Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

Title : MAREA 2. Design and Optimization of a Distributed Communications Middleware

Master Degree: Master in Science in Telecommunication Engineering & Management

Author: Santiago Pérez Fernández

Director: Juan López Rubio

Date: October 26, 2013

Title: MAREA 2. Design and Optimization of a Distributed Communications Middleware

Author: Santiago Pérez Fernández

Director: Juan López Rubio

Date: October 26, 2013

Overview

In recent years, the rapid growth of distributed embedded systems have been vigorously pushing middleware systems and technologies in different areas like: telecommunications, health care, automotive, defense, avionics, etc. The aim of this master thesis is to develop a new optimized version of the middleware MAREA 1, a software specifically designed to fulfill Unmanned Aircraft Systems (UAS) communications and their application to the design of complex distributed UAS avionics.

This document presents the software architecture of MAREA 2 and discusses design decisions, as well as some of the techniques employed to develop the middleware. MAREA 2 provides a more modular, flexible and reusable architecture and implements some new functionalities and features proposed in *Service Oriented Architecture for Embedded (Avionics) Applications* (López J., 2009). Another of the main contributions of this master thesis is the performance evaluation and optimization of the middleware through the analysis of some key performance parameters. The present document provides a comparison between the new and previous version of the middleware both in terms of design and performance.

Títol: MAREA 2. Design and Optimization of a Distributed Communications Middleware

Autor: Santiago Pérez Fernández

Director: Juan López Rubio

Data: 26 d'octubre de 2013

Resum

En els últims anys, el ràpid creixement dels sistemes distribuïts embeguts ha impulsat amb determinació els sistemes i tecnologies middleware a diferents àrees com: telecomunicacions, salut, automoció, defensa, aviònica, etc. L'objectiu d'aquest projecte de fi de màster es desenvolupar una nova versió del middleware MAREA 1, un software específicament dissenyat per complir amb les comunicacions dels Sistemes Aeris No Tripulats i la seva aplicació en el disseny de sistemes aviònics distribuïts complexos per Avions No Tripulats.

Aquest document presenta l'arquitectura de software de MAREA 2 i aborda les decisions del seu disseny, així com algunes de les tècniques emprades pel seu desenvolupament. MAREA 2 proporciona una arquitectura més modular, flexible i reusable i inclou algunes de les noves funcionalitats i característiques presentades a *Service Oriented Architecture for Embedded (Avionics) Applications* (López J., 2009). Una altra de les principals contribucions d'aquest projecte de fi de màster es l'avaluació i optimització del middleware a través de l'anàlisi d'alguns paràmetres clau de rendiment. El present document fa una comparativa entre la nova i l'anterior versió del middleware tant en relació al disseny com al rendiment.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

The main objective of this master thesis is to design and implement a new version of the middleware MAREA 1, a software specifically designed to fulfill Unmanned Aircraft Systems (UAS) communications and their application to the design of complex distributed UAS avionics.

Middleware is a system software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware, which provides facilities in order to build and use distributed systems [1].

The time and cost of the deployment of middleware solutions across heterogeneous environments continues to grow as technology evolves and becomes more complex. The objective of this master to address this issue regarding the design and deployment of services, is to implement a new service approach based on the deployment unit concept (López J., 2009).

An important work-line inside the middleware solutions is the support of naming services, where redundant services must be targeted with load balancing and fault tolerance. In relation to this topic, the objective of this master thesis is to implement a naming service that allows to find, share and access services and its communication primitives hiding the network complexity, and providing location transparency.

Actual middleware solutions have a hard time to build and deploy distributed real-time and embedded systems. In recent years, the development of efficient middleware architectures has become one of the big challenges of distributed real-time and embedded systems. One of the objectives regarding this, is to study and use software optimization techniques to improve the performance of the new design of MAREA middleware.

The document is structured in five different chapters. In addition to the chapters, the document includes some appendices with further information. A brief overview of each of the chapters follows:

- **Chapter 1:** presents a brief introduction to MAREA 2, and provides a high-level overview of the system architecture from an end user programmer's point of view.

- **Chapter 2:** is the first of three chapters that exclusively deals with the internal implementation details of MAREA 2. This chapter describes the design of the network system architecture and compares it with the one from MAREA 1 by contrasting results.

- **Chapter 3:** tackles the new design and implementation of the service container and provides a comparison with service container from MAREA 1.

- **Chapter 4:** introduces the design of the protocol layer and describes it interactions with all the other components of the architecture.

- **Chapter 5:** provides a summary of the conclusions reached and points out some future work.

CHAPTER 1. MAREA

The first part of this chapter introduces basic concepts about embedded systems and middleware technologies required for understanding the content of the present document.

The second part introduces MAREA middleware architecture, the communication mechanisms available to communicate services and the MAREA naming scheme. The last section of this chapter tackles the design and implementation of MAREA services from an end user programmer's point of view.

1.1 Middleware Context

This section sketches out briefly middleware systems and technologies. The first part introduces and describes distributed systems. Next, middleware is formally defined and the different types of middleware are analyzed. The section comes to an end some of the techniques applied in the development of next-generation middleware systems.

1.1.1 Distributed Systems

Distributed computing and distributed systems have gained popularity and importance over past years. The main purpose of this type of systems is to interact and exchange data between its set of components in order to share resources. This sort of systems could be perceived as a single integrated computing facility.

The common characteristics of this type of systems are: resource sharing, fault tolerance, concurrency, scalability, transparency and openness.

Distributed systems offer facilities to increase the performance, availability and reliability of applications. In general, this type of systems are cheaper than a centralized single system, because a large number of small, low-power systems tend to be cheaper than single computer.

This type of systems is more complex to build and maintain than an equivalent centralized system. One example of this is the software developing complexity introduced to ensure a proper coordination and communication between the distributed components. This complexity would be a major unwanted problem for application developers.

Another problem is the effect of heterogeneity in distributed systems. Distributed systems may contain many different types of components (software and hardware) working together in cooperative way to solve problems.

1.1.2 Middleware

Complexity and heterogeneity drawbacks of distributed systems could be solved or relived using a middleware. Middleware is a system software that resides between the applications and the underlying operating systems, network protocol stacks, and hardware, which provides facilities in order to build and use distributed systems [1].

This type of software provides a transparent and abstract vision of the low-level details (e.g. network communication, encoding, concurrency, protocol handling, etc.) facilitating end user programming. Middleware typically provides two different types of transparency to distributed systems:

- **Access transparency:** Hides differences between remote and local operations like data representation and invocation mechanisms.

- **Location transparency:** Hides the location of components. The different components could be redistributed (e.g. moved between computers) without changing any of the other components.

1.1.3 Types Of Middleware

The following subsection presents the main types of middleware and describe the most important requirements (scalability, reliability, heterogeneity, transparency, etc.) addressed by each alternative. The most widely known middleware implementations are also mentioned for each type of middleware.

1.1.3.1 Message Oriented Middleware (MOM)

Message oriented middleware (MOM), is a middleware that allows the communication between the components through messages. In this type of middleware, the coordination between the components could be achieved synchronously or asynchronously depending on the communication model of the MOM.

- **Message queuing:** This asynchronous indirect communication model uses a queue in order to exchange messages. The messages from the producer are stored into the consumer's queue after being sent. In this type of communication model, persistent queues are used when the reliability is required in front of performance. Quality of service (QoS) policies are also a good solution to provide reliability.

- **Message Passing:** In this direct communication model, the messages are sent directly to the interested parts through publish-subscribe pattern. First, the different parts register interest in receiving messages on a particular message topic. Then, consumers will receive any message corresponding to the subscribed topic.

MOM has a limited support for data heterogeneity because marshalling has to be implemented by the programmers. This type of middleware offers location transparency, inherent from publish-subscribe model, but has a limited support for access transparency. This lack of access transparency limits replication and migration transparency, complicating the scalability.

MOMs are build around a reliability paradigm that is suitable for applications where the availability of a network or all components is not warranted [2]. The most common implementation of this type of middleware are Sun's Java Message Queue and IBM's MQSeries.

1.1.3.2 Procedural Middleware (PM)

Procedural middleware (PM) is based on the concept of Remote Procedure Calls (RPC). RPC is an interprocess communication (IPC) mechanism which is designed to exchange data and invoke functions between client and server processes.

In this type of middleware, RPC servers contain procedures which could be invoked by remote clients across the network. PM provides location transparency because clients can invoke remote procedures as if it were local. RPC communications are synchronous, because clients remains blocked until the remote procedures have been executed.

RPC presents good heterogeneity because the relationships between servers an clients is defined through common interface with IDL (Interface Definition Language). Client does not need to know the language that server supports because the IDL compilers can translate the clients request. IDL compilers also marshalls and interprocess data automatically.

PM has a limited scalability due to the absence of replication and load balancing mechanisms. As opposite of MOM, PM does not support group communications.

This type of middleware is specially useful for simple point-to-point applications. The most widely known implementations of PM are Microsoft RPC Facility and Open Software Foundation's Distributed Computing Environment DCE.

1.1.3.3 Transactional Middleware (TM)

Transactional middleware (TM) supports the development of distributed systems through asynchronous transactions using a client/server model. A transaction is an atomic and logical sequence of events or operations. This implies that all set of operations are either executed or not at all.

TM use a request message to ask the system to execute a transaction. The transaction processing (TP) monitor is the responsible to coordinate the requests between clients and servers.

Regarding reliability, TP monitors manage the transactions with the two phase commit protocol (2PC), which is commonly used by relational database management systems (DBMS) to provide fault tolerance. Once "prepare to commit phase" have finished, the TP monitor asks the TM to commit the transactions and make it final to all servers ("commit

phase"). If at least one of them indicates that it cannot be executed, then the TP monitor asks all nodes to do a rollback in order to do not apply any change.

TP monitors provide reliability by means of support for replications and load balancing for the different server components.

One of the drawbacks of TM is the overhead introduced to manage and guarantee the transactions. This type of middleware also does not provide automatic marshalling and unmarshalling capabilities.

As mentioned before, TM is typically used in DBMS where the transactions have to be synchronized and controlled over multiple databases. BEA's Tuxedo, IBM's CICS, and Transarc's Encina are some typical implementations of this type of middleware.

1.1.3.4 Object Oriented Middleware (OOM)

Object oriented middleware (OOM) is an evolution of PM which extends its features adding object oriented capabilities.

OOM supports distributed object requests based on a client/server model. This type of middleware provides a local representation of remote objects, and hides the communication between remote objects and its local representation. The main idea is that all the objects can be accessed and invoked remotely anywhere in the network. OOM supports both synchronous and asynchronous communication.

There are many examples of OOMs like CCM (CORBA Component Model), SUN's Enterprise Java Beans, Java/RMI (Remote Method Invocation), Microsoft's DCOM (Distributed Component Object Model), CORBA (Common Object Request Broker Architecture), etc.

OOM presents good heterogeneity. For instance, CORBA supports different programming languages in the server and client. Java/RMI resolves the heterogeneity issue using Java Virtual Machine (JVM).

One of the drawbacks of this type of middleware is scalability. Only some specific implementations like Enterprise Java Beans and CORBA support replication and load balancing respectively.

OOM has high runtime and network communication overhead introduced by the support of features like service discovery. The bottlenecks appeared in the OOM technologies were due to many factors which includes excessive data copying, less compact encoding and complex encoding rules [3].

For other part, this type of middleware simplifies and provides a rapid integration of programming tasks for distributed and heterogeneous environments. This type of middleware is starting to be integrated with MOM.

1.1.4 Adaptive and Reflective Techniques

Most of the middleware used until today present a lack of flexibility and adaptability to different application environments and areas. Adaptive and reflective techniques have been noted as a key emerging paradigm for the development of dynamic next-generation middleware platforms [1]. This type of techniques improves configurability and provides dynamic adaptation to middleware systems and technologies.

Adaptive middleware is software whose functional behavior can be modified dynamically to optimize for a change in environmental conditions or requirements [4].

Reflective middleware applies reflection technique at middleware level. Reflection provides the ability for a program to observe or change its own code as well as all aspects of its programming language during runtime.

Reflexive and adaptive techniques are both useful separately, but become very powerful tool together. Reflective capabilities trigger adaptive capabilities by allowing system inspection in case a behavior adaptation is needed. During this work new reflective and adaptive capabilities will be added to MAREA middleware architecture.

1.1.5 Conclusions

This section introduces some basic concepts about distributed systems and middleware. Middleware provides mechanisms and tools that simplify the development of distributed applications. The distributed transparency provided by this type of software reduces the complexity of handling widely distributed systems.

The idea of this section is not to overwhelm the reader with explanations but only to provide as much information as is necessary to understand basic concepts about middleware. Some of these concepts are mentioned in the subsequent chapters with the aim of describe implementation details about MAREA 2.

1.2 Description

Middleware Architecture for Remote Embedded Applications (MAREA) is a middleware designed by ICARUS group specifically designed to fulfill Unmanned Aircraft Systems (UAS) communications and their application to the design of complex distributed UAS avionics [5].

The ICARUS group is composed by researchers of the Technical University of Catalonia - Barcelona Tech (UPC) mainly from the Computer Architecture Department of the Castelldefels School of Telecommunications and Aerospace Engineering (EETAC). The basic aim of this research group, formed in 2005, is develop technologies to automate air traffic management (ATM) with low cost UAS in civil airspace as well as more intelligent platforms that allow the deployment of civil applications for unmanned aircraft.

MAREA proposes a modular architecture based on services (SOA). These type of architectures ensures extensibility, flexibility and interoperability across heterogeneous environments. MAREA is a mixed MOM/OOM which implements a Data Distribution System communication model based on the publish-subscribe pattern. This middleware also offers additional features like RPC and file-based data transfer.

1.3 Communication Primitives

MAREA provides to the developers a different range of possibilities to interact and communicate the services between them through the following communication primitives:

- **Variables:** This type of communication primitive is designed to share periodic and short deterministic information between different services following a publish-subscribe model. In this type of primitives the data is sent in a best effort way, through UDP transport, taking in account the periodic behavior of the publisher and the limited lifetime of the information. An appropriate use case of this type of primitive is the telemetry provided by a GPS navigation device.

- **Events:** Similar to the variables, events are also used to share periodic and short information between services following a publish-subscribe model. As opposite of variables, the information in events is guaranteed to be delivered to all the subscribed services through TCP transport. This type of primitive is designed to share information about important and unpredictable facts. An example of this type of communication primitive can be any alarm used to inform about a critical system failure.

- **Remote invocation:** MAREA also offers an alternative Data Distribution System communication model to the publish-subscribe pattern like remote invocation. In this type of primitive the communication is established only between two services following the request/reply pattern using a client/server model. In remote invocation, as opposite of variables and events, the relation between the two services is punctual and it only lasts the execution time of the remote call. This type of primitive allows the use of multiple parameter and a single result value. Remote invocation is useful in these situations where a one-off action has to be taken.

- **File-based data transfer:** This type of communication primitive is used to transfer continuous information in an efficient way. In file-based data transfer the information is sent in chunks in a non reliable way through UDP Transport. File-based data transfer also provides a reliable control mechanism in order the consumer could notify the missing packets to the provider. This type of communication primitive is useful to share any kind of information like images and configuration files.

1.4 System Architecture

MAREA describes an architecture based on reusable services that can be distributed over a network of low-cost computing devices. As shown in figure 1.1, two distributed services are running and interacting in two different MAREA instances on the top of the middleware core. MAREA core has been designed according to the following two layer system architecture: service container and network layer.
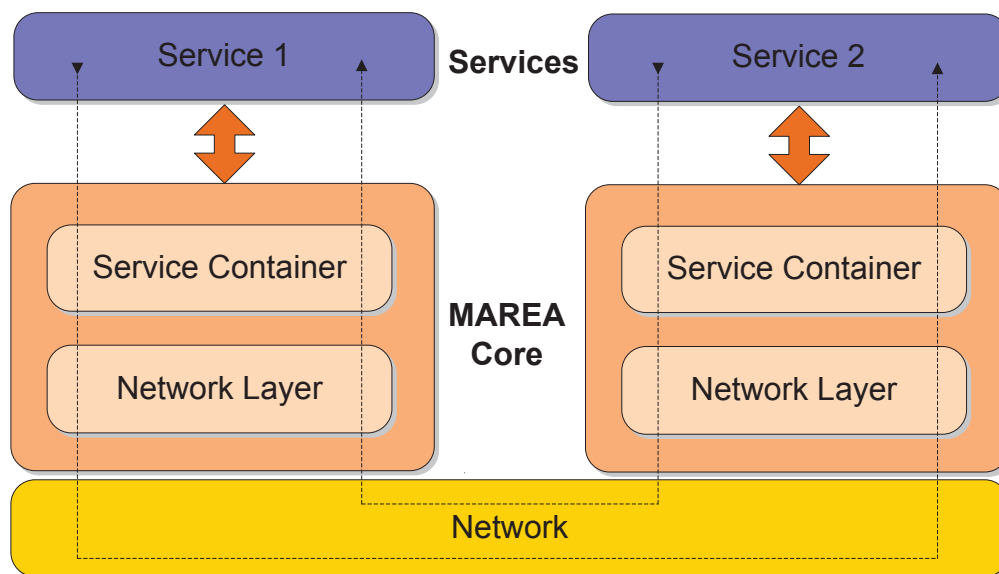
Figure 1.1: A high level view of MAREA core layers

The network layer, which is explained with more detail in chapter 2, is on charge of translate MAREA protocol messages in streams of bytes and send them through the network. This layer can also undertake the inverse operation: receive a stream of bytes through the network and translate them into MAREA protocol messages. One of the purposes of the network layer is to provide flexibility allowing the use of different transports and encodings depending on the scenario characteristics (e.g. hardware and software limitations). Network layer is a highly reusable component that could potentially send and receive any type of of object over the network.

