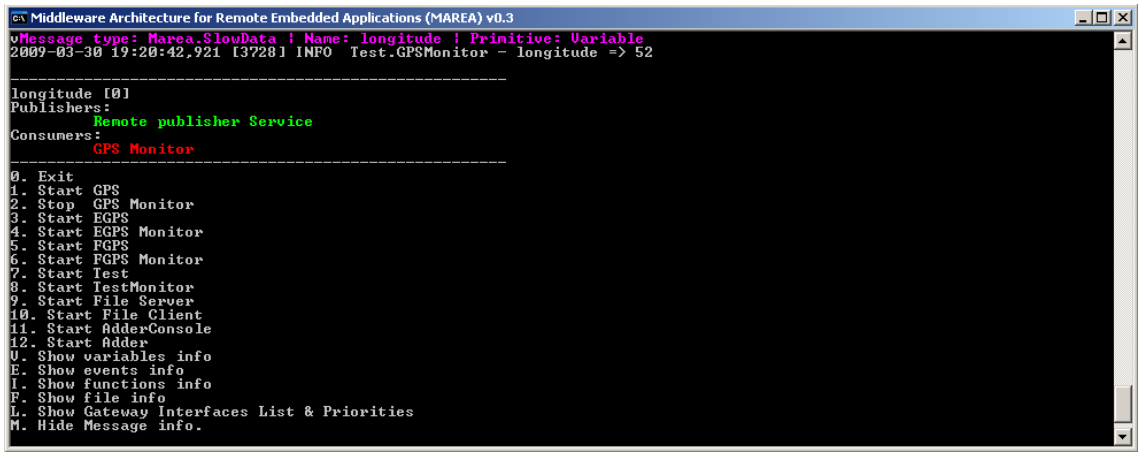


Fig. 14

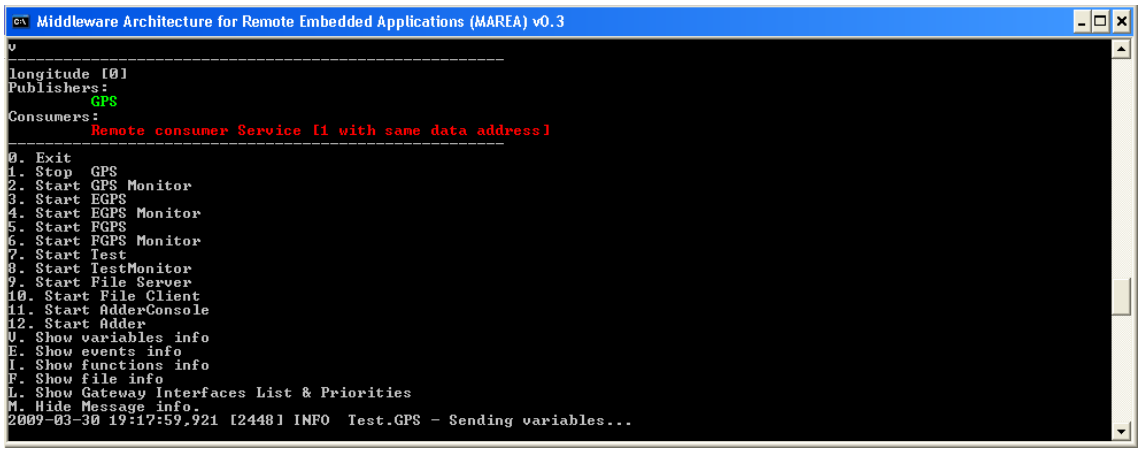


Fig. 15

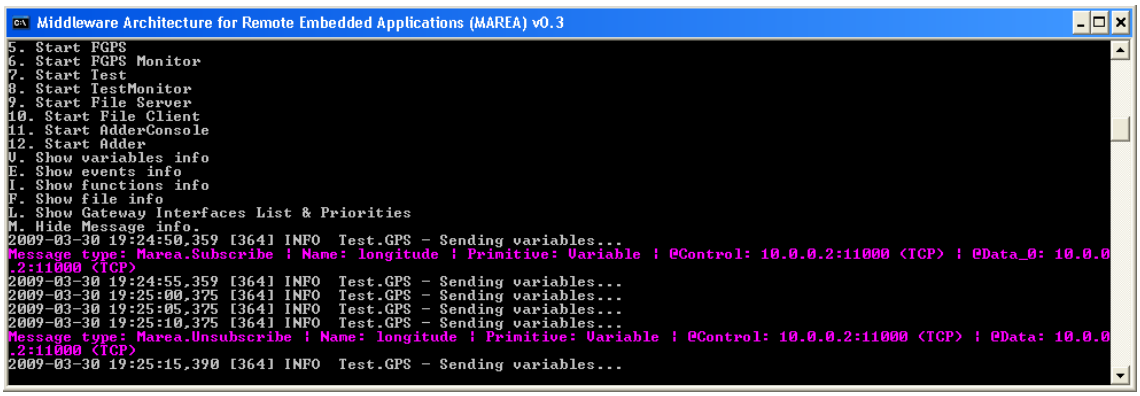


Fig. 16

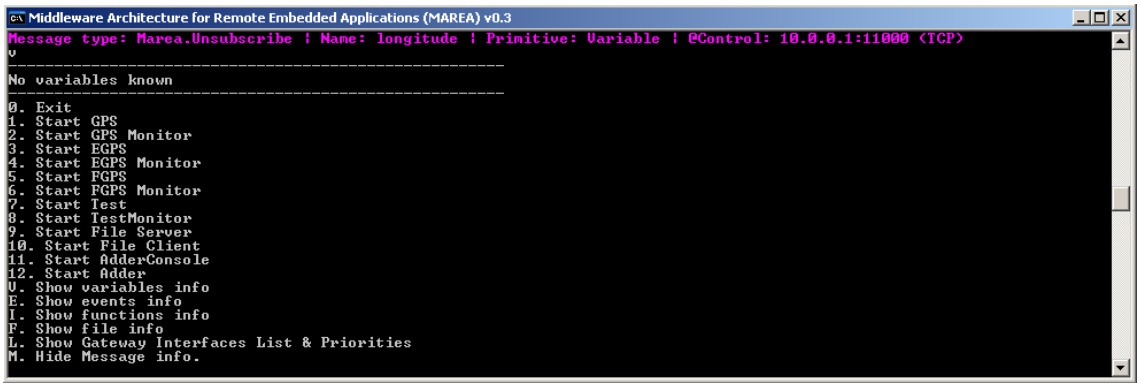


Fig. 17

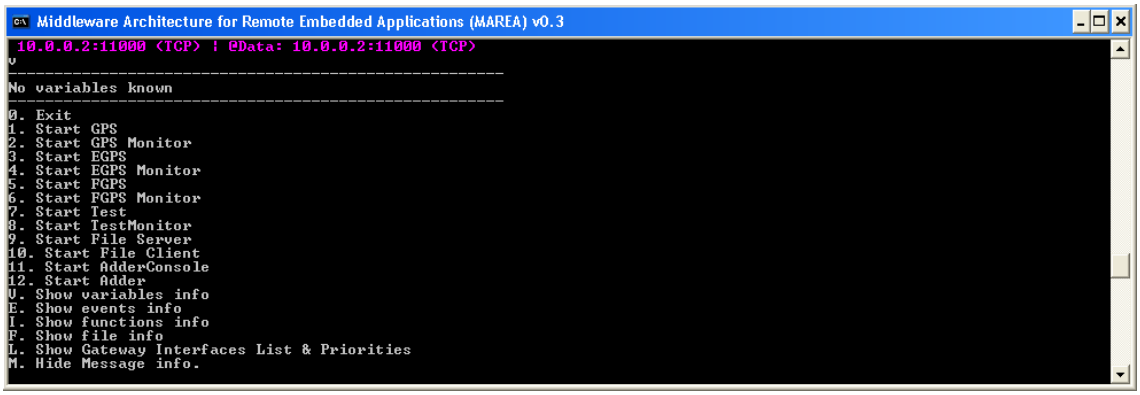


Fig. 18

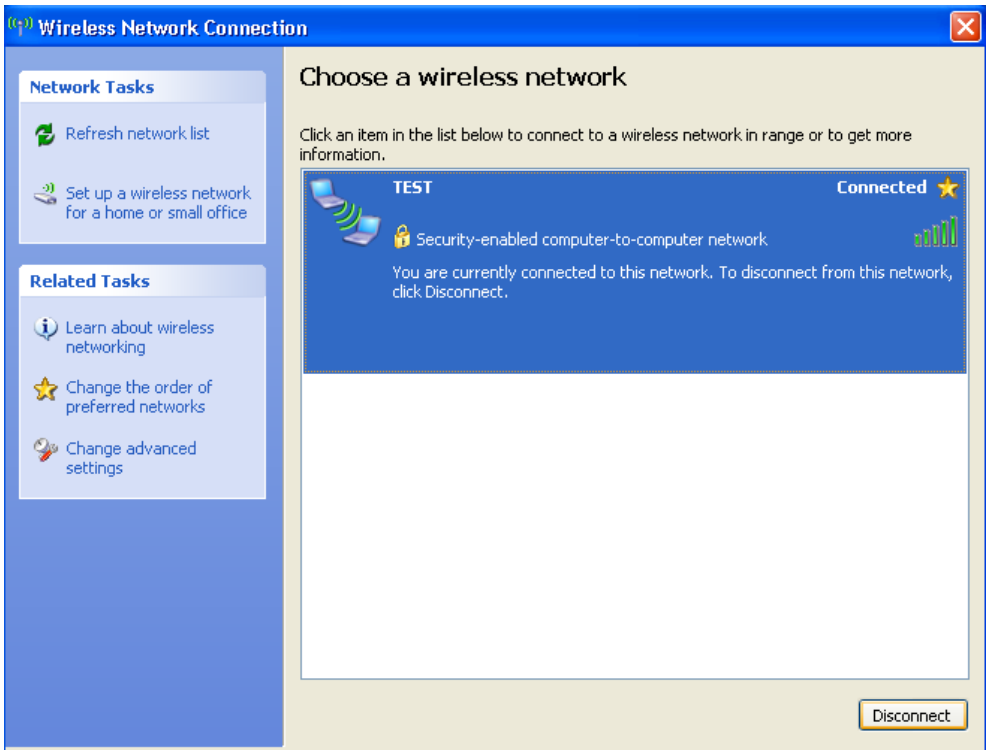

Wireless Network Connection
Network Tasks
Refresh network list
Set up a wireless network for a home or small office
Related Tasks
Learn about wireless networking
Change the order of preferred networks
Change advanced settings
Choose a wireless network
Click an item in the list below to connect to a wireless network in range or to get more information.
TEST
Connected
Security-enabled computer-to-computer network
You are currently connected to this network. To disconnect from this network, click Disconnect.
Disconnect