MAREA services are managed and executed by the service container. This component allows message passing between services making communication between both local and remote services transparent, manages the communication protocols and primitives, etc. Service container decouples the services from the core hiding implementation details of some aspects like message management, service location and message delivery. Notice that, only one single service container is executed in each node of the distributed network.

The service container delegates the responsibility of managing and processing MAREA protocol messages to a sublayer called protocol layer (chapter 4). This component also manages the exchange of messages in order to discover and use the communication primitives of different services.

For one hand, network layer and protocol layer are responsible for providing message oriented capabilities to the system. For the other hand, the remaining parts of the service container are in charge of provide object oriented capabilities. Therefore, MAREA can be considered as a mixed MOM/OOM middleware.

1.5 Naming Service

The naming service allows MAREA to find, share and access services and its communication primitives hiding the network complexity. This component should address the different service and its communication primitives with name resolution.

An important requirement of the naming service is that the resolution name should not be bounded statically to specific locations. The main problem with that is if the service is moved to another location the reference to it becomes invalid. The proposed solution to this problem is to allow the usage of location independent service identifiers. The naming service provides fault tolerance to the system (services could be replicated in different nodes for redundancy).

The service identifier expression is a string comprising five fields separated by a slash which correspond to four different hierarchical levels. Each of the fields represent a hierarchical level name space, with exception of the service and the primitive are in the same level. The information of these hierarchical levels is represented by the following string:

/<subsystem>/<node>/<instance>/<service>/<primitive>

- **Subsystem:** A subsystem is defined as a logical group of nodes.

- **Node:** A node is defined as a group or just a single processor which runs MAREA middleware.

- **Instance:** The instance identifies a specific instance of a service.

- **Service:** The service identifies the type of the service.

- **Primitive:** The primitive identifies a specific communication primitive.

1.5.1 Address Types

MAREA hierarchical level naming scheme supports two different address types: single and query.

1.5.1.1 Single

Single addressing is applied to those service identifiers which represent a unique primitive or service (depending if the field of the primitive level is present or not in the service

identifier) inside the network or MAREA domain.

For instance the service identifier /Vehicle 1/Devices/Main/GPS represents a unique instance called Main of the service type GPS inside the node called Devices of the subsystem Vehicle 1.

1.5.1.2   Query

Query addressing is applied to those service identifiers which represent a group of primitive or services inside the network or MAREA domain.

This type of addressing allow the usage of special characters in each of the hierarchical levels to provide flexibility to the service location. The table 1.1 shows a brief description of each one.

Special character	Description	Usage
*	Selects any element of the actual field/hierarchical level	Any of the fields of the service identifier
*	Forces dynamic name resolution for the given service identifier	At the beginning of the service identifier
#	Forces static name resolution for the given service identifier	At the beginning of the service identifier
!	Forces static name resolution and locks the service with highest priority of the set referenced by the service identifier	At the beginning of the service identifier

Table 1.1: MAREA naming special characters

For instance, */Vehicle 1/Devices/*/GPS means all the GPS type instances in the node Devices belonging to subsystem Vehicle 1, regardless of their instance name.

MAREA presents by default dynamic name resolution. This means that if a new service is started, the naming service is able to notice it and update all the name resolutions which contain a reference to this service. In some specific cases its required the use of static name resolution. In these situations future changes will not be taken into account. For instance, #/Vehicle 1/*/*/GPS means all GPS type instances of Vehicle 1, regardless the node and their instance name, obtained at search time. In this specific case, changes like the addition of new GPS services or the removal of existent GPS services will not be taken into account.

1.6   Service

MAREA services have been designed according to the deployment unit concept. A deployment unit is a compressed file that contains a single module of application code loadable

by the service container [6].

MAREA defines the following three different types of deployment units:

- **Service Deployment Units (SDU):** Contains the implementation of the service. It has a dependency with its corresponding IDU.

- **Interface Deployment Units (IDU):** Contains the description of the service. This description is defined by an interface.

- **Library Deployment Units (LDU):** Contains libraries or other dependencies required by the service.

For example, consider a barometric and a radioelectric altimeter which both implement the same interface (IDU), but the implementation (SDU) of them is different. With the deployment unit approach, the interface can be shared among them. Following with this example, a possible LDU could contain data about the altimetry which is information susceptible to be used by other services (e.g. GPS).

The proposed way to distribute deployment units is to build a private package management system using the NuGet tool. The final idea is setup a private NuGet server in order to host the deployment units as NuGet feed. A proof of concept has been made in order to test the feasibility of using NuGet as a package management system. Some of the details are introduced in appendix A.1.

The following subsections detail the implementation of the IDU an SDU of a service with an example of a multiple battery management system. The system is composed of a BatteryManager service on charge of monitoring the charge of the batteries and multiple Battery services connected to the sensing hardware installed in the different battery packs. From now to the end of the document, this example is referenced in several sections.

1.6.1   Service Description

MAREA services are described by an interface (IDU) which contains the different communication primitives published by all the services that implement the interface.

The listing 1.1 contains the interface of all the Battery services. It declares three different primitives: a variable exposing the amperage, an event notifying low charge condition and a function that controls the recharging hardware of the battery.

The interface also contains additional data to describe the service and their communication primitives. Optional information like description, units or a property to publish or not the primitive, is assignable by using different attributes.

Listing 1.1: Battery IDU

```
[ServiceDefinition("This service represents a battery")]
public interface IBattery
{
 [Description("This variable represents the amperage of the battery")]
 [Unit("A")]
 [Publish]
 Variable<double> amperage { get; }

 [Description("This event informs that the battery is low and needs to be recharged")]
 [Publish]
 Event<None> lowBattery { get; }

 void Recharge(bool b);
}
```

The listing 1.2 contains the interface of all the BatteryManager services. It declares an event notifying a warning condition.

Listing 1.2: BatteryManager IDU

```
[ServiceDefinition("This is a battery manager")]
public interface IBatteryManager
{
 [Description("This event warns when any of the batteries of the system is low.")]
 Event<String> globalBatteryWarning {get;}
}
```

1.6.2 Service Implementation

The SDU defines a specific implementation or definition for an interface (IDU). MAREA service implementation offers the following common interface operations to manage services:

- **Start:** This operation is executed when a service is started. This method is responsible for allocating resources and starting the service functionality.

- **Stop:** This operation is executed when a service is asked to finish. This method is responsible for stopping the service functionality and freeing the allocated resources.

Considering the battery management system, each different battery package will require a different service implementation. However, the service implementation details of the Battery services will remain hidden by the same compatible interface. In this way, all the Battery implementations are interoperable and could be accessed in a uniform, transparent manner. This means that the BatteryManager does not notice any differences among the different Battery implementations.

The listing 1.3 depicts the implementation of a dummy Battery service. The Start method is on charge to create and start a thread which executes the Run method. This dummy method is on charge to notify the value of the amperage every second. A lowBattery event is also generated if the amperage value is under 0.5 A.

Then the Recharge method is implemented. This method is inherited from the service description (IDU) like the variable and event communication primitives located in IBattery Members region. Recharge method is called transparently when other service calls this specific remote invocation communication primitive.

Listing 1.3: Battery SDU

```
public class Battery : Service, IBattery
{
 protected Thread th;
 protected bool isStarted = false;

 public override bool Start()
 {
  th = new Thread(new ThreadStart(Run));
  isStarted = true;
  th.Start();
  return true;
 }
 public override bool Stop()
 {
  isStarted = false;
  return true;
 }

 protected void Run()
 {
  Random r = new Random();
  bool state = false;

  while (isStarted)
  {
   amperage.Notify(id, r.NextDouble());

   if(amperage.Value < 0.5)
    lowBattery.Notify(id, None.Instance);

    Thread.Sleep(1000);
 }

 public void Recharge(bool t)
 {
   Console.WriteLine("Recharge=>"+ t);
   //Here should go the command to recharge or not the battery
 }

#region IBattery Members

 public Variable<double> amperage { get; private set; }

 public Event<None> lowBattery { get; private set; }

#endregion
}
```

One of the benefits of the deployment unit paradigm is that types and parameters of the service implementation are checked with the service interface at compile time. In this way, the compilation process will not start if the parameters and types of the interface and the implementation do not match.

The listing 1.4 depicts the implementation of a BatteryManger service. This service consumes the communication primitives published by the batteries which implement the Battery interface (see IBattery object called bat).

To consume a variable or event it necessary to subscribe the primitive to a specific method. This method will act as a callback when new data is available (see AmperageChanged and LowBatteryChanged methods). As shown in listing 1.4 this subscription and unsubscription process is executed inside the Start and Stop methods respectively.

Each time a new value of amperage is generated the AmperageChanged method is called. If the new value of the amperage is less than 0.1 A the event globalBatteryWarning is fired. This event uses a string with the name of the battery to discriminate the battery with low charge among all the batteries of the system.

Listing 1.4: BatteryManager SDU

```
class BatteryManager : Service , IBatteryManager
{
 [LocateService("*/EC-UPC/*/bat1/Battery")]
  private IBattery bat;

  public override bool Start()
  {
   bat.amperage.Subscribe(id, AmperageChanged);
   bat.lowBattery.Subscribe(id, LowBatteryChanged);
   return true;
  }

  public override bool Stop()
  {
     bat.amperage.Unsubscribe(id, AmperageChanged);
     bat.lowBattery.Unsubscribe(id, LowBatteryChanged);
     return true;
   }

  public void AmperageChanged(String name, double amps)
  {
   Console.WriteLine("["+id+"]"+" Variable: "+name+" Value: " +amps);
   if(amps<0.1)
    globalBatteryWarning.Notify(id, name);
  }

  public void LowBatteryChanged(String name, None none)
  {
   Console.WriteLine("[" +id + "]"+" Event: " + name +" Value: " +none);
  }

  #region IBatteryManager Members

  public Event<string> globalBatteryWarning { get; private set; }

  #endregion
}
```

1.6.2.1 Service location and consumption

MAREA services could consume primitives from a specific service or a set of services which implement the interface (IDU) of the service. This depends on the naming addressing type used (section 1.5.1) in the LocateService attribute.

Now consider the a scenario based on the battery management system in a single node. This node runs two different Battery services and one BatteryManager.
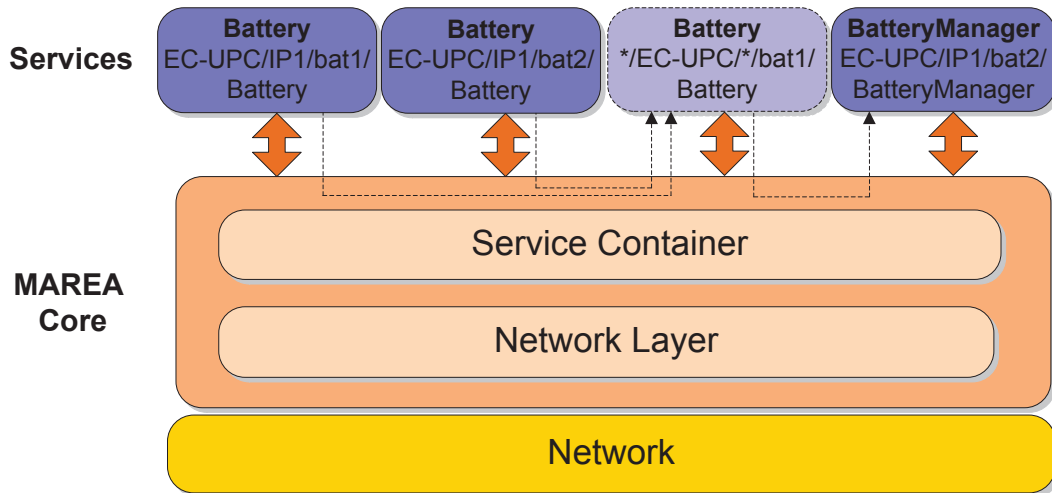
Figure 1.2: Services interacting with a query proxy in a multiple battery management scenario

According to the implementation of the BatteryManager depicted in listing 1.4 the Battery-Manager located in node 2 will consume the communication primitives from all the services which match with the query */EC-UPC/*/bat1/Battery (see LocateService attribute in IBattery object).

Before following with the example, is necessary to define proxies. Proxy objects are services that implement the same interface (IDU) as the represented service and control the access to the represented service. Proxies just act as redirectors an do not add any extra specific functionality for themselves.

As shown in figure 1.2, the service container will generate a proxy object that represent the set of services selected by the query (service in light blue color). In this case the represented service corresponds to the result of the query */EC UPC/*/bat1/Battery, which corresponds to both Battery services.

Another kind of proxy used by MAREA middleware is remote proxy. This type of service provides a local representative for a service that reside in a different service container or node than the current one. The functioning of both, remote and query proxies, is detailed in section 3.3.1.

1.7   Conclusions

MAREA 2 proposes an architecture based on services that ensures extensibility, flexibility and interoperability across heterogeneous environments. MAREA is a mixed MOM/OOM that provides four MAREA different types of communication primitives to interact and communicate the different services: variable, event, remote produce call and file-based data transfer.

MAREA core has been designed according to the following two layer system architecture: service container and network layer. Service container is responsible for managing and executing services following the deployment unit approach. Network and protocol layers are on charge of offering network access and remote message delivery capabilities respectively. The new implemented naming service allows to find, share and access services and its communication primitives hiding the network complexity.

CHAPTER 2. NETWORK LAYER

The first part of this chapter presents a general overview of the new network system architecture. The following subsections describe each network component with more detail, and present results to the measurements of some performance parameters that represent most critical capabilities and characteristics of the network system architecture. Some of these the key performance parameters are: simultaneous connections, round-trip time (RTT) and memory allocation.

The chapters comes to and end with a comparison between the current and previous design of the network layer.

2.1 Architecture

The network layer is the lowest level layer on the MAREA stack. This layer it is on charge of providing an optimized, modular and reusable usage of the network capabilities. MAREA network system architecture consist of two main sublayers: encoder and transport.
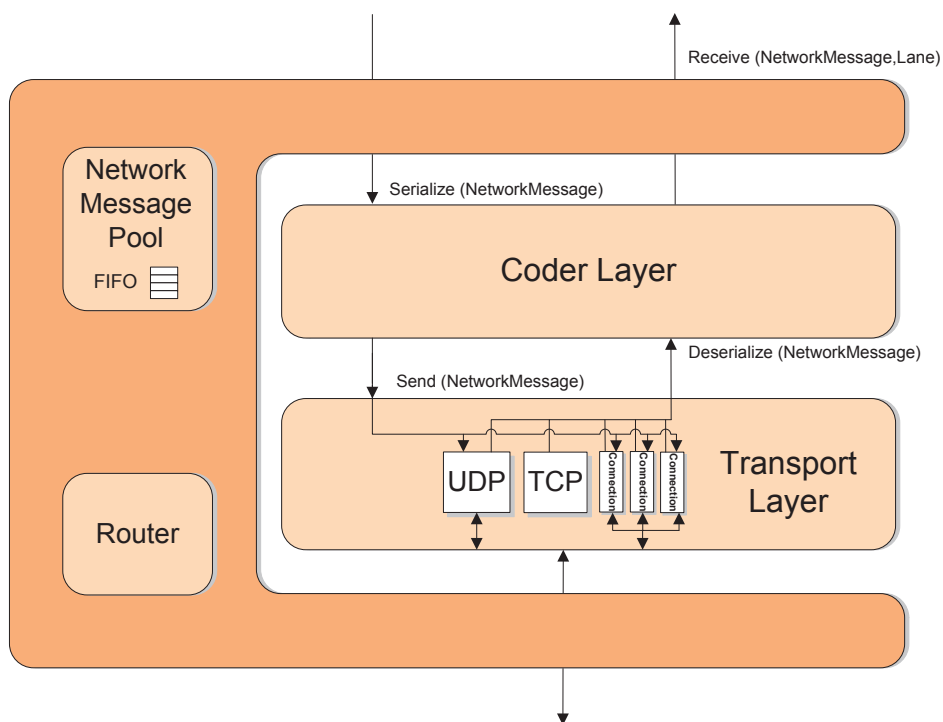

Figure 2.1: MAREA 2 network layer architecture

One one hand, the encoder layer is responsible for coding MAREA protocol messages into byte sequences. This component also undertakes the inverse operation of decode byte sequences into MAREA protocol messages.

On the other hand, the transport layer is on charge of send and receive data (byte sequences) from the underlying network through transports (UDP, TCP).

The Router is primarily responsible for selecting encoders and transports dynamically by using network lanes (section 2.2.1). This component plays a very important role in offering adaptive capabilities to the network layer and in building a highly reconfigurable system.

The idea of the proposed design, in order to improve the performance in terms of speed, is to minimize the amount of time used by the garbage collector to create and destroy object instances. The architecture component called NetworkMessage Pool (section 2.3) achieves this improvement using a memory pool mechanism.

In order to provide uniformity and simplicity to the design of each sublayer, the communication between them is done using a common interface. Each of the sublayers send and receive a NetworkMessage entity (figure 2.2) to the upper and lower layers. This common data structure, which contains all the necessary information used by the encoding and transport layers, is modified as it travels downward or upward the architecture.
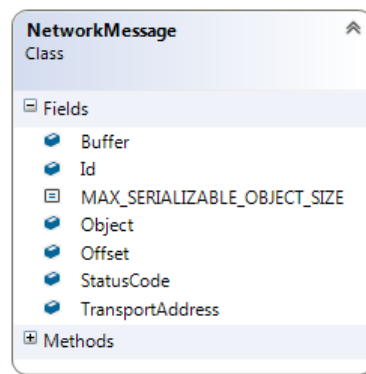
Figure 2.2: NetworkMessage entity

Next is detailed the flow of NetworkMessage entities through the entire network system architecture depending on the network lane used. The interaction with different network components (encoder, transport and NetworkMessage Pool) is also explained.

- **Output lane:** Every time a MAREA protocol message is received from the service container a NetworkMessage is dequeued from the NetworkMessage Pool. Then, the MAREA protocol message is stored in the field Object of the NetworkMessage entity. The network output lane is consequently executed and the NetworkMessage entity automatically starts to flow downward the network architecture.

  First, the encoder sublayer serializes the MAREA protocol message, contained in the field Object of the NetworkMessage entity, and stores the resulting byte stream in the Buffer field of the same NetworkMessage. At the same time, the total length of the serialized data is assigned to the field Offset.

  Second, the transport layer sends the set of bytes specified by the Offset field from the buffer of the NetworkMessage entity through the network. It is important to note that the network lane takes reference directly to the TCP connection or UDP

transport depending on the type of protocol required. Finally, the NetworkMessage is enqueued in the NetworkMessage Pool.

- **Input lane:** Every time a byte stream data representation of a MAREA protocol message is received in the transport layer from the network, a NetworkMessage is dequeued from the NetworkMessage Pool. Then, the byte stream is stored in the field Buffer of the NetworkMessage entity. At the same time, the total length of the received data is assigned to the field Offset. The network input lane is consequently executed and the NetworkMessage entity automatically starts to flow upward the network architecture.

  First, the encoder sublayer deserializes the set of bytes of specified by the Offset field from the buffer into a MAREA protocol message, which is consequently stored in Object field of the NetworkMessage entity.

  Second, the network layer forwards the MAREA protocol message stored in the field Object of the NetworkMessage to the service container. Finally, the NetworkMessage entity is enqueued in the NetworkMessage Pool.

In relation to the transport layer, when a message is received the fields StatusCode and TransportAddress are set in order to inform the upper layers about the reception status and the source of the incoming MAREA protocol message.

The field Id of the NetworkMessage entity is a byte code identifier used by the encoder layer to encode and decode MAREA protocol messages. This identifier is also reused by the service container in order to process the MAREA protocol message according to the type of message and the protocol (discovery, publish-subscribe, rpc) which belongs.

2.2   Router

MAREA is able to use different encoders and transports in order to build a modular and configurable network architecture. The Router has the ability to select the elements (encoders and transports), of the layered network architecture, at execution time depending on needs and the state of the network. The main aim of this element is to create and manage the network lanes.

2.2.1   Network Lanes

A network lane can be defined as a set of references to the bindings establish between the different network architecture elements (encoder and transports) used at a particular moment. Lanes have been implemented as linked lists of delegates or multicast delegates. A delegate is an object that allows the programmer to encapsulate a reference to a method. A delegate is similar to a function pointer in C or C++ but is object-oriented, type-safe, and secure [7].

The following characteristics of multicast delegates have been taken in account to create network lanes:

- The invocation list of multicast delegates is called synchronously and orderly.

- If an exception occurs in a delegate, the remaining delegates of the list are not invoked.

According to the figure 2.2 the Router use two different network lanes to send and receive data, output and input lanes respectively.

Lanes	Invocation List
Output lane	MareaCoder.Serialize(NetworkMessage m)
	Transport.Send(NetworkMessage m)
Input lane	Coder.Deserialize(NetworkMessage m)
	Container.Receive(NetworkMessage m)

Table 2.1: Output and input network lanes

A way to trap link loss or disconnection exceptions is required in order to notify the upper layers that an error has occurred. There exist two different solutions to this issue:

- Use the method GetInvocationList to get each individual delegate from the multicast delegate and invoke each delegate within the try block of an exception handler. This solution is very powerful but it has counterparts like the use of system resources and execution time due to the handling of exceptions.

- Use a field in the NetworkMessage entity as a status code (figure 2.2).

The second alternative has been implemented in order to accomplish with the objective of improve the performance.

2.3   NetworkMessage Pool

The non deterministic process of garbage collection is executed .NET virtual machine in order to maintain the memory clean. This process can introduce non deterministic pauses into the execution of a program which are not correlated with the algorithm being processed.

One solution in order to reduce garbage collection interruptions is use a memory pooling mechanism. The main idea of this type of mechanisms is to provide a managed set of functions in order to allocate and deallocate memory. Pooling mechanisms keep references to object instances that are beyond destruction, allowing it to be reused when needed. With

this technique no objects (NetworkMessage entities) are released to be garbage collected until the middleware is shut down.

The proposed design to implement a memory pool mechanism is to use a FIFO queuing discipline for NetworkMessage entities. In this common queue disciple, the elements are added to the tail and removed from the head using a pair of object references to the tail and the queue.

The final purpose of the NetworkMessage Pool is to reduce the amount of work that has to be done by the garbage collector and consequently minimize the time used by its own execution.

2.3.1   Results

A memory allocation profiling test has been executed in order to evaluate the behavior of the NetworkMessage Pool. The figure 2.3 and the table 2.2 present the total bytes allocated by MAREA 1 and the MAREA 1 network backport (section 2.6) during an echo request/response test with two MAREA instances. Each instance runs a different service which sends or responds to the message. This results correspond to the total bytes allocated by the requester.

The test has been executed 10000 times to send variables (UDP) and events (TCP) primitives with a total payload of 1000 bytes and frequency of 100 Hz. MAREA 1 network backport has been tested in two different modes: reusing network lanes and creating every time on demand.
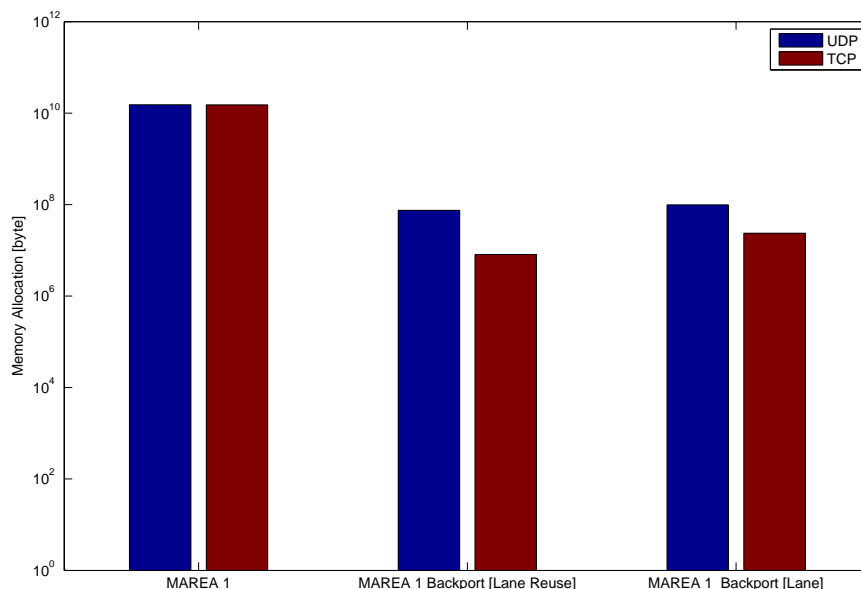

Figure 2.3: Number of bytes allocated by MAREA 1, MAREA 1 network backport in an echo test (requester side): 1000 Bytes, 10000 times, 100 Hz

Middleware	Bytes Allocated	
	UDP	TCP
MAREA 1	15222996197	15159086765
MAREA 1 Backport [Lane Reuse]	74992540	8092972
MAREA 1 Backport [Lane]	98911888	23693442

Table 2.2: Number of bytes allocated by MAREA 1, MAREA 1 network backport in an echo test (requester side): 1000 Bytes, 10000 times, 100 Hz

2.4   Encoding Layer

The encoding layer is on charge of serializing and deserializing MAREA messages. This layer provides an abstraction layer such all logic above contained in the upper-layers does not need to know the particulars about how messages are serialized and deserialized.

Serialization is the act of taking an in-memory object or object graph (set of objects that reference each other) and flattening it into a stream of bytes [8]. The reverse operation is deserialization which takes a data stream and regenerates into an in-memory object or object graph.

2.4.1   Previous Work

The initial version of MAREA has been implemented with the idea of providing several encoding layer implementations (XML serialization, binary serialization and MAREA coder) in order to allow adaptability and interoperability between the devices and the network. The first two implementations of the encoding layer use .NET Framework serialization mechanisms such as binary serialization through BinaryFormatter class, and human-readable XML serialization through XMLSerializer class.

Feature	BinaryFormatter	XMLSerializer
Level of automation	*****	****
Type coupling	Tight	Loose
Version Tolerance	***	*****
Can serialize nonpublic fields	Yes	No
Preserves the object reference	Yes	No
Suitability for interoperable messaging	**	***
Flexibility in reading/writing XML files	-	****
Compact output	****	**
Performance	****	* To ***

Table 2.3: Serialization engine comparison between .NET BinaryFormatter and XMLSerializer [8]

The BinaryFormatter is easy to use and automatic, but it is not such flexible as XMLSerializer. On the other hand, XMLSerializer is slower and less powerful because it is not able to restore shared object references. Furthermore, XML serialization does not convert private fields, indexers, methods, or read-only properties (except read-only collections). In order to do this, is mandatory to use the BinaryFormatter class.

One of the main drawbacks of these two mechanisms is the performance overhead. Serializing a message with BinaryFormatter is expensive because of the metadata present. This is more noticeable in XMLSerializer because the overhead introduced by the XML tags is bigger.

Another disadvantage of these two mechanisms is the interoperability between the different virtual machine representations of .NET Frameworks. For instance, XML serialization is not available on the Micro Framework and binary serialization works different in .NET Framework and .NET Compact Framework.

The last implementation of the encoding layer, which is called MAREA coder, has been designed in order to solve these two drawbacks controlling the serialization and deserialization of the different types. This technique, allows the programmer to have more control over the serialization and deserialization processes and ensures serialization compatibility.

The first implementation of this encoder was made using introspection to serialize and deserialize each message dynamically. The results of this first approach were not satisfactory in terms of speed because introspection it is a slow process. Custom serialization solves this issue by using specific methods or routines to serialize and deserialize specific MAREA messages.

MAREA coder has better performance in terms of speed and serialized data size than .NET BinaryFormatter and XMLSerializer implementations. MAREA coder is not dependent of the .NET Framework, so the interoperability between different virtual representations of the .NET Frameworks is not a problem like in .NET BinaryFormatter and XMLSerializer implementations.

2.4.2 Drawbacks

MAREA coder has some drawbacks inherited from custom serialization like complexity, especially in those cases like tree of objects or object graphs that might contain cycles. In these cases the code could be really hard to study.

Another point that has to be taken in account of this approach is development speed. Custom serialization does take time for testing, developing and maintenance. For instance, if some messages are added or modified in the protocol layer, the corresponding methods to serialize and deserialize these messages must be added or modified too in order the encoder layer continues to work properly.

MAREA coder has been designed with two serialize and deserialize entry point methods that implement a large switch statement to get the type of object that has to be serialized/deserialized. A large switch statement means the method is large, hard to read and

can generate very high complexity metrics.

2.4.3 Improvements

The following subsection presents an alternative design for the encoding layer in order to solve the drawbacks of MAREA coder.

The new proposed design is based on an automatic tool called MAREAGen. This tool generates classes automatically with methods to serialize and deserialize MAREA entities marked as serializable. This eliminates the need for developers to implement serializing and deserializing code and guarantees run-time type safety.

Serialize and deserialize methods have been implemented as static because they have no instance. This type of methods is slightly faster than instance methods because are called with type name instead of an instance identifier. Serialize and deserialize methods also include inlining through the method implementation option aggressive inlining (listing 2.1). This option allows compiler to eliminate the cost of method calls if it is possible.

Listing 2.1: Class to serialize/deserialize MAREA SlowData messages

```csharp
public class MG_SlowData
{
  /**
   * This static constructor is called by MAREA in order to load
   * and register the identifier and specific methods to serialize
   * and deserialize this type.
   **/
  static MG_SlowData()
  {
    M2CoderTables.GetInstance().AddClass(typeof(Marea.SlowData), 50,
    MG_SlowData.Decode, MG_SlowData.Encode);
  }

  public static readonly ulong MAREAGEN_FINGERPRINT= 11653293;

  /**
   * This method serializes all the different objects contained
   * inside a SlowData message into the given byte array.
   **/
  [MethodImpl(MethodImplOptions.AggressiveInlining)]
  public static void Encode(object theSlowData, byte[] buffer, ref int offset)
  {
    // Serialize fields...
  }

  /**
   * This method deserializes all the different objects of  a
   * SlowData message from the given byte array. This method also
   * returns the whole SlowData object
   **/
  [MethodImpl(MethodImplOptions.AggressiveInlining)]
  public static object Decode(byte[] buffer, ref int offset)
  {
    SlowData slowdata = new SlowData();
    // Deserialize fields...
    return slowdata;
  }
}
```

MAREAGen provides fingerprints in each of every generated class, derived from the type

definition, in order to provide an unequivocal and fast way to detect code that had not been recompiled.

At the end of every execution, MAREAGen generates a XML and a DLL file which contains a list of the generated types with its unique byte code identifier and the generated classes to serialize and deserialize MAREA serializable classes respectively.

The proposed solution to solve the complexity issues is to include a hash table and an array of delegates for serialization and deserialization methods. Each of these collections also stores a reference to the byte code identifier provided by MAREAGen: the key values in case of the hash table and the index in case of the array.

One of the things that has to be accomplished in order to store all the delegates into a same dictionary is that all of the different serialize and deserialize methods must have a compatible signature (encode and decode methods from listing 2.1).

With this approach, the complexity of having large methods with a lot of switch statements is reduced by moving each of the code sections, for every type of MAREA message, to a specific serialize and deserialize methods.

Furthermore, the speed performance should be improved using the dictionary with the byte code identifiers, because with this solution long switch statements are avoided. The proposed approach is also more scalable: if the number of MAREA messages grows the serialization time should maintain constant, because it only depends on the time to access time to the delegates contained in the dictionary.

The table 2.4 presents the proposed byte code identifier distribution according to the different types used by MAREA Coder. MAREA protocol message identifiers are assigned at the beginning in order to reuse them in the service container. Similar to the encode layer, the service container has an array of delegates used to process each MAREA protocol message according to its identifier. In this case the position of the delegates inside the array correspond to the MAREA protocol message byte code identifier.

Id	Type
From 0 to 63	MAREA protocol messages
64	Null
From 65 to 126	MAREA Coder basic types
127	Not null
From 128 to 255	MAREA Coder types created by MAREAGen

Table 2.4: MAREAGen identifier distribution

Once MAREA is started the DLL file generated from MAREAGen tool is used by MAREA coder in order to add the byte type codes and delegates of the generated classes in the dictionary. This task is performed automatically by calling the constructor of those classes that have been previously generated by MAREAGen.

Marea coder serialize and deserialize specific methods have also been modified in order

to reduce the data size. For instance, the size of serialized System.Double is now reduced from 11 Bytes to 5 Bytes (1 of this 5 bytes is Marea to encode the number 19 that indicates that the payload is a double).

Another new feature of this implementation is the support for some of the most commonly used .NET collection types like lists, dictionaries and hash tables.

The following bugs have also been resolved according to the results obtained in the implemented unit testing with NUnit tool:

- Compatibility with UTF8 character encoding.

- Overflow exceptions for high values (double.MaxValue) in double types.

- Support for run-time/dynamic Enum types.

- Full support of polymorphic capabilities: Inheritance and control of null objects in object trees.