Fig. 19

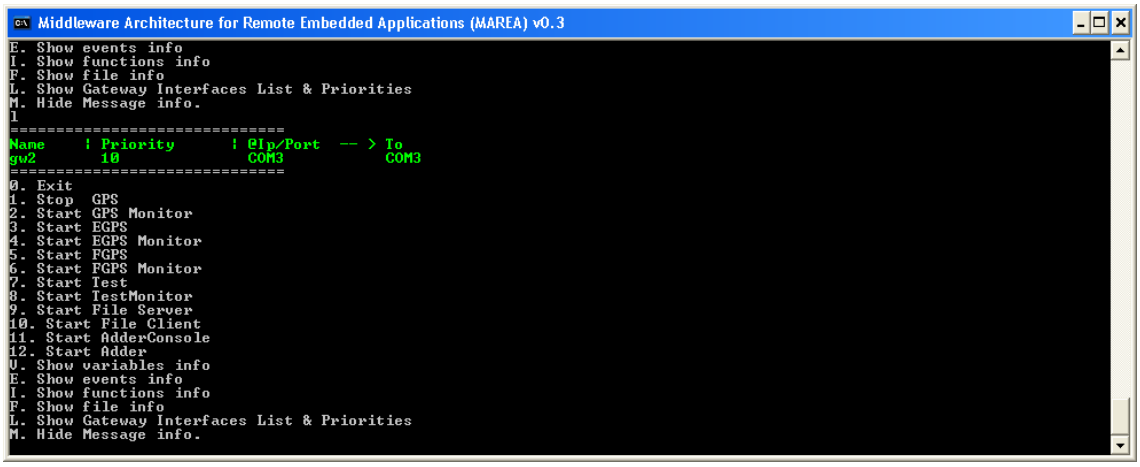

Middleware Architecture for Remote Embedded Applications (MAREA) v0.3
E. Show events info
I. Show functions info
F. Show file info
L. Show Gateway Interfaces List & Priorities
M. Hide Message info.
1
Name ¦ Priority ¦ @Ip/Port -- > To
gw2 10 COM3 COM3
0. Exit
1. Stop GPS
2. Start GPS Monitor
3. Start EGPS
4. Start EGPS Monitor
5. Start FGPS
6. Start FGPS Monitor
7. Start Test
8. Start TestMonitor
9. Start File Server
10. Start File Client
11. Start AdderConsole
12. Start Adder
V. Show variables info
E. Show events info
I. Show functions info
F. Show file info
L. Show Gateway Interfaces List & Priorities
M. Hide Message info.