2.4.4 Results

The figure 2.4 and the table 2.5 present the total serialization and deserialization time for some specific types for the previous and the new implementation of MAREA coder. The most representative type is the specific MAREA message SlowData. This message is used by MAREA to transmit the data different MAREA primitives (variables and events). This time is 223.49 and 19.64 μs for the old and new implementation respectively. Notice that the y axis in the figure is a logarithmic scale.
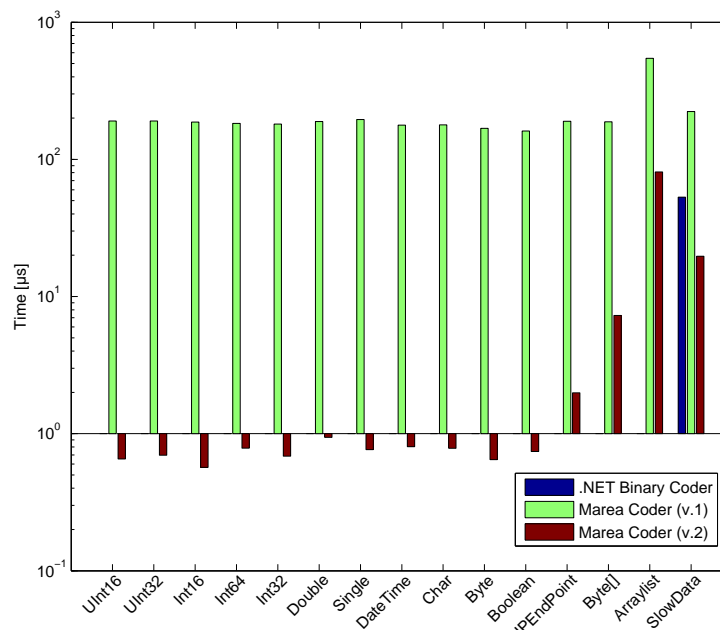

Figure 2.4: Encoder layer implementations: Total serialization and deserialization time

Type	Marea Coder (v.2)	Marea Coder (v.1)	.NET Binary Coder
	Total Serialization and Deserialization Time (µs)		
UInt16	0.6542	190.4008	
UInt32	0.6965	190.5905	
Int16	0.5677	187.2317	
Int64	0.7839	182.9995	
Int32	0.6850	181.1064	
Double	0.9407	188.7585	
Single	0.7644	195.3172	
DateTime	0.8038	177.8468	
Char	0.7824	178.3560	
Byte	0.6464	168.3250	
Boolean	0.7418	161.2008	
IPEndPoint	1.9847	189.7303	
Byte[]	7.2739	187.8259	
Arraylist	80.9052	545.5369	
SlowData	19.6484	223.4966	52.9961

Table 2.5: Encoder layer implementations: Total serialization and deserialization time

The figure 2.5 and the table 2.6 present the total serialization and deserialization time for the new supported types in MAREA coder. These latencies are also compared with the .NET BinaryFormatter coder implementation.
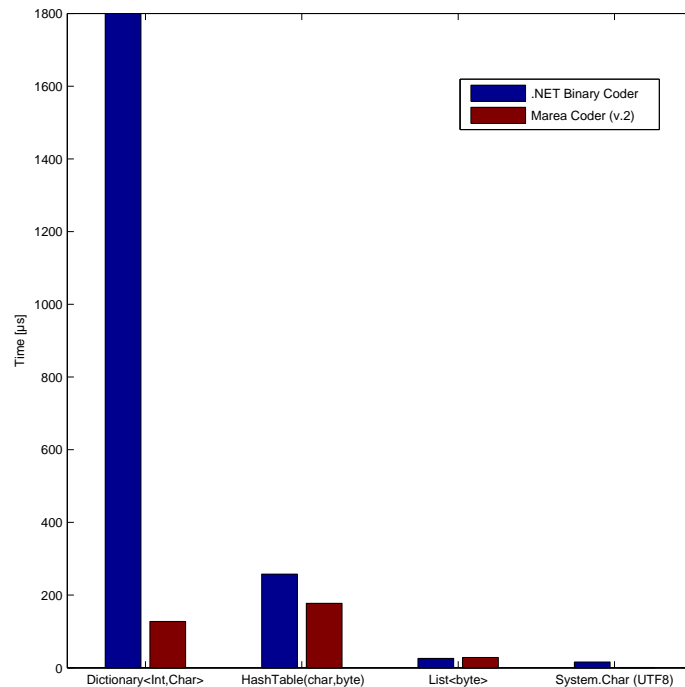

Figure 2.5: Encoder layer implementations: Total serialization and deserialization time for new supported types

Type	.NET Binary Coder	MAREA Coder (v.2)
	Total Serialization and Deserialization Time (μs)	
Dictionary<Int,Char>	1799.6832	127.3783
HashTable(char,byte)	257.7913	177.4298
List<byte>	26.0283	28.5439
System.Char (UTF8)	15.9436	0.7926

Table 2.6: Encoder layer implementations: Total serialization and deserialization time for new supported types

2.5 Transport Layer

The transport layer provides communication facilities to send and receive data (byte sequences) from the network. This layer provides transfer of data with a certain degree of transparency supporting the two most common transport protocols: TCP and UDP.

TCP transport guarantees reliable end-to-end connection oriented communications. This type of connections require a handshake mechanism in order to negotiate the terms of the connection. The exchange of segments related with this mechanism can adversely affect the performance.

This problem can be resolved by reusing the established connections instead of opening new TCP connections. The use of persistent connections results in less network traffic, use less time in order to establish new connections and allows the TCP protocol to work more efficiently.

Each MAREA protocol message is sent by TCP transport adding previously a magic number to the corresponding byte sequence representation of itself. This magic number consists in a synchronization header of 3 bytes followed by an integer (4 bytes) to specify the payload length. The first 3 bytes are use to indicate that the data is synchronized. If this first 3 bytes do not correspond to the expected ones, the transport layer detects that an error condition has happened.

UDP transport provides unreliable (best-effort) datagram communications. In this type of transport, as the opposite of TCP transport, one datagram socket is opened and closed for the dispatching of each message.

2.5.1 Drawbacks

MAREA old transport layer model use the .NET synchronous socket API in order to implement transports. The blocking mode of it set of calls require different threads to accept connections and perform socket I/O operations.

The use of one thread for each individual connection is non-scalable, especially in Windows systems. The management of a large number of threads is highly inefficient due

to the ineffectiveness of the scheduler to determine which thread should be receiving the processor time.

The memory overhead is also a handicap. For instance, in Windows the default memory overhead is 1 MB for both native and Common Language Runtime threads.

2.5.2 Improvements

The scalability performance issue can be solved by using asynchronous sockets. Asynchronous I/O operations alleviate the need to create and manage threads [9]. This type of sockets use threads internally at the OS Level, which is much faster.

Asynchronous sockets implement specific methods that use the AsyncCallback class to call completion methods to the following operations: send, receive, connect, accept. Callbacks allow the application to continue processing other events while network operations are been executed.

Two new TCP and UDP asynchronous transport modes have been added in transport layer in order to improve the scalability and the performance.

2.5.3 Results

Synchronous and asynchronous UDP transports have been compared using a round-trip isolated test. The figure 2.6 and the table 2.7 present the mean round-trip times during an echo request/response test with simultaneous transports. The mean round-trip time has been calculated according to the average value of the round-trip time of each simultaneous transport.
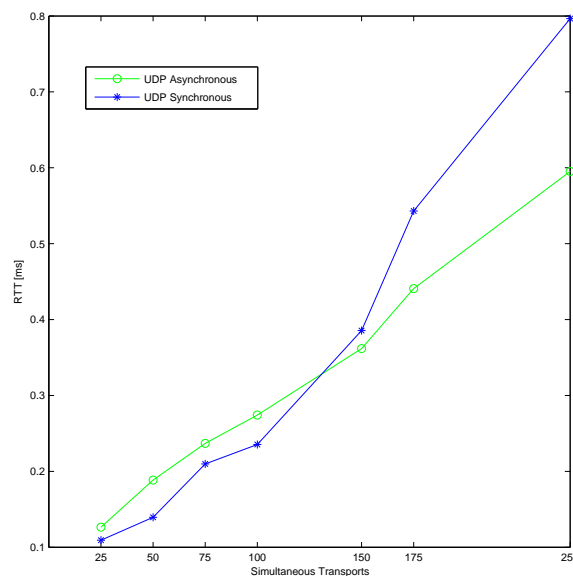

Figure 2.6: Mean round-trip times for an echo test using isolated synchronous and synchronous UDP transports: 1000 Bytes, 10000 times, 100 Hz

UDP Transports	RTT (ms)	
	Asynchronous	Synchronous
25	0.1265	0.1092
50	0.1885	0.1395
75	0.2368	0.2097
100	0.2742	0.2354
150	0.3617	0.38550
175	0.4409	0.5431
250	0.5952	0.7966

Table 2.7: Mean round-trip times for an echo test using isolated synchronous and synchronous UDP transports: 1000 Bytes, 10000 times, 100 Hz

According to the results, the round-trip time tends to grow exponentially with the number of simultaneous connections. The round-trip becomes lower in asynchronous mode, in comparison to synchronous mode, from around 150 simultaneous transports.

2.6   MAREA 1 Network Backport

The whole new MAREA 2 network architecture has been backported to MAREA 1 in order to ensure its proper functioning. Backporting is the action of taking a certain software modification (patch) and applying it to an older version of the software than it was initially created for [10]. This software backport also provides a fair and realistic way to compare the performance between the old and new network architecture.

2.6.1   Results

MAREA 1 and MAREA 1 network backport have been compared using a round-trip test. The figure 2.7 presents the round trip time distribution during an echo request/response test with two MAREA instances. Each instance runs a different service which sends or responds to the message.

The test has been executed 10000 times to send variables (UDP) with a total payload of 1000 bytes and frequency of 100 Hz. The mean round-trip time is 0.2399 and 0.433 ms for MAREA 1 network backport and MAREA 1. The standard deviation is 0.1756 and 0.3813 ms respectively.

The same test has been done using events (TCP). The mean round-trip time is 0.3501 and 0.6564 ms for MAREA 1 network backport and MAREA 1.The standard deviation is 0.2206 and 0.4061 ms respectively.
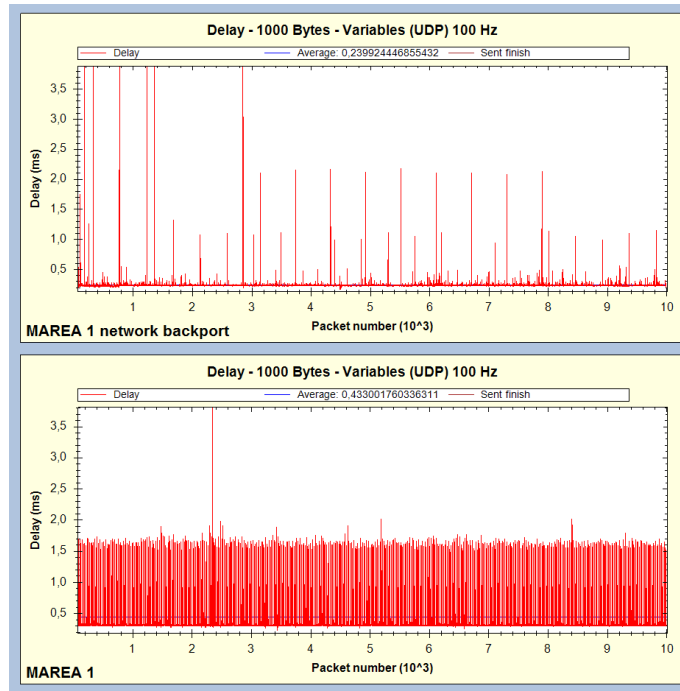
Figure 2.7: Round-trip time distribution for an echo test of MAREA 1 and MAREA 1 network backport using variable primitives: 1000 Bytes, 10000 packets, 100 Hz

2.7 Comparison with MAREA 1 Network Architecture

MAREA 1 network system architecture consist of three main sublayers: encoder, transport and Lane Manager.
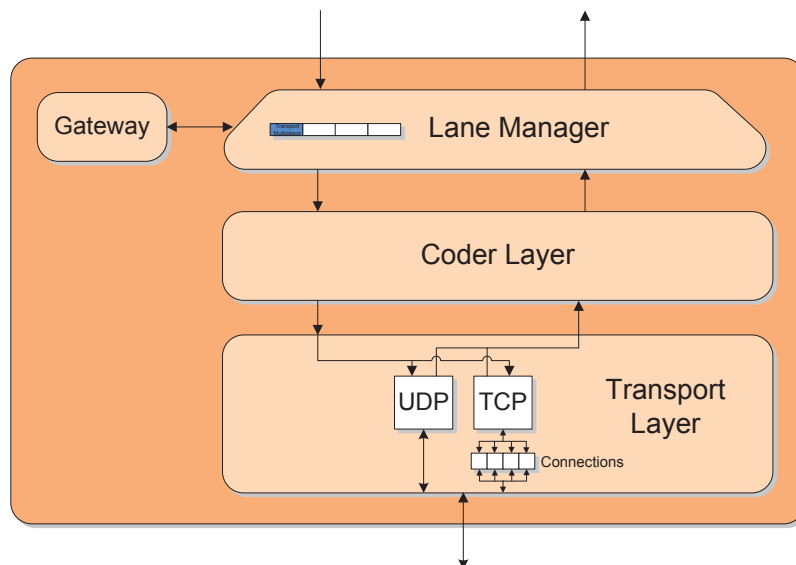

Figure 2.8: MAREA 1 network layer architecture

One of the main differences between the old and new MAREA network designs is the top entry point of the architecture which is called Lane Manager. This component is the responsible to control which transports and encoders are used by the middleware at a particular moment. In contrast, the Router is the responsible for performing this task in the new architecture.

As opposite of the new architecture, network lanes (subsection 2.2.1) are not a set of references to the bindings establish between the different network architecture elements. Instead of this, there are object references (copies) to these elements.

The main idea of taking out the Lane Manager in the new architecture is to simplify the design and improve the speed by removing unnecessary repeated searches to the lane, which in normal conditions is always the same. Every time a message is sent in MAREA 1 network architecture, a search has to be done into dictionary in order to get the default lane.
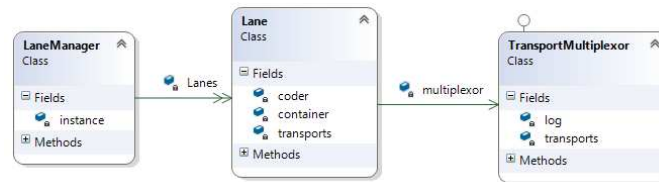

Figure 2.9: Lane Manager class diagram

The new design does not use a dictionary in order to avoid slowdowns in the performance. Instead of this, the network layer use only one output and input lane for the outgoing and incoming data respectively. These lanes are reused while the network continues to work properly.

The service container, which is the immediate upper layer, is the responsible for keeping and providing the hint to the network layer every time. Only if a network change happens, new lanes are demanded to the Router. This solution is more complex but reduces the acquisition time of the lanes.

In the transport sublayer, lanes also take profit of its new approach by taking reference directly to the TCP connection which is required in a particular moment instead of searching it in a dictionary.

Another big difference between the two architectures is the sublayer architecture design. For one hand, the MAREA 2 network sublayers (figure 2.1) are uniform. In MAREA 1 the network sublayers (figure 2.8) are inconsistent because every single sublayer presents different interfaces in order to communicate with to the upper and lower sublayers.

On the other hand, in the old architecture every sublayer is dependent of the upper and lower one because it has to keep a reference in order to communicate with them. The new sublayer architecture is more independent and flexible, because the responsibility of manage the bindings between the different sublayers is delegated to the network lanes instead of the sublayers by itself. Another difference, that has been mentioned before, is the pooling mechanism used in the new architecture which is explained in section 2.3.

In MAREA 1, the Gateway module is on charge of interconnect networks that use different types or protocols and architectures. Its main goal is the translation of the source network protocol to the destination network protocol, and vice versa, in order to allow the communication between them. In MAREA 2, this task is accomplished by the Router.

2.8 Conclusions

Network layer provides an optimized, modular and reusable usage of the network capabilities. MAREA network system architecture consist of two main sublayers: encoder and transport. The Router component together with the network lane approach builds a highly reconfigurable and adaptive architecture.

Most of the optimization work is focused in components of the network system architecture: encoder and transports, which are the most delay-sensitive layers of the whole architecture. The global performance of network sublayers has been improved implementing the following new features:

- A serialization code generation utility called MAREAGen which generates classes automatically with methods to serialize and deserialize MAREA messages in order to take advantage of the custom serialization benefits.

- A memory pooling mechanism to minimize garbage collection interruptions.

- New transport implementations with asynchronous sockets which are more scalable than traditional synchronous sockets.

CHAPTER 3. SERVICE CONTAINER

The first part of this chapter sketches out the new version of the service container. Then, the Service Manager and remote services are presented. The last sections present some services used by the middleware, and make a brief comparison between the previous and new version of the service container.

3.1 Architecture

Service container, which is the highest level layer on the MAREA stack, consist of two main components: protocol layer and Service Manager.

Protocol layer controls the exchange of MAREA protocol messages in order to discover and implement the communication primitives of remote services. This layer implements three different protocols: discovery, publish-subscribe and remote procedure call, which are detailed in chapter 4.
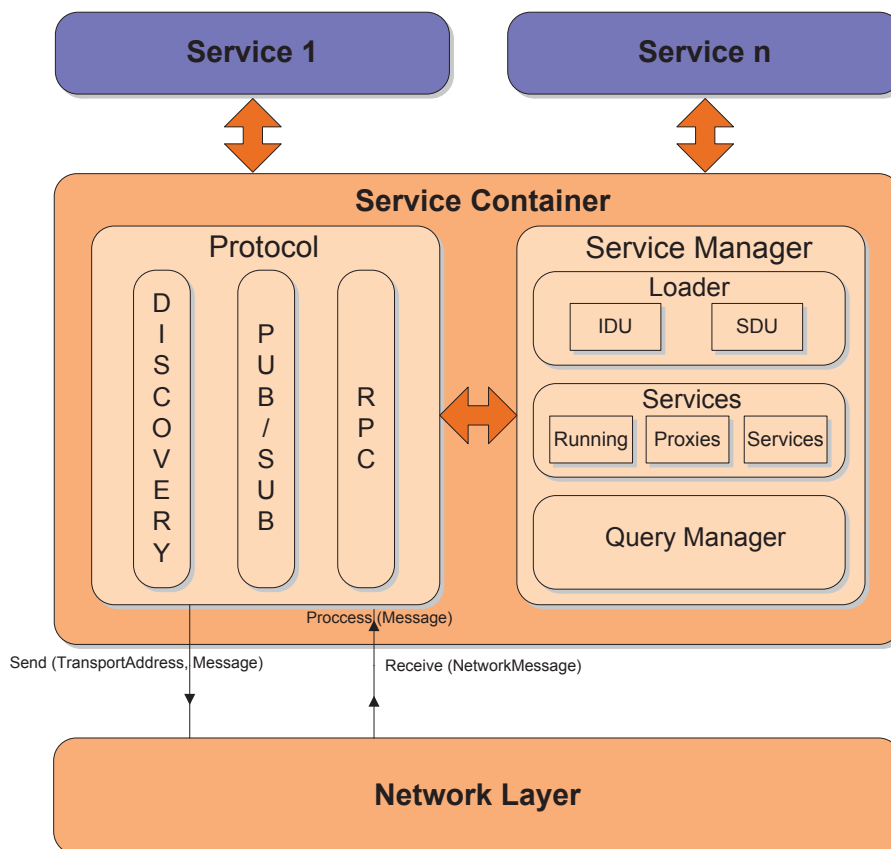

Figure 3.1: A high level view of service container

Service Manager is on charge of control the startup and shutdown the services at any moment during MAREA execution. To perform this task the service manger provides the

following set of service collections:

- **Running:** Contains the local services that have been started and are actually running in the service container.

- **Proxies:** This type of services are proxy objects that could represent a remote service or set of services selected by a query.

- **Services:** This collection contains references to all the services (running and proxies).

Services operate following the Inversion Of Control paradigm. The term Inversion of Control (IoC) is a computer programming technique wherein the flow of the control of an application is inverted. Rather than a caller deciding how to use an object, in this technique, the object called decides when and how to answer the caller, so the caller is not in charge of controlling the main flow of the application [11].

Services do not provide a main() method that starts the ball rolling and procedurally calls methods to send and receive. Instead, service container is responsible for instantiating running and controlling the entire life cycle of services. Services only allow to specify different configuration aspects like what to receive, from whom, and how to process it.

Another of the functions regarding service management is service loading. This task is accomplished dynamically at the beginning of the middleware's execution. All MAREA services are loaded in such a way that information about the service description (IDU) and implementation (SDU) is cached.

The Query Manager provides methods to manage and search services that match with queries. This component also provides some functionalities for the creation and retrieval of query proxies used to implement the naming service.

3.1.1   Relationship Between Service Container And Network Layer

As shown in figure 3.1, service container provides to different methods to receive and send the incoming and outgoing protocol messages.

First, a NetworkMessage entity is passed as a parameter in Receive method. As explained in section 2.1, the NetworkMessage contains the protocol message inside the field called Object. In this way, every time a NetworkMessage is received, the service container process the protocol message inside of it, and passes it subsequently to the specific protocol through the corresponding Process method. In addition, during the protocol message serialization stage, the encoder layer is responsible for setting the field Id of the NetworkMessage entity according on the protocol message type. This identifier is also reused by the service container in order to address the received protocol message to the specific Process protocol method. Notice that, service container Receive function is the last method inside the network input lane, which is a multicast delegate (table 2.1).

On the other hand, Send method is called from the protocol layer to transparently pass MAREA protocol messages from the protocol layer to the network layer.

3.1.2   Relationship Between Service Container And Protocol Layer

As explained in section 2.4.3, MAREA protocol message identifiers are assigned by MAREAGen with values from 0 to 63 in order to reuse them in the service container (table 2.4).

The service container contains an array of delegates used to process each MAREA protocol message according to its identifier. The position inside of the array represents the protocol message identifier, while the object stored inside is the delegate. This delegate basically points to the Process method used to handle the specific incoming MAREA messages according to the protocol type (discovery, publish-subscribe or remote procedure call protocol).

Service container provides two different methods to add and remove the delegates from the array. These methods are automatically called inside the start and stop methods of the specific protocol.

The final aim of this approach is to extend the use of delegates in the service container in order to reduce the coupling and improve maintainability in the upper layers. This new design also reduces the degree of complexity of adding more protocols in the service container.

3.2   Service Manager

The Service Manager does basically two different functions that are going to be described in the following subsections: service loading and service start up and shutdown.

3.2.1   Service Loading

Service Manager is responsible for caching information about the interface (IDU) and implementation (SDU) of services during the system startup. This information is necessary to manage services and the communication primitives during the middleware execution.

Service Manager searches for services in assemblies in order to carry out the service loading. One of the tools used to manage assemblies, types and namespaces in MAREA is the assemblies manager. This component, which is also used used by MAREAGen, has been implemented as a cache with the idea to improve the performance.

3.2.2   Service Start Up And Shutdown

Once the service loading has been done, the Service Manager reads an XML which includes the list of services that will be automatically started by the middleware. However, any MAREA service could be started or stopped subsequently while the interface and implementation of the service exist in the SDU and IDU collections.

The following subsection explains one of the steps accomplished during the service startup: the creation of communication primitives.

3.2.2.1   Communication primitives management

Service Manager is responsible for creating and getting variable and event primitives from services. As shown in listings 1.2 and 1.1 this type of primitives are implemented using generics.

Generics make possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by the programmer. To create generic type instances at run time is necessary to use reflection.

On the one hand, reflection provides extensibility to applications by allowing them to see their own inner workings. But on the other hand, reflection is a slow and expensive process. The proposed solution to solve this problem is to implement our own cache on top of the one that exists in the .NET Framework (Pobar, 2005)[12].

The attributes and metadata of communication primitives is stored together with the specific implementation of the service (SDU). The idea is to minimize the reflection operations and execute them at the service loading phase, during the middleware startup process.

Variables and events provide a common interface to control and manage primitives. For one hand, this interface implements methods to subscribe and unsubscribe the primitive to specific methods in order to receive or not new values of the communication primitive using a callback pattern. Notice that these two operations are executed when services are started and stopped (listing 1.4)

Subscribe and unsubscribe functionalities are used to add or remove a method from the invocation list of a multicast delegate respectively. This specific method has to accomplish with the following signature: a string which represent the address of the primitive and an object, of the same type as the generic type of the primitive, which carries new primitive values (see AmperageChanged and LowBatteryChanged BatteryManager methods in listing 1.4).

For the other hand, this interface provides a Notify method to inform to all the subscribed services about the new data. When the Notify method is called all the methods inside the delegate's invocation list are fired (see Run Battery method in listing 1.3).

The figure 3.2 shows the interaction between the communication primitives of one Battery-Manager and three Battery services. The BatteryManager is consuming the Amperage variable and the Low Battery event from all the Batteries of the current subsystem. Notice

that the Battery service with the address EC-UP/IP2/Bat2/Battery is a remote proxy which represent a remote service (node field info is different from the other services).
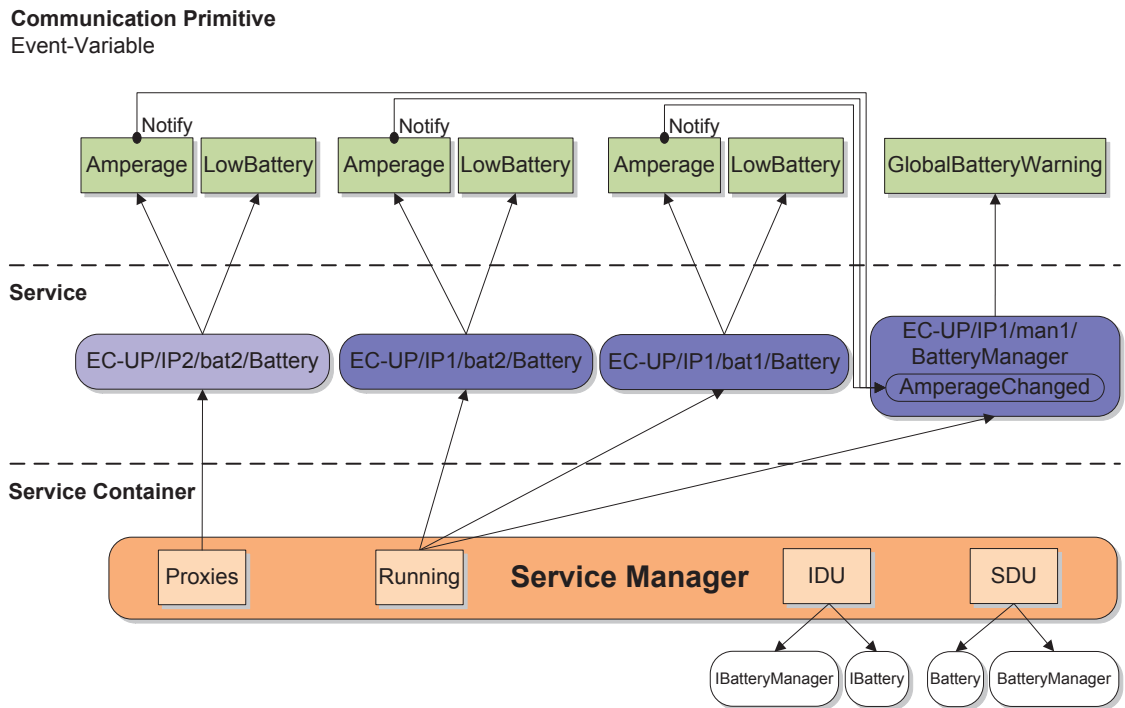

Figure 3.2: Interaction between communication primitives and services from Service Manager's point of view in a battery management system

3.3 Proxy And Remote Consumer Services

The following section describes some of the services involved in the communication between remote consumers and proxy services.

Figure 3.3 shows the scenario of a battery management system with four service containers. In this example, blue color is used for representing services that are running in service containers, while light blue an grey colors are used for representing remote producer proxies (section 3.3.1.1) and remote consumers (section 3.3.1.2) respectively. Each container is running one service: one Battery and three BatteryManager.

Proxies and remote consumers services are used to share communication primitives between services that are actually running in different service containers. The following subsections describe both proxy and remote consumer services.
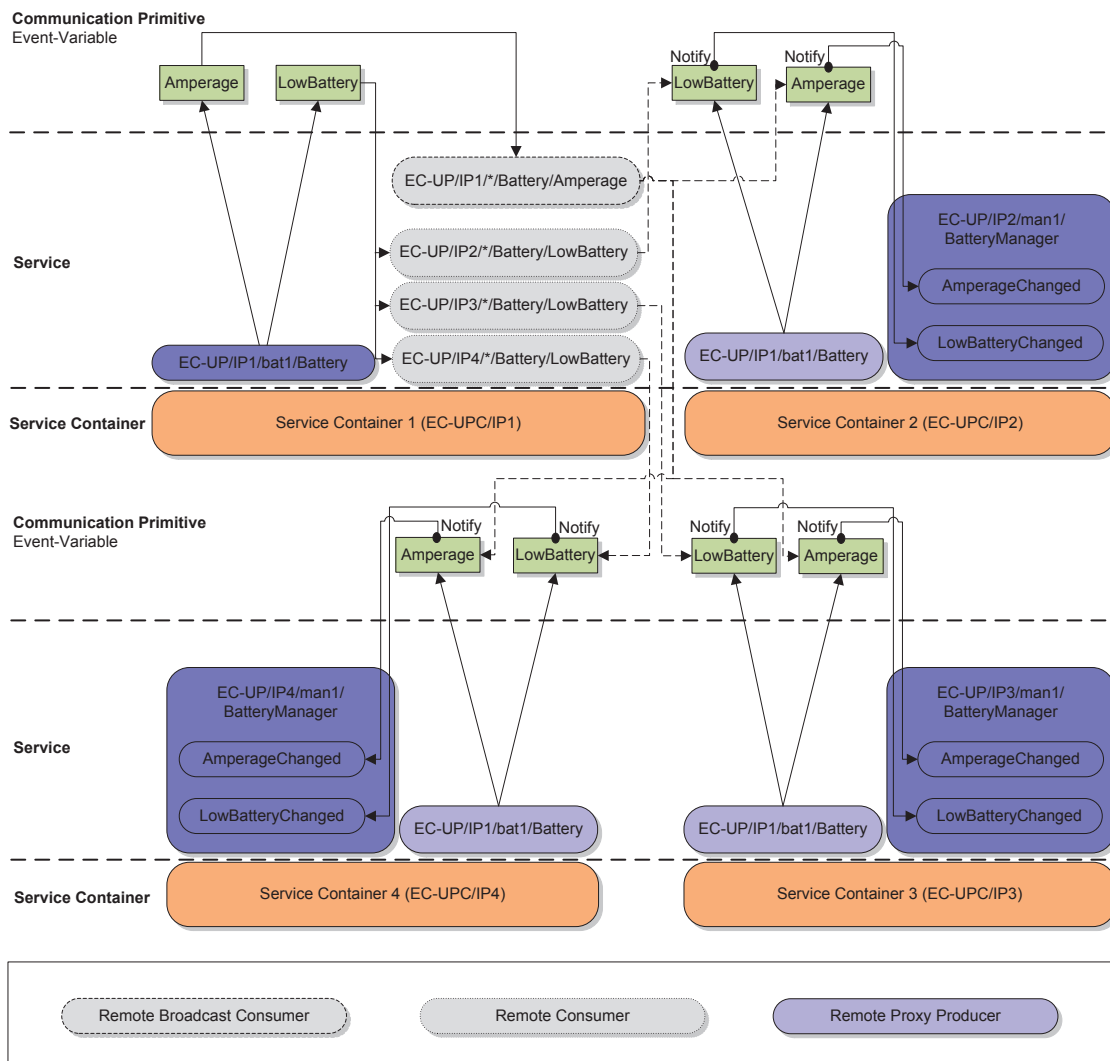
Figure 3.3: Interaction between communication primitives and remote proxies in a battery management system

Notice that, the communication between the different service containers is simplified (dashed line). In a real scenario the information flows downward and upward the network layer. Also some details, like BatteryManager's event called GlobalBatteryWarning, are passed over.

3.3.1 Proxy Services

Proxy services are objects that could represent a remote service or set of services selected by a query. This type of services implement the same interface as the represented service (IDU) and consequently, hold its own set of communication primitives.

The proxy approach based on encapsulation makes the code orthogonal and maintainable and provides location transparency between services. There is a more detailed explana-

tion of the features each type of proxy in the following subsections.

Each MAREA service needs to implement specific remote producer and query proxies. The source code of proxies is automatically generated by MAREAGen together with the template engine StringTemplate (appendix A.1.6.5), with the idea of simplify service development to end user programmers.

3.3.1.1 Remote Producer Proxy

Remote producer proxies are used to represent remote services, which is the same as services that have been started in a different service container than the current one.

The implementation of this type of proxies is quite simple because, as mentioned before, only includes the set of communication primitives held by the interface (IDU) of the remote represented service. In this type of proxy, communication primitives are created following lazy initialization. This technique defers object creation until the object is first used. Lazy initialization is primarily used to improve performance, avoid wasteful computation, and reduce program memory requirements [13].

As shown in figure 3.3, services that are consumers of communication primitives from a remote service should subscribe primitives from the remote producer proxy to local methods, that will act as a callbacks. Notice that, this behavior is the same as if the remote producer proxy and the consumer service were local services.