Fig. 20

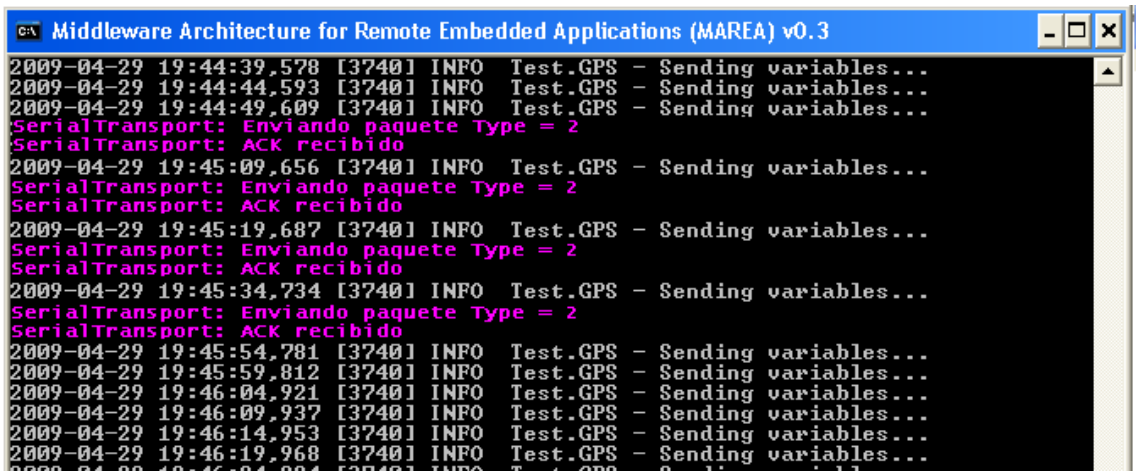


Fig. 21

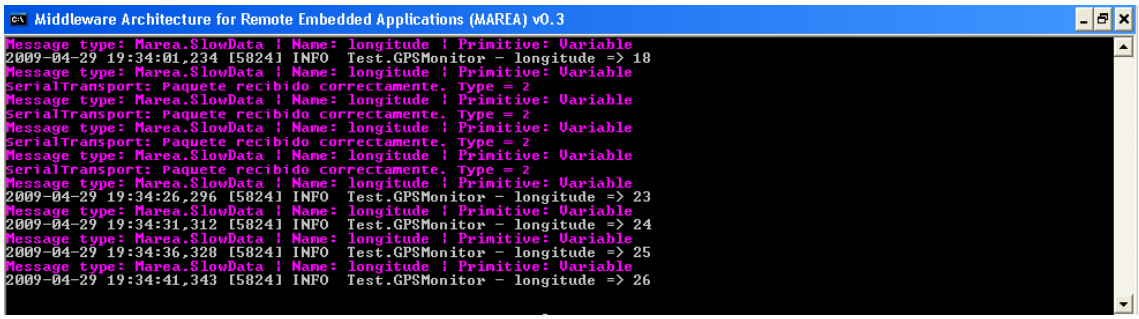


Fig. 22

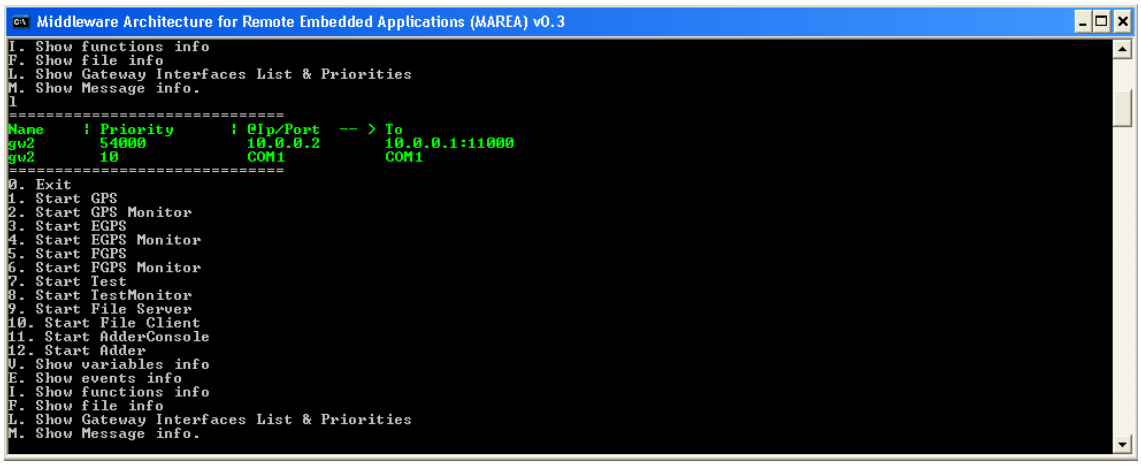


Fig. 23

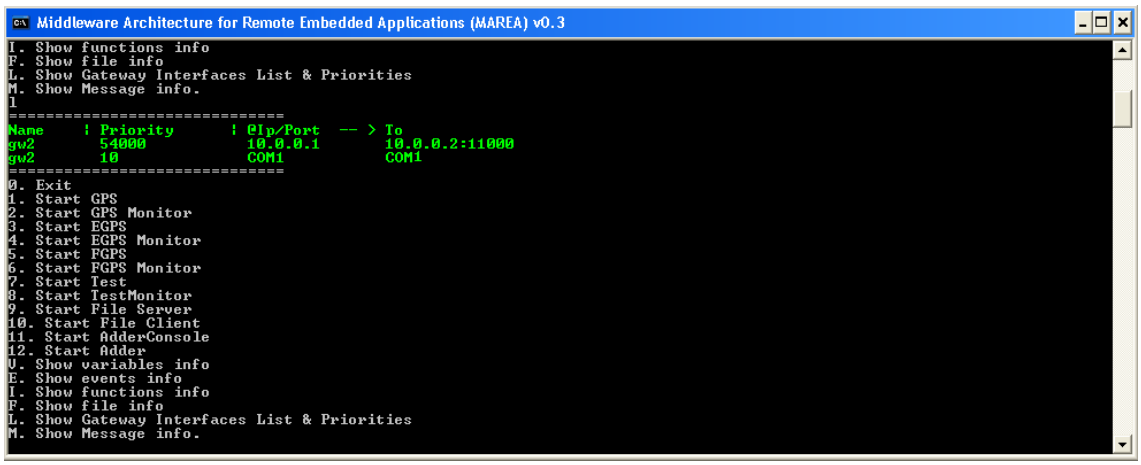

Middleware Architecture for Remote Embedded Applications (MAREA) v0.3
I. Show functions info
F. Show file info
L. Show Gateway Interfaces List & Priorities
M. Show Message info.
l
Name ¦ Priority ¦ @Ip/Port -- > To
gw2 54000 10.0.0.1 10.0.0.2:11000
gw2 10 COM1 COM1
0. Exit
1. Start GPS
2. Start GPS Monitor
3. Start EGPS
4. Start EGPS Monitor
5. Start FGPS
6. Start FGPS Monitor
7. Start Test
8. Start TestMonitor
9. Start File Server
10. Start File Client
11. Start AdderConsole
12. Start Adder
V. Show variables info
E. Show events info
I. Show functions info
F. Show file info
L. Show Gateway Interfaces List & Priorities
M. Show Message info.

Fig .24

ANEXO 8: Test control

Eventos

Tabla 1 Tabla resultados Eventos

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	15,57698	12,78513	11,99257	11,85678	11,05879	985,7535	975,4102	942,5365	961,4783	901,2502
712	15,24576	12,35875	12,00548	11,92458	11,24589	965,3146	923,2156	957,5456	955,2515	910,5684
912	14,91454	13,19616	12,01838	11,99238	11,43299	975,2195	952,141	972,5547	949,0247	919,8866
1112	10,25789	10,49811	11,15618	10,09916	10,48115	924,4368	921,1616	932,1187	942,7978	929,2048
1312	10,25998	10,49924	10,15619	10,15649	10,4922	903,9979	1006,151	915,6156	936,571	915,1946
1512	10,26479	10,5052	12,15617	10,22489	10,58749	975,1963	974,0212	955,6775	930,3442	916,7367
1712	10,26778	10,50794	10,51076	10,28591	10,62662	869,1888	978,6106	947,4724	924,1174	914,3907
1912	10,27123	10,51148	10,67736	10,34877	10,67978	848,7499	983,2492	939,2673	917,8906	912,0447

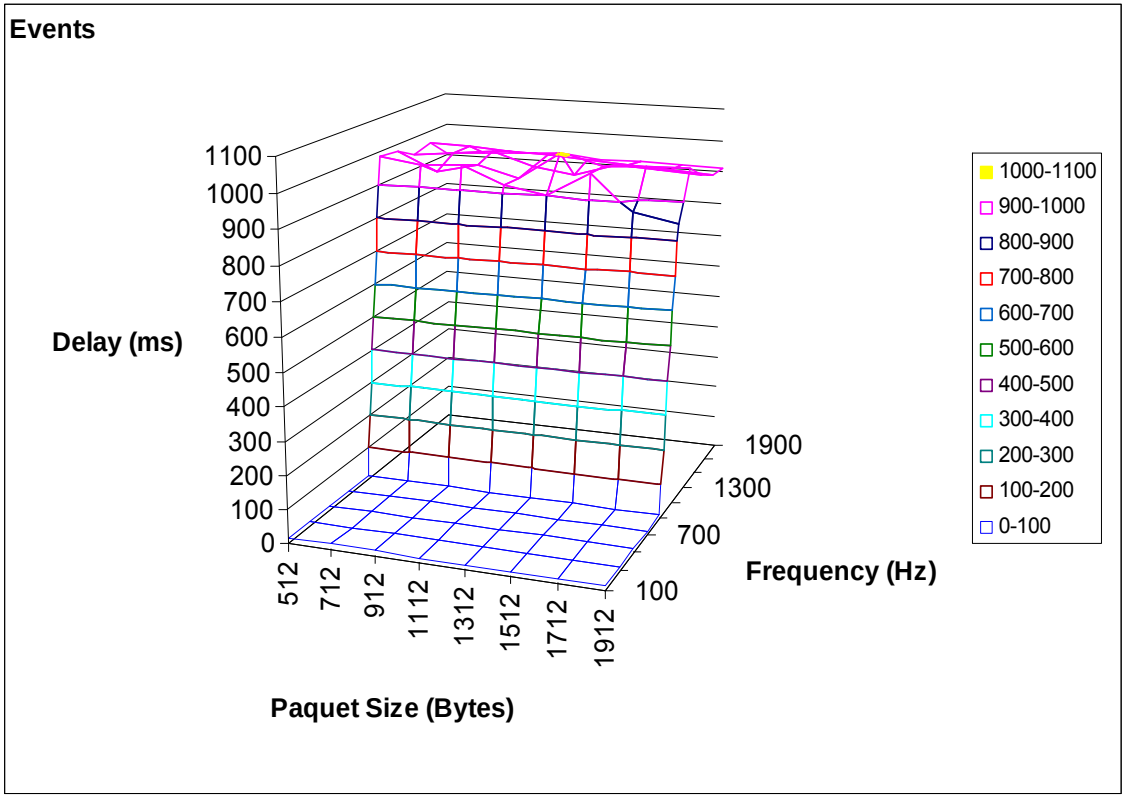


Fig. 1 Gráfico resultados Eventos

Variables

Tabla 2 Tabla resultados Variables

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	9,561891	9,21919	9,594616	9,491291	9,507653	15,18926	23,19416	16,19192	16,85944	17,36077
712	9,156163	9,611122	10,06608	9,161655	9,616614	16,21916	15,15159	15,19442	16,49416	17,01949
912	9,118689	10,19462	9,319209	9,419469	9,519729	18,14319	15,19491	15,99419	16,79347	17,59275
1112	4,356489	4,298489	4,240489	4,182488	4,124488	7,189018	7,992119	8,119419	8,246719	8,198412
1312	4,518614	4,054615	4,126256	4,406189	4,589142	8,060914	8,00191	8,181056	8,194189	8,294189
1512	4,151587	4,156489	4,161391	4,166293	4,171195	7,991115	8,014106	8,051895	8,089683	8,127471
1712	5,061106	4,515612	5,16161	4,30845	4,20487	7,891115	7,991101	10,16946	8,241189	8,118941
1912	4,354516	4,240111	4,125706	4,011302	4,545615	8,005189	8,056187	8,021849	8,184911	9,12929