3.3.1.2 Query Proxy

Query proxies are used to represent a set of services selected by a query. This type of proxies also use lazy initialization to create the communication primitives held by the interface (IDU) of the represented service.
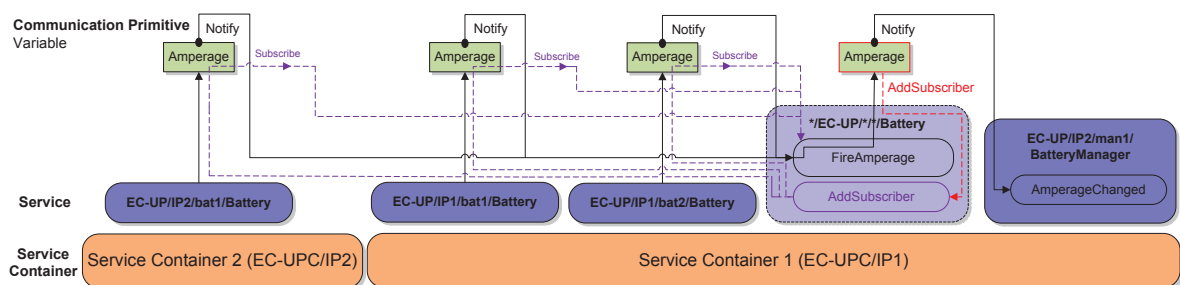

Figure 3.4: Interaction between communication primitives and a query proxy in a battery management system

Figure 3.3 shows the scenario of a battery management system with two service containers. Service container 1 is running two Battery and one Battery Manager services, while service container 2 is running a Battery service. The service in light blue color in

service container 1 is a proxy that represents all the services that match with the query */EC-UP/*/*/Battery. To simplify the figure the remote proxy corresponding to the service EC-UP/IP2/bat1/Battery, from service container 2, has been omitted in service container 1. Some other details of the battery management system are also passed over in figure 3.4, for instance, LowBattery and GlobalBatteryWarning events from Battery and Battery Manager services are not included.

As shown in listing 3.1 and figure 3.4, when a variable or and event primitive is created for the first time the method RegisterSubscriber from the specific primitive is also called. This method registers a delegate that points to AddSubscriber method contained in the query proxy (red dashed line in figure 3.4).

Notice that, this delegate is different from the one used to subscribe primitives to methods in order to notify new data values from the primitive. In fact, this delegate is automatically fired when a service subscribes to any of the communication primitives hold by the query proxy.

Listing 3.1: Query proxy variable lazy initialization example

```
private Variable<double> _amperage;
public Variable<double> amperage
{
 get
 {
  if (_amperage == null)
  {
   _amperage = container.CreatePrimitive<Variable<double>>(id, "amperage");
   ((Primitive)_amperage).RegisterSubscriber(AddSubscriber);
  }
  return _amperage;
 }
}
```

First, the AddSubscriber method looks for all services that match the query. Then, the primitive (variable or event) of each service that matches the query is subscribed to a specific method of the query proxy (FireAmperage method in figure 3.4). This method receives the values from the primitives, of those services that match with the query, and propagates them through the primitive of the query proxy. The propagation is accomplished by the Notify method of the query proxy primitive. The subscription process of the primitives that match the query to a specific method of the query proxy is represented with a purple dashed line in figure 3.4.

3.3.2   Remote Consumer Services

The purpose of remote consumer services is to transparently forward the values of each variable and event communication primitive through the network. Remote consumer services, in contrast with proxy services, do not implement the interface of the represented service. In this case, one remote consumer is needed for each primitive (services in grey in container 1, figure 3.3).

There exist two different types of remote consumers according to the type of primitive used:

- **Remote consumer:** This service is used to transparently forward the values of each event through the network. As shown in figure 3.3, one remote consumer is created for each of the services that will consume a specific event (remote consumers EC-UP/IP2/*/Battery/LowBattery, EC-UP/IP3/*/Battery/LowBattery and EC-UP/IP4/*/Battery/LowBattery in service container 1).

- **Remote broadcast consumer:** This service is used to transparently forward the values of each variable through the network. As shown in figure 3.3, one single remote broadcast consumer service is created independently the number of the services that will consume this specific variable (remote broadcast consumer EC-UP/IP1/*/Battery/Amperage in service container 1). This is implemented in this way because variables are sent via broadcast. In addition, remote broadcast consumer implementation includes a list to control all the services that are consuming the variable.

3.4   Services

The following section introduces three different services used to execute and have access to some of the features and functionalities of MAREA middleware.

3.4.1   Node Manager

One of the main functions of Node Manager is to control and manage services from remote containers. This task is accomplished using remote procedure calls. Node manager provides methods to remotely start and stop services, get running and available services, start and stop the containers, etc.

This service is loaded and started by default in service containers. Other future features related with the node management should be included in this service.

3.4.2   MAREA Console

MAREA provides to end-users a specific console based service to manage and monitor the service container. MAREA Console works along with the Node Manager service to interact with other containers that are available in the network.

The main features and functionalities that are available from the MAREA Console are:

- **Service management:** Starts and stops new instances of a service in a given container.

- **Services information:** Shows all the running and available services in a given container.

- **Debug information:** Shows the debug messages generated by the services (e.g. variables and events values, remote procedure call results, etc.).

- **Configuration:** Gets and sets some configuration parameters (e.g. MAREA default subsystem).

- **Middleware information:** Shows information about the middleware: description, version, company, etc.

- **Command help support**

Figure 3.5: MAREA Console service

The major difference between the old and new version of the MAREA Console is the tool and interface used to implement it. The previous version of this service was implemented by using a .NET console application and providing and alfa-numerical configuration menu interface to get access to the different features.

The new version of MAREA Console has been implemented with the Thorn utility (appendix A.1.6.4). This tool is very useful to build quickly command line interface applications with a bash prompt appearance.

3.4.3   MAREA GUI

The aim of MAREA GUI service is to provide access to the same features and functionalities used by the MAREA Console using a GUI (graphical user interface) instead of a console.

MAREA GUI has been implemented during the service container developing stage only for debugging purposes. The future idea is to build a more user-friendly GUI based on a tree structure in order to navigate through the different levels of the naming scheme hierarchy.
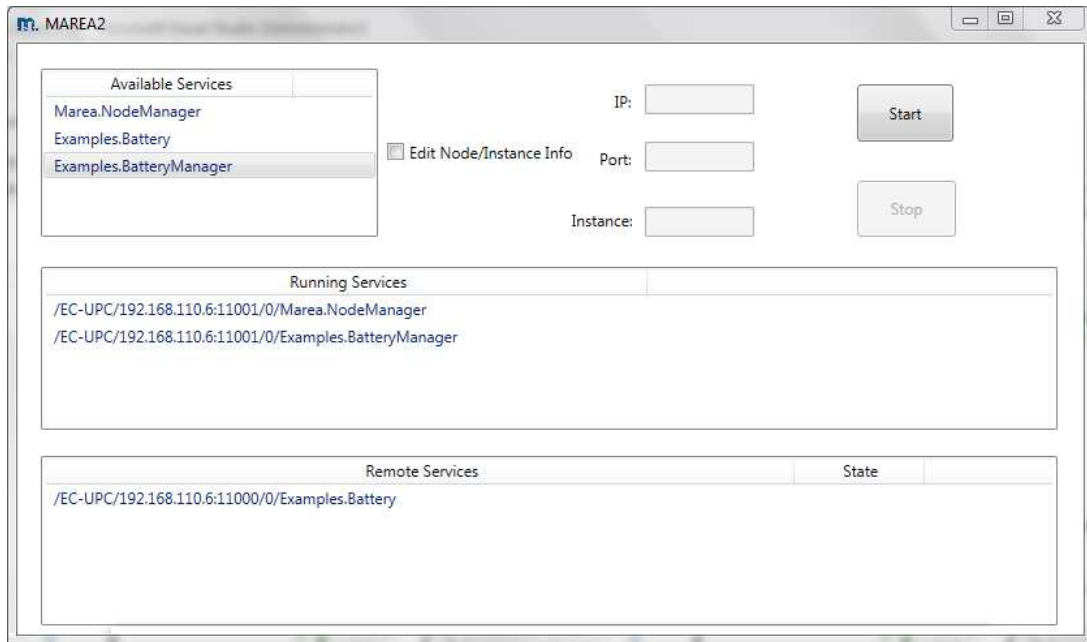
Figure 3.6: MAREA GUI service

3.5 Comparison with MAREA 1 Service Container Architecture

A noticeable difference between the new and old service container is the existence protocol layer. In MAREA 1, the features and functionalities carry out by this layer are directly accomplished by the service container. The new version of the service container provides an exclusive layer to process and send MAREA messages according to the communication primitive used.

The new service container includes the following new features and enhancements:

- New service implementation which accomplishes with the deployment unit concept.

- A naming service based on location independent service identifiers used to find, share and access services and its communication primitives hiding the network complexity.

The new design also makes special emphasis on making the code related with the communication primitives orthogonal. For instance, the previous version of the service container, in contrast with the new one, provides different managers to control each communication primitive (variables, events, remote invocation, file-based data transfer).

In this new version, the communication primitives are also simplified, because they keep delegates instead of lists of publishers and consumers.

3.6 Conclusions

The service container consist of two main components: protocol layer and Service Manager. Service Manager is on charge of service loading and service start up and shutdown. Protocol layer controls the exchange of MAREA protocol messages in order to discover and subscribe the communication primitives of remote services. The implementation and internal details of the protocol layer are presented in the next chapter.

MAREA provides to end users a console an a GUI service to manage and monitor the service container. The service container implements services following the service deployment approach and interacts with them following the Inversion Of Control paradigm.

Proxies and remote consumers services are used to share communication primitives between services that are running in different service containers.

CHAPTER 4. PROTOCOL LAYER

This chapter introduces the different protocols used by the middleware to dynamically discover remote services (those which reside in a different service containers) and consume their communication primitives. The different type of messages and some the most important features of each protocol are presented in the following subsections. Some improvements made in all the whole protocol architecture are introduced at the end of this chapter.

4.1 Discovery Protocol

The main purpose of this protocol is to discover and advertise services using a dynamic and non-centralized mechanism. Discovery protocol behaves actively to request services and also passively to listen service announcements by exchanging different types of discovery messages (red messages in figure 4.1).
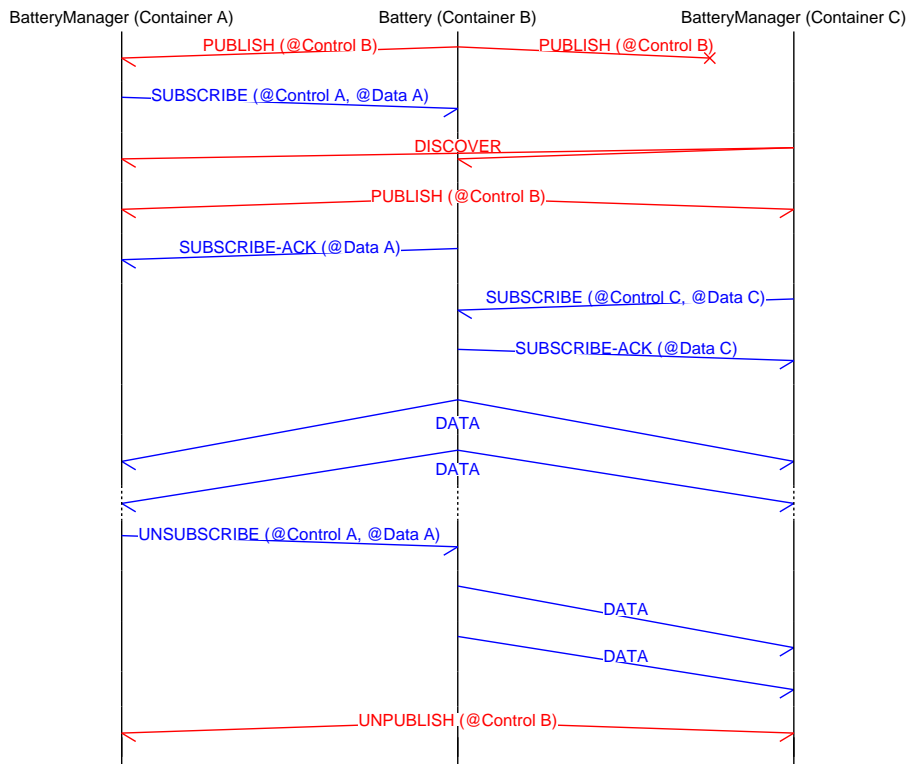

Figure 4.1: Discovery and subscription protocol message exchange bettwen a Battery and two BatteryManager services into different service containers

As shown in figure 4.1, the discovery protocol is divided in two phases: the discovery and advertisement of services (discover and publish messages) and the publish service termination (unpublish message).

In contrast to MAREA 1, this new implementation discovers and announces services instead of the communication primitives. This reduces considerably the message traffic in

the network during the initial discovery stage.

4.1.1   Messages

This subsection makes a brief description of the different types of messages used in discovery protocol.

4.1.1.1   Discover

This type of message is used to request services that are actually running in other containers. Discover message could request a single or group of services depending on the type of address is used (single or query, subsections 1.5.1.1 and 1.5.1.2 respectively). This type of message is broadcasted using the UDP protocol.

Considering a Battery is running in container B, as shown in figure 4.1, if a BatteryManager is started afterwards in container C, a discover broadcast message is sent requesting a service or a group of services according to the address contained in the LocateService attribute of the object Battery inside the BatteryManger SDU (listing 1.4).

In case of the requested service address is a query, all the containers will respond to the discover message with a publish message for each of the running services that actually match with the given query. Otherwise, if the requested service address is not a query, a single publish message will be sent by the container which actually owns the requested service.

The discovery protocol retransmits discover message periodically, for each service, as long as there is no service which publishes one or more communication primitives that are consumed by the given service.

4.1.1.2   Publish

Publish message is used to advertise a service that is actually running in a container. This type of message, which contains the control transport address of the container that offers the service, is broadcasted using the UDP protocol.

Publish messages are used in one of these two possible scenarios:

- To advertise a specific service that has been requested through a discover message (see discover and publish message exchange between container B and C in figure 4.1).

- When a service is started (see first broadcast publish message in container B of the figure 4.1).

4.1.1.3 Unpublish

This type of message is used to notify that a service has been stopped. This implies that all its communication primitives have stopped publishing information. Similarly to publish and discover messages, unpublish message is also broadcasted using the UDP protocol.

4.2 Publish-Subscribe Protocol

Publish-subscribe protocol is on charge of manage the subscriptions and data transfer of the variable and events primitives. Unlike discovery protocol, publish-subscribe protocol works at communication primitive level instead of service level.

As shown in figure 4.1, the publish-subscribe protocol (blue messages in figure 4.1) is divided in three phases: the subscription (subscribe and subscribeACK messages), the primitive data transfer (data messages) and the unsubscription (unsubscribe message).

Subscription and unsubscription messages are sent in a reliable way through TCP protocol. On the other hand, the protocol used to transport data messages depends on the type of primitive, which is TCP and UDP for events and variables respectively (section 1.3).

4.2.1 Messages

This subsection makes a brief description of the different types of messages used in publish-subscribe protocol.

4.2.1.1 Subscribe

Subscribe message is used to specify the primitive which wants a service to be subscribed. This type of message, besides the address of the primitive, specifies the control address of the service container which requires the subscription and a single or a set of data address to receive the incoming primitive.

4.2.1.2 SubscribeACK

SubscribeACK message is sent as a response of remote subscription request (subscribe message). The main purpose of this message is to negotiate the address used in the future data transfer of the primitive. This message is used to confirm the transport data address, if more than one have been offered in the request (subscribe message).

4.2.1.3 Unsubscribe

This type of message is used to end the subscription of a primitive. The aim of unsubscribe messages is to inform to the publisher, the service that holds the primitive, that a service subscribed to the primitive is not longer a consumer. As shown in figure 4.1, unsubscribe messages specify the control and data transport address of the cosumer.

4.2.1.4 Data

This message is used to send data to those services that have been subscribed to the primitive. Data messages are sent systematically due to the nature of the variables and events primitives (section 1.3), which are both used to share periodic information. This type of message contains three different fields: address, type and data value of the primitive.

4.3   Remote Procedure Call Protocol

Remote procedure call protocol implements a point-to-point synchronous communication model using the remote invocation primitive. As explained in section 1.3, remote invocation follows a request/reply pattern using a client/server model.

One of the features added in the new version of this protocol is the support of session tokens as a way to avoid reply attacks. To allow the client to assign a certain result to a previous request, the client assigns a token to each request. The server always returns this token together with the result so that the client can easily associate a result with the corresponding previous request.

4.3.1   Messages

The remote procedure call (RPC) paradigm is implemented through the exchange of call and reply function messages. Both type of messages are sent reliably by using the TCP protocol. In the figure 4.2, a Battery and BatteryManager are running in different service containers. The BatteryManager service offers the method Recharge according to its IDU definition.
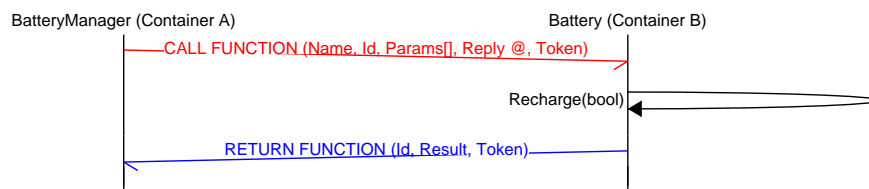

Figure 4.2: Remote procedure protocol message exchange bettwen a Battery and a BatteryManager services into different service containers

The BatteryManager (container A) starts the communication by sending a call function message to make the remote invocation call of the method Recharge of the service Battery (container B). Once the execution of this method has been done, a reply function message is returned with the results of the procedure's execution.