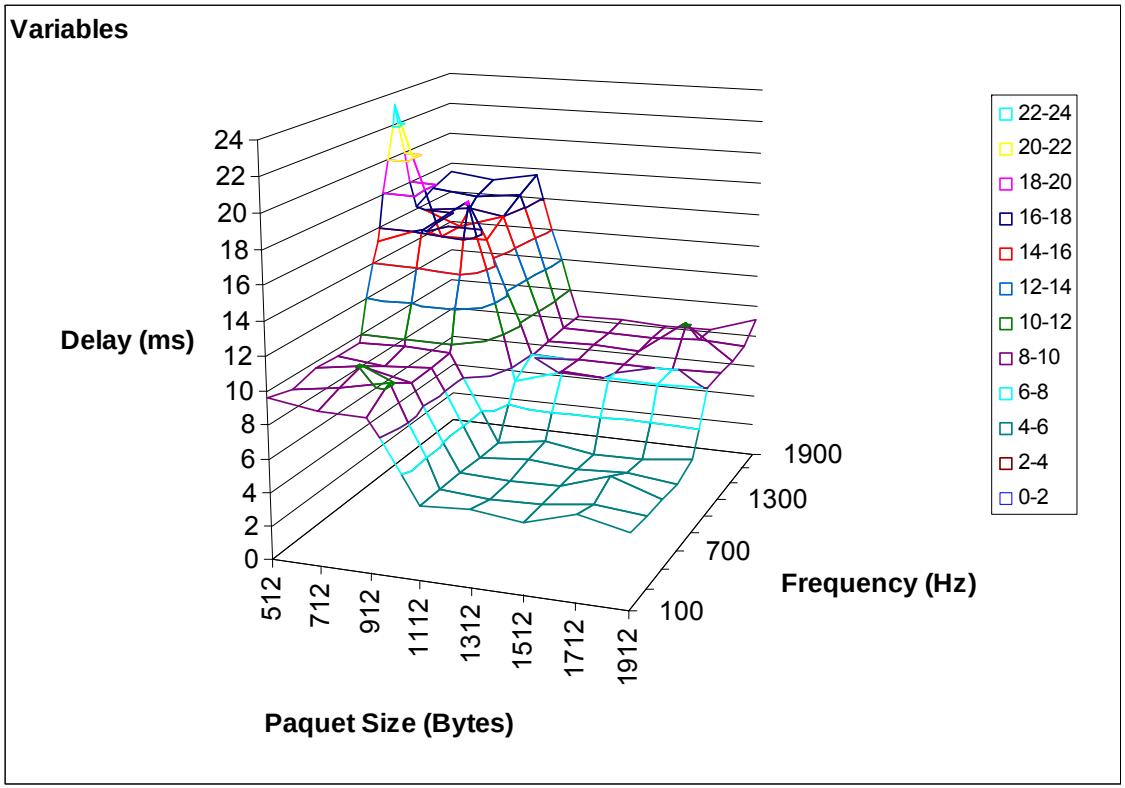


Fig. 2 Gráfico resultados Variables

ANEXO 9: Test Wifi

Eventos

Tabla 1 Tabla resultados Eventos

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	16,35583	13,42438	12,5922	12,44962	11,61173	1021,119	1024,181	1002,192	1009,552	946,3127
712	16,00805	12,97668	12,60575	12,52081	11,80819	1013,58	1025,515	1005,423	1085,182	956,0968
912	16,19466	12,95156	12,89616	12,95419	11,21982	1023,98	999,7481	1021,182	996,4759	1065,189
1112	10,19191	11,02301	10,52273	10,60412	11,00521	970,6586	1030,12	1036,942	989,9377	975,665
1312	10,75616	11,0242	11,49416	10,66432	11,01681	949,1978	1017,903	1003,952	983,3996	960,9543
1512	10,77803	11,03045	10,87259	10,73614	11,11686	1023,956	1022,722	992,1891	976,8614	962,5735
1712	10,78117	11,03333	11,0363	10,8002	11,15795	912,6483	1027,541	994,846	970,3232	960,1102
1912	10,78479	11,03706	11,21123	10,86621	11,21377	998,1941	1032,412	986,2306	963,7851	957,6469

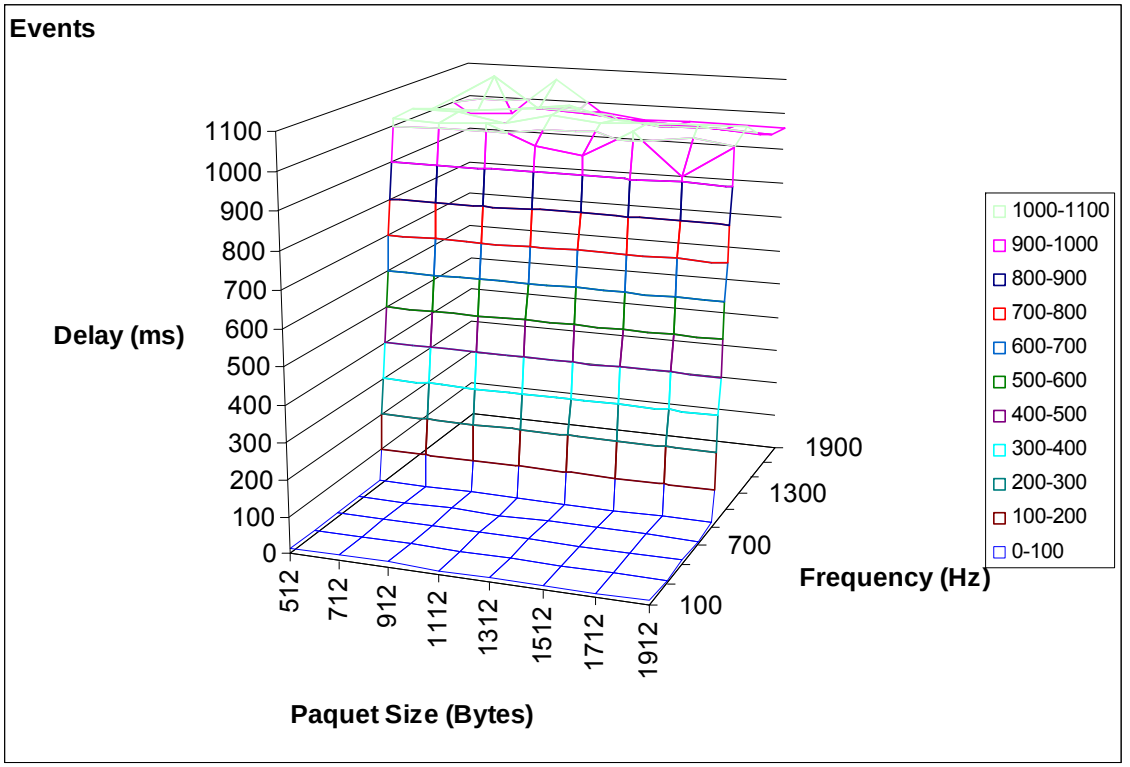


Fig. 1 Gráfico resultados Eventos

Variables

Tabla 2 Tabla resultados Variables

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	11,81043	12,58489	11,85085	11,72323	11,74344	18,76111	19,99614	19,99955	20,82405	21,44327
712	11,30929	11,87124	12,43319	11,31608	11,87802	20,0332	18,71458	18,76748	20,37287	21,02173
912	12,51895	11,38684	11,51068	11,63452	13,00095	22,40967	18,76809	19,75533	20,74256	21,7298
1112	5,380946	5,309306	5,237667	5,166027	5,094388	11,19191	9,871517	10,02875	10,18599	10,12632
1312	6,056189	5,008084	5,096572	5,561592	6,189499	9,95649	9,883611	10,10488	10,12111	10,24462
1512	5,127859	5,133914	5,189456	5,146024	5,152078	9,870277	9,898675	9,945349	0	0
1712	5,705425	6,094194	5,449548	5,32161	5,193672	9,746761	9,87026	10,10592	0	0
1912	5,378508	5,618161	5,095893	5,219814	5,614546	10,19156	9,950651	0	0	0