4.3.1.1   Call Function

The call function message contains the following fields:

- **Name:** Contains the name of the remote method.

- **Identifier:** Identifies the procedure with a random number.

- **Parameters:** Contains an array with the parameters of the call. As an inherited limitation of using the .NET Func and Action parameterized delegates, the maximum number of parameters is sixteen.

- **Reply address:** Contains the address of the procedure's caller.

- **Token:** A session token that the caller will transmit as part of the future response.

4.3.1.2   Reply Function

The reply function message contains the result of the procedure's execution together with the identifier and the token of the call function message. This message is returned to the remote call procedure requester according to the field reply address of the call function message.

4.4   Improvements

One of the problems with MAREA 1 is that all the protocols and its functionalities are included inside the service container. The goal of the new implementation of the protocol layer is to decouple the different protocols into smaller independent modules. This approach makes protocols easier to modify by decomposing them around smaller design decisions.

The different protocols have been implemented in independent classes with two different methods for each type of message of the specific protocol. The idea is to use one them to process the incoming MAREA messages from the service container and the other to build and send the MAREA messages to the service container.

Each protocol offers a start and stop method in order to make the whole protocol layer highly configurable. The different protocols could be used by calling these methods, depending on the scenario and requirements of the system. The implementation of these two methods basically adds or removes the different delegates from the array located in service container.

4.5 Conclusions

In contrast to MAREA 1, MAREA 2 implements a new component called protocol layer. Protocol layer is on charge of offering remote message delivery capabilities according to the communication protocol used for each communication primitive. Protocol layer allows to easily plug additional protocols for implementing other primitives (e.g. file-based data transfer) and other communication features.

The protocol layer implements three different protocols: discovery, publish-subscribe and RCP. Firstly, discovery protocol, as its name suggests, aims to discover and an announce services. Secondly, publish-subscribe protocol is on charge of manage the subscriptions and data transfer of the variable and events primitives. Finally, RPC protocol defines a point-to-point synchronous communication model based on a request/reply pattern in order to implement remote invocation protocol.

CHAPTER 5. CONCLUSIONS

This chapter provides a final analysis of the master thesis. The first sections present results and the different conclusions: the project conclusions and the personal conclusions. This chapter comes to an end with some of the future lines of work and the environmental impact.

5.1 Results

Although MAREA was not designed for hard real-time applications, one of the objectives of this master thesis is to optimize, evaluate and compare the performance between MAREA 1 and MAREA2 middlewares.

For this evaluation, the performance analysis consists in communicate two services that are deployed in two different service containers. The communication between them is accomplished following an echo request/response exchange-pattern. The requester service starts a timer and sends a message with a timestamp to the replier service. The replier service simply returns the message to the requester. Then, the requester calculates the round-trip time with the actual time and the timestamp of the message.

5.1.1 Round-Trip Time

The test has been executed 10000 times to send variables (UDP transport) and events (TCP transport) with a total payload of 1000 bytes and frequency of 100 Hz. The chosen frequency is 100 Hz, because in avionics the typical requirements are in the 20-100 Hz range. The test has been executed with the following three versions of MAREA middleware: MAREA 1, MAREA 1 network backport and MAREA2.

In relation to variables, the mean round-trip time is 0.8313, 0.2797 and 0.3587 ms for MAREA 1, MAREA 1 network backport and MAREA 2 respectively. In events, the mean round-trip time is 0.9422, 0.3922 and 0.6481 ms for MAREA 1, MAREA 1 network backport and MAREA 2 respectively. The table 5.1 shows the standard deviation of the RTT results. In both variables and events the mean RTT has been reduced a 56.93% and 31.21% respectively.

Middleware	Variable-RTT (ms)		Event-RTT (ms)	
	Mean	STD	Mean	STD
MAREA 1	0.8313	0.6594	0.9422	0.5453
MAREA 1 Backport	0.2797	0.2992	0.3922	0.5221
MAREA 2	0.3587	0.4081	0.6481	0.5737

Table 5.1: Mean round-trip times for an echo test using MAREA 1, MAREA 1 network backport, MAREA 2: 1000 Bytes, 10000 times, 100 Hz
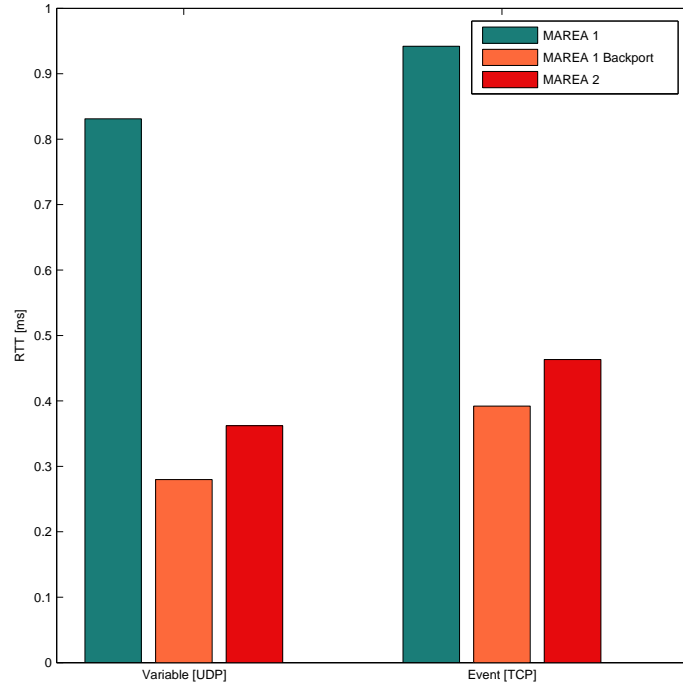
Figure 5.1: Mean round-trip times for an echo test using MAREA 1, MAREA 1 network backport, MAREA 2: 1000 Bytes, 10000 times, 100 Hz

5.1.2   Memory allocation

The number of bytes allocated by the different versions of the middleware has been also measured during the execution of the same test in the requester side. Notice that the y axis of the figure 5.2 is in logarithmic scale.

In relation to variables, the total number of bytes allocated are 15834797234, 85587107 and 139311411 bytes for MAREA 1, MAREA 1 network backport and MAREA 2 respectively. In events, the total number of bytes allocated are 14805029654 , 54609735 and 122285449 bytes for MAREA 1, MAREA 1 network backport and MAREA 2 respectively. In both variables and events the total number of bytes has been reduced a 91.2% and 99.17% respectively.

Middleware	Memory Allocation (bytes)	
	Variable	**Event**
MAREA 1	15834797234	14805029654
MAREA 1 Backport	85587107	54609735
MAREA 2	139311411	122285449

Table 5.2: Number of bytes allocated by MAREA 1, MAREA 1 network backport, MAREA 2 in an echo test (requester side): 1000 Bytes, 10000 times, 100 Hz
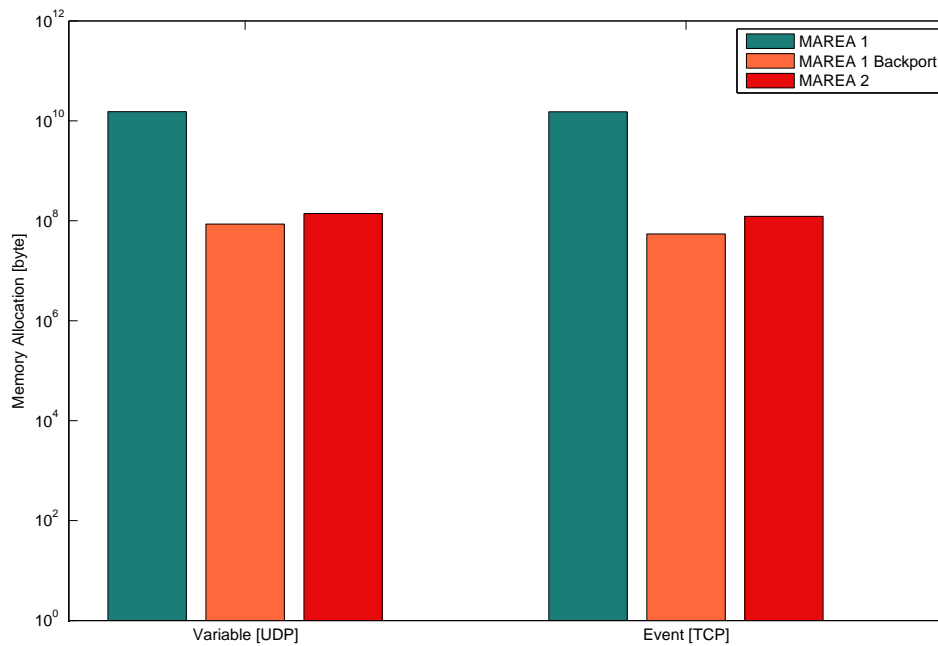
Figure 5.2: Number of bytes allocated by MAREA 1, MAREA 1 network backport, MAREA 2 in an echo test (requester side): 1000 Bytes, 10000 times, 100 Hz

In MAREA 2 the number total of bytes allocated and the mean RTT has been significantly reduced from MAREA 1, for both variables and events primitives.

The number total of bytes and the mean RTT results for MAREA 1 network backport are slightly lower compared to MAREA 2. One plausible explanation could be that in MAREA 2, the service container should manage one additional proxy service to support the queries. This additional service may introduce some delay due to the propagation of variable and event primitives. The other reason is that no specific performance optimization work has been done in the service container. In this sense, there is still room for performance improvement in the service container.

5.2   Project Conclusions

The specified objectives at the start of the project have been successfully reached.

A new version of the middleware MAREA 1, has been designed and implemented. The enhancements provided by MAREA 2 design is a more modular, flexible and reusable architecture. The new design offers a starting point for providing reflective and adaptive capabilities, and builds a highly reconfigurable system. The use of delegates both in the service container and network layer reduces the coupling and improves maintainability of the different sublayers. Based on the first feedback received from developers, at least regarding the network layer, MAREA 2 programming complexity has been reduced com-

pared to MAREA 1 middleware.

The new service implementation based on the deployment unit concept takes benefit from interfaces and the Composite Reuse Principle, one of the most powerful concepts in modern object orientated languages. Work with interfaces become much more flexible because different services can be easily switched out. In this regard, this new approach defines a SOA composed of plug-compatible services. In addition, services are loosely coupled from each other, making the entire system more adaptable to change.

One of the new features added in MAREA 2 is the support of a naming service. The implemented naming service based on dynamic name resolution allows the middleware to find, share and access services and its communication primitives hiding the network complexity. The naming service provides fault tolerance to the system (services could be replicated in different nodes for redundancy).

Most of the optimization work is focused in the components of the network layer. The global performance of this layer has been improved implementening the following features:

- MAREAGen is a custom serialization tool that generates code automatically to serialize and deseserialize entities marked as serializable. MAREAGen eliminates the need for developers to implement serializing and deserializing code and guarantees run-time type safety. This tool is also responsible for creating proxies automatically.

- A memory pool mechanism has been implemented to minimize garbage collection interruptions. The implemented pool supports the insertion and removal of NetworkMessage entities using a FIFO discipline.

- Two new TCP and UDP asynchronous transports have been added in transport layer in order to improve the scalability.

Some key performance parameter like round-trip time (RTT) and memory allocation have been analyzed in order to evaluate and compare the performance between MAREA 1 and MAREA2 middlewares. In MAREA 2, the RTT and number total of bytes allocated by the sender in an echo request/response test, between two services that are deployed in two different service containers, have been reduced considerably with respect to MAREA 1.

One of the topics covered during this master thesis is the improvement of the middleware quality with continuous integration. Building a large software systems, like middlewares, from several modules, integrating them into a working product is difficult and time consuming. Continuous integration is very useful tool used in combination with automated unit tests because it save both time and money over the lifespan of a project. Jenkins (appendix A.4.1) open-source continuous integration server and NUnit (appendix A.1.6.2) unit testing framework have been used to offer support for both continuous integration and unit testing.

5.3 Personal Conclusions

Beyond academic achievements, all the process involving this master thesis has been certainly rewarding. This project has given me a real chance to design and develop an application from beginning to end. In this sense, it has been a very rewarding challenge.

This project has allowed me to get familiar with middleware, which represents the confluence of two key areas of information technology (IT): distributed systems and component-based design and programming.

There are many skills acquired or consolidated during this time: from the initial touch-down of the MAREA middleware, reflection, inversion of control with the delegation pattern, guidelines for application performance, the orthogonality and its importance in software development, reduction of software complexity by capturing successful patterns and creating reusable components, etc.

I also have realized that the development of a project is a quite complex task and requires hard effort and dedication, but most of all a strict control of timings in order to accomplish with the established work plan.

In addition, the experience working with my tutor has been very positive too, because he leave so much leeway. The direct and close communication with him, has allowed me to fulfill with the objectives and deadlines.

5.4 Future Work

The results of this master thesis point to several interesting directions for future work:

- **Addition of new protocols:** The first protocol that should be added is a file-based data transfer protocol to implement the file communication primitive. An interesting future work-line is to implement a P2P protocol in order to support content distribution capabilities.

- **Static name resolution support:** In some specific cases services require the consumption of a specific service instance during its whole execution. In these situations is necessary to use static name resolution. In contrast to the implemented dynamic name resolution, static resolution does not offer redundancy and, consequently, fault tolerance. This means that if the provider service fails or shutdowns, the consumer cannot automatically change to another service.

- **Extend capabilities to other platforms:** One of the first steps is to extend the middleware usability to other .NET platforms like .NET Compact Framework and .NET Micro Framework, simpler versions of the .NET Framework designed to run in mobile and embedded devices. Another interesting step is to extend its use to devices like Raspberry Pi, a new minimalist computer built for the ARMv6 architecture, together with the cross software platform Mono (appendix A.3.1) and IOSharp

(appendix A.3.3). These extensions would significantly leverage the meaning of MAREA, a middleware specially designed for systems composed by a number of low-cost distributed computing devices connected by a network.

- **Performance optimization:** Most of the performance optimization work of this master thesis is focused in the network architecture, the most delay-sensitive layer of the whole architecture. In this sense, there is still a path to follow with the service container. Another future option to optimize the performance, ensure hard real time requirements, and also improve the portability to other platforms is to make a new implementation of the middleware in C++. A very interesting alternative to make a new implementation in C++, can be the use of Alternative (appendix A.3.2), a tool for easy port applications from high-level languages to native languages.

- **QoS control mechanisms:** The idea is is to implement an adaptive message QoS control in the service container that handle workload variations dynamically.

- **Software usability:** One of the significant aspects that not was covered in MAREA 1 is the usability level. Usability techniques increase user efficiency and productivity and, consequently, user satisfaction.

  One of the proposals related to this topic, is the creation of Visual Studio customized service oriented project templates. The use of templates is one of the numerous ways to boost productivity in Visual Studio by automating redundant steps. Templates provide more time to the developer to concentrate on the more challenging aspects of their work. Templates would provide a starting point for users to create MAREA service projects according to the deploy unit concept. Templates would include the files that are required for each particular project type (IDU, SDU and LDU): standard assembly references, project properties and compiler options.

  Another topic that would be convenient to take in consideration regarding to usability is end user distribution of services based on the deployment unit concept. It would be interesting to build a distribution platform of MAREA services based on NuGet (appendix A.1.1). The idea is that each service counts with an exclusive Git (appendix A.2.1) repository to host the source code and a NuGet package to host the binaries. The final goal is to provide a generic tool that enables the distribution of the source code and binaries depending on the programmer specific needs.

5.5 Environmental Impact

At last but not least it is necessary to talk about the environmental impact of the work described in this document. As can be seen from the present document, this project consists in the design and development of a software application. This has not a direct environmental benefit, but this middleware was mainly specifically designed to fulfill UAS communications and their application to the design of complex distributed UAS avionics. UAS can perform air operations that manned aviation can hardly do, with evident economic savings and environmental benefits while reducing the risk to human life.

REFERENCES

[1] Schmidt, D.C. and Schantz, R.E., "Middleware for Distributed System - Evolving the Common Structure for Network-centric Applications", *Encyclopedia of Software Eng.*, Wiley & Sons, New York, 2001. Available at `http://www.agentgroup.unimore.it/didattica/ingss/Lec_Middleware/Schmidt_Middleware.pdf` (visited 26, January, 2013).

[2] Bagula, A.B., Denko, M.K. and Zennaro, M., "Middleware for Mobile and Pervasive Services", Chap. 7 in *Handbook of mobile systems applications and services*, Taylor and Francis Group, Kumar, A. and Xie, B., pp. 248-249, Boca Raton (FL), 2012.

[3] Khan, S., Qureshi, K. and Rashid, H., "Performance Comparison of ICE, HORB, CORBA and Dot NET Remoting Middleware Technologies", *International Journal of Computer Applications*, 3(11), 15-18 (2010). Available at `http://www.ijcaonline.org/volume3/number11/pxc3871105.pdf` (visited 26, January, 2013).

[4] Loyall, J., Schantz, R., Zinky, J., et al. "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications", *Proceedings of the 21st International Conference on Distributed Computing Systems*, Mesa (AZ), 2001. Available at `http://www.cse.wustl.edu/~cdgill/PDF/ICDCS01.pdf` (visited 15, January, 2013).