*Los valores en 0 indican que no se ha realizado correctamente el test por saturación de la conexión

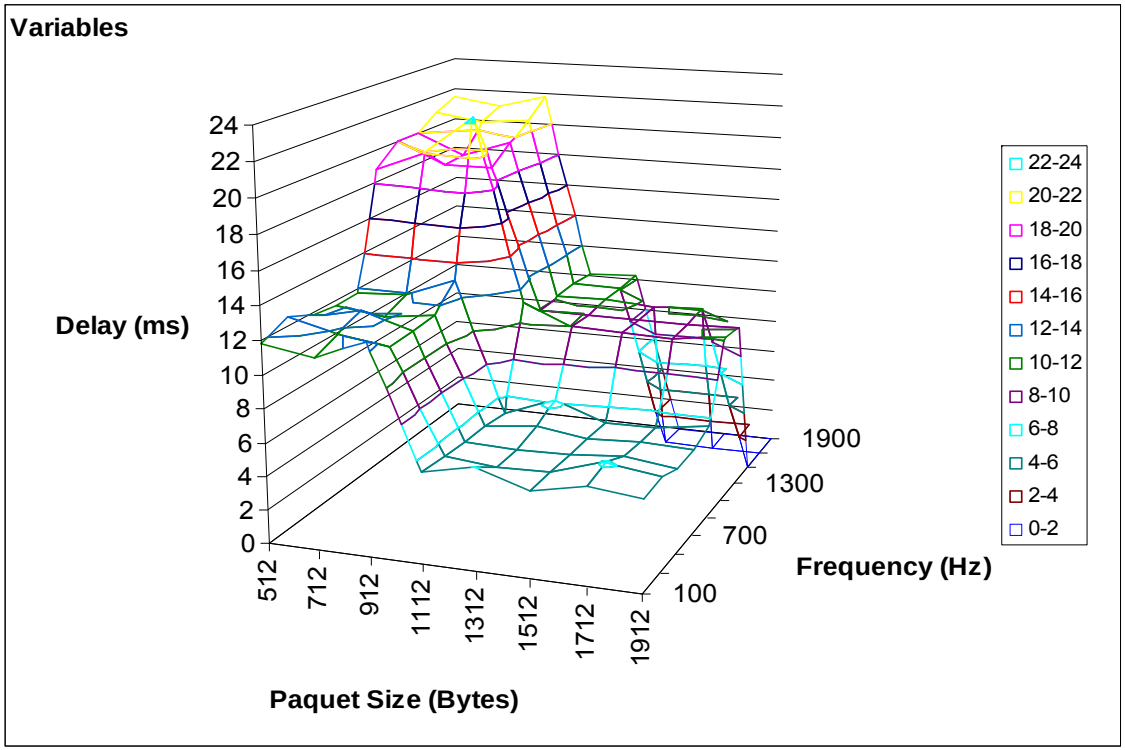


Fig. 2 Gráfico resultados Variables

ANEXO 10: Test Radio Frecuencia

Eventos

Tabla 1 Tabla resultados Eventos

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	20,5869	16,89712	15,84967	15,6702	14,61556	0	0	0	0	0
712	20,14916	16,33361	15,86672	15,7598	16,1916	0	0	0	0	0
912	19,71141	15,7701	15,88378	14,01916	15,56189	0	0	0	0	0
1112	12,82845	15,19191	13,24484	13,34729	14,19149	1221,757	0	0	0	0
1312	14,21919	13,87605	14,46757	13,42306	13,86674	1194,745	0	0	0	0
1512	13,56619	13,88391	13,68521	13,51346	13,99267	1288,842	0	0	0	0
1712	13,61984	13,88754	11,1911	13,5941	14,19846	1148,74	0	0	0	0
1912	13,5747	13,89222	14,11145	13,67718	14,11465	1256,416	0	0	0	0

*Los valores en 0 indican que no se ha realizado correctamente el test por saturación de la conexión

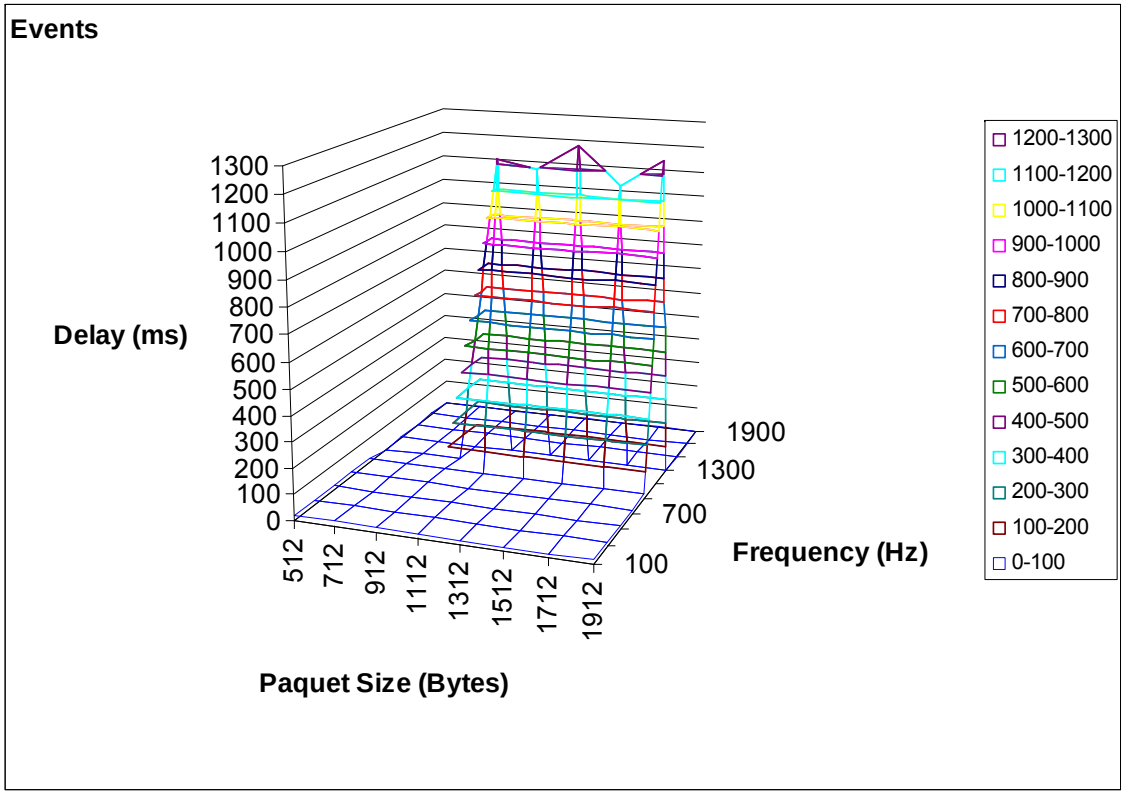


Fig. 1 Gráfico resultados Eventos

Variables

Tabla 2 Tabla resultados Variables

By\Hz	100	300	500	700	900	1100	1300	1500	1700	1900
512	14,39008	36,33446	14,44009	14,89492	14,30957	26,1985	0	0	0	0
712	15,58489	23,10411	18,54486	15,18492	14,11051	29,83256	0	0	0	0
912	23,68485	15,19098	17,08303	13,37029	14,55056	28,98329	0	0	0	0
1112	10,48833	7,948916	7,828303	6,169213	8,248808	21,81484	0	0	0	0
1312	9,030802	5,965474	9,934042	6,984919	7,416208	14,91697	0	0	0	0
1512	6,097996	6,189419	8,194926	8,191919	6,498792	0	0	0	0	0
1712	6,556457	9,087634	6,24785	7,87489	0	0	0	0	0	0
1912	8,04058	10,94963	7,591665	7,776424	8,362392	0	0	0	0	0

*Los valores en 0 indican que no se ha realizado correctamente el test por saturación de la conexión

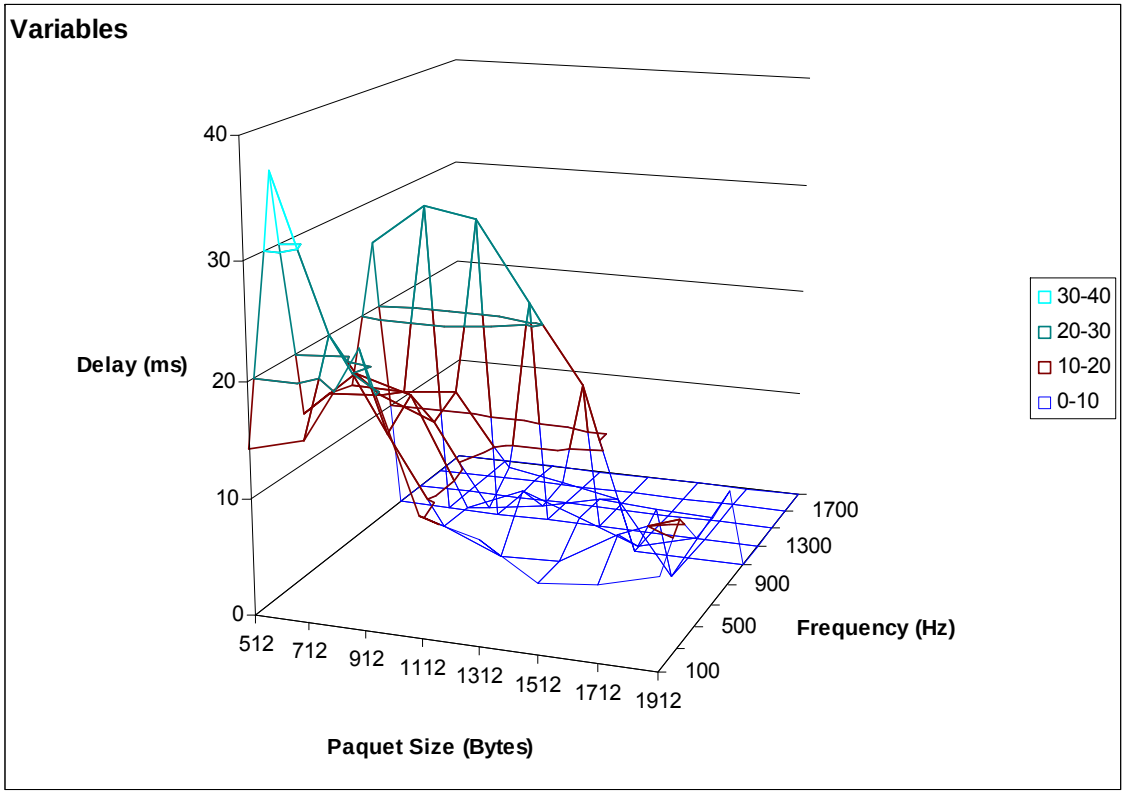


Fig. 2 Gráfico resultados Variables

ANEXO 11: CD Código del proyecto