[5] López, J., Royo, P., Barrado, C., Pastor, E., "Applying marea middleware to UAS communications", *In Proceedings of the AIAA Infotech@Aerospace Conference and AIAA Unmanned Unlimited Conference 2009*, Seattle (WH). Available at `http://upcommons.upc.edu/e-prints/bitstream/2117/9248/1/infotech09.pdf` (visited 15, March, 2013).

[6] López, J., "Service Oriented Architecture for Embedded (Avionics) Applications", *The PhD Program on Computer Architecture Technical School of Castelldefels Technical University of Catalonia*, Barcelona, 2011. Available at `https://dl.dropbox.com/u/2857619/thesis-small.pdf`(visited 5, October, 2013).

[7] Kiely, D., "Delegates Tutorial" in *The Microsoft Developer Network (MSDN)*. Available at `http://msdn.microsoft.com/en-us/library/aa288459(v=vs.71).aspx` (visited 15, February, 2013)

[8] Albahari, J. and Albahari B., "Serialization", Chap. 17 in *C# 5.0 in a Nutshell: The definitive reference*, O'REILLY, Roumeliotis, R., pp. 691-728, Sebastopol (CA), 2012.

[9] Kiely, D., "Get Closer to the Wire with High-Performance Sockets in .NET" in *The Microsoft Developer Network (MSDN) Magazine*. Available at `http://msdn.microsoft.com/es-es/magazine/cc300760(en-us).aspx` (visited 5, March, 2013)

[10] Books Llc, Source Wikipedia, "Software Quality: Software Crisis, Kludge, Second-System Effect, Workaround, Reliability Engineering, Fault-Tolerant System", Books Llc, Memphis (Tennessee), 2011.

[11] Garofalo, R., "Desing Patterns", Chap. 2 in *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*, O'REILLY, Jones, R., pp. 25-60, Sebastopol (CA), 2011.

[12] Pobar, J., "Reflection: Dodge Common Performance Pitfalls to Craft Speedy Applications" in *The Microsoft Developer Network (MSDN) Magazine*. Available at `http://msdn.microsoft.com/en-us/magazine/cc163759.aspx` (visited 25, August, 2013).

[13] Microsoft, "Performance: Lazy Initialization" in *The Microsoft Developer Network (MSDN)*. Available at `http://msdn.microsoft.com/en-us/library/dd997286.aspx` (visited 25, August, 2013).

APPENDICES

Title: MAREA 2. Design and Optimization of a Distributed Communications Middleware

Master Degree: Master in Science in Telecommunication Engineering & Management

Author: Santiago Pérez Fernández

Director: Juan López Rubio

Date: October 26, 2013

APPENDIX A. TOOLS

A.1   Package management system

A package management system, is a collection of software tools to automate the process of installing, upgrading, configuring, and removing software packages. It typically maintains a database of software dependencies and version information to prevent software mismatches and missing prerequisites.

A.1.1   NuGet

NuGet is a free, open source developer focused package management system for the .NET platform intent on simplifying the process of incorporating third party libraries into a .NET application during development.

The NuGet tools provide the ability to create, publish and consume packages. Each packages consists in a nupkg (NuGet package) file which contains packaged source code or libraries that can be used for any developing program components.

A.1.2   Package Management

Manage package references in projects becomes very simple with the NuGet extension, which is included by default in Visual Studio 2012. The packages can be added, updated and removed in projects using the Manage NuGet Packages dialog box.
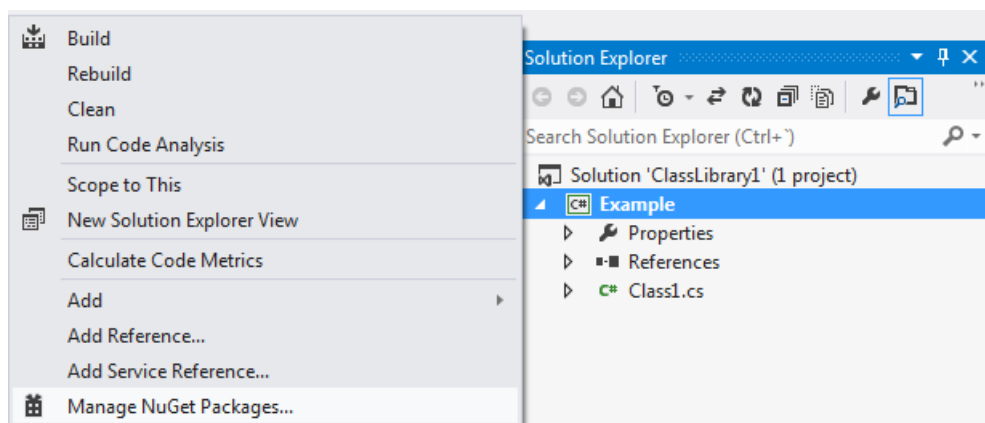
Figure A.1: Manage NuGet Package option in right click menu project
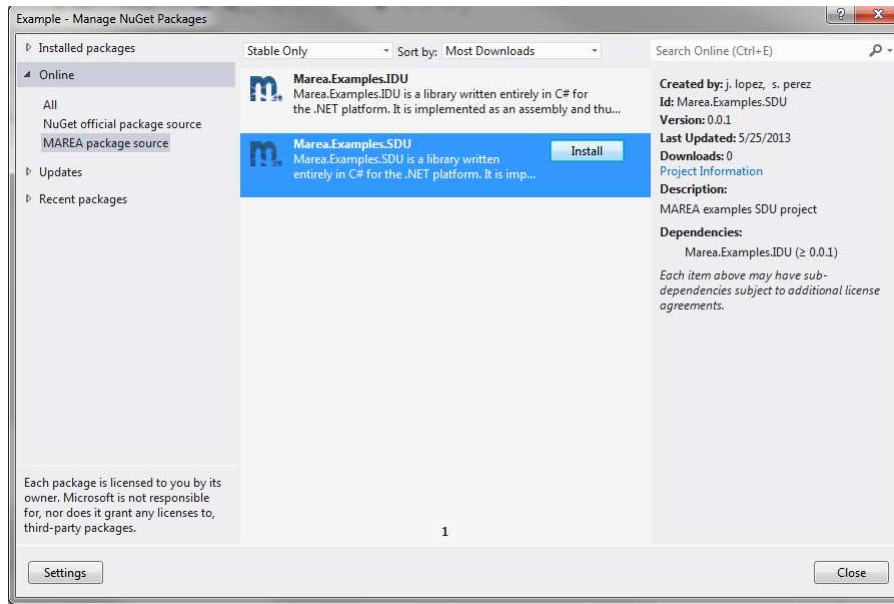
Figure A.2: MAREA service package management in Manage NuGet Packages dialog box

A.1.3 Package Creation

The first step to create a NuGet package is to configure a nuspec (NuGet specification) file. Nuspec files are manifests in XML format that specify all the settings and the package dependencies.

The metadata of the nuspec file contains the following fields: id, version, authors (collection of author elements), description, language, tags (collection), licenceUrl (Uri), projectURL (Uri), iconUrl (Uri), requrireLicenceAcceptance(boolean) and a set of dependencies. Each dependency element has the following attributes: id (the ID of a package that this package depends on) and version (the required version of the dependency package.

The figure A.3 shows an example of nuspec file. Notice that the nuspec file is in the root folder of the project.

Figure A.3: Nuget specification (nuspec) file from a MAREA service project

The following command creates a package(nupkg file) from a given project and a nuspec file:

nuget pack "ProjectName".csproj

A.1.4   Package Publication

Once the package has been created, it needs to be pushed to a NuGet server A.1.6.3. Depending on the NuGet server configuration, an access key could be required to publish and delete packages.

The following command pushes a package to a Nuget server:

nuget push "PackageName".nupkg -s "NugetServerURL" "AccessKey"

A.1.5   Package Managment Automation

An external Visual Studio tool has been created in order to facilitate to developers the creation and publication of his own packages. With this tool the developers are able to create and push services just by selecting the project and clicking the option MAREA 2 package source->Create and Push service on the Tools menu.
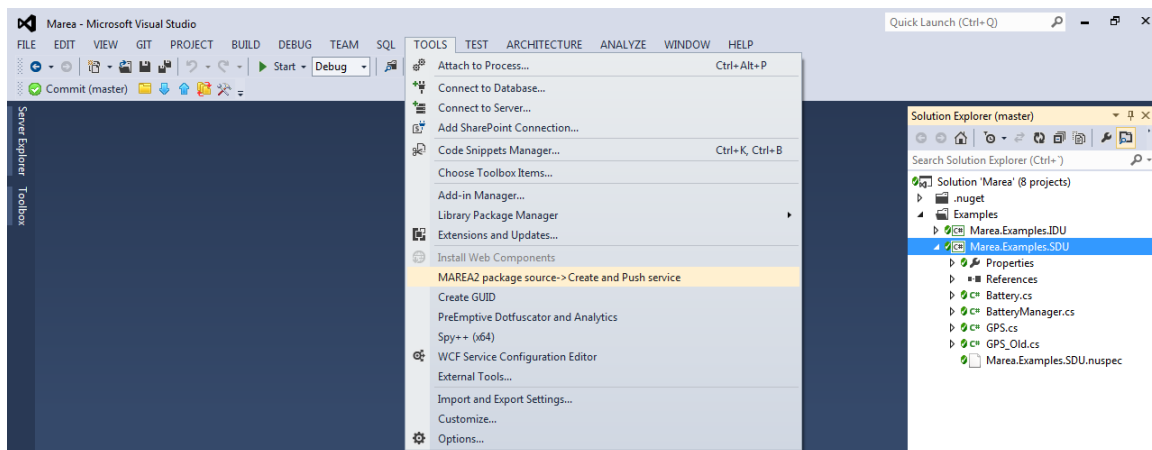
Figure A.4: Visual Studio Create and Publish package external tool

This tool has been created specifying the following command and arguments:

- **Command:** *C:/WindowsMicrosoft.NET/Framework64/v4.0.30319/MSBuild.exe*

- **Arguments:** */t:Build,Package,Publish /p:Configuration=Debug; NuGetServer=http://localhost:7073; NuGetKey=3237c377-4253-4b48-91ec-7b5457df28d5 $(ProjectDir)$(ProjectFileName)*
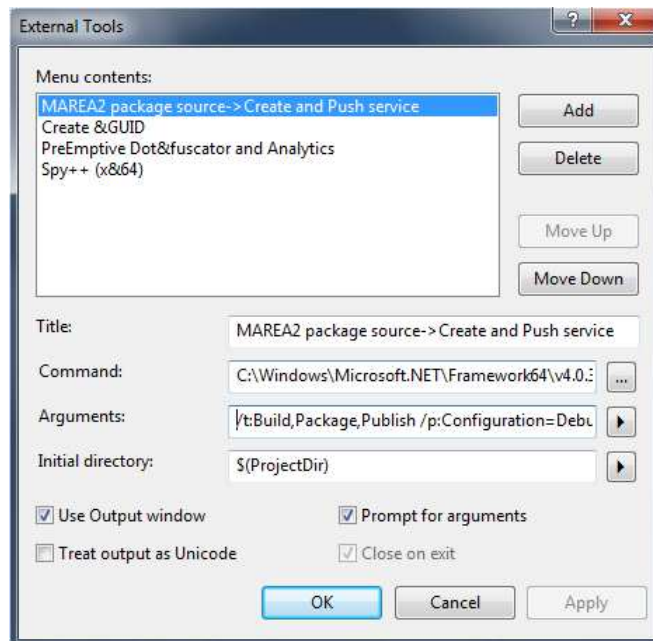
Figure A.5: Create and Publish package external tool details

A.1.6 Third Party Packages

This subsection includes the different third party libraries that have been used to develop MAREA. All of them have been obtained and installed through the NuGet package management system.

A.1.6.1 Log4net

Log4net, a port of the popular Java library log4j, is an open source library that allows .NET applications to log output to a variety of sources (e.g., console, files or SMTP). The information is logged via one or more loggers which provide a the following five logging levels: debug, information, warnings, errors, fatal.

A.1.6.2 NUnit

NUnit, a port from JUnit, is a unit-testing framework for all .NET languages. It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities.

NUnit does not support Visual Studio integration. Instead of this it provides an external program compiled either as a console app or a GUI. This program is able to run and execute the unit tests from an assembly.
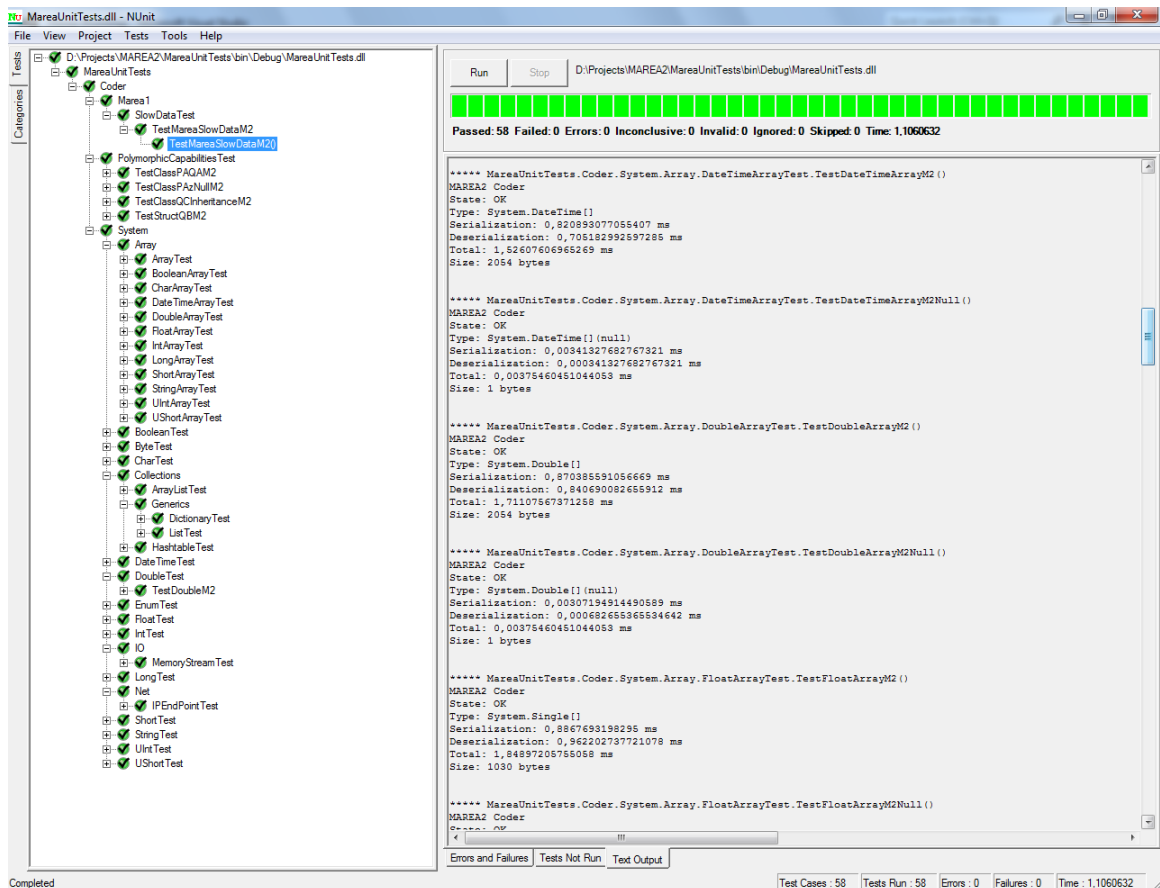
Figure A.6: MAREA unit tests executed by NUnit GUI application

This framework has been especially used to test encoder layer, naming and service management functionalities. The listing A.1 shows an example of a unit test to serialize and deserialize a double with two diferent pair of parameters.

Listing A.1: MAREA encoder layer unit test: serialization and deserialization of a double

```
private byte[] seralizedData = null;
private long start, serializeTicks, deserializeTicks;
private long  clock_freq = PerformanceTimer.Clock_freq();

[SetUp]
public void RunAfterAnyTest()
{
 serializeTicks = 0;
 deserializeTicks = 0;
}

[TestCase(0.100000234523, 0), NUnit.Framework.Description("Coder(double , System.Double)")]
[TestCase(double.MaxValue,0)]
public void TestDoubleM2(double oDouble, double rDouble)
{
 for (int i = 0; i < CoderTestsConstants.CODIFICATIONS; i++)
 {
  start = PerformanceTimer.Ticks();
  seralizedData = AdaptedMareaCoder.Send(oDouble);
  serializeTicks += PerformanceTimer.TicksDifference(start);

  start = PerformanceTimer.Ticks();
  rDouble = (double)AdaptedMareaCoder.Receive(seralizedData);
```

```
  deserializeTicks += PerformanceTimer.TicksDifference(start);
 }

 Console.WriteLine(CoderTestsConstants.MAREA2);
 Results results = ResultsManager.GetResults(serializeTicks, deserializeTicks, clock_freq,
 CoderTestsConstants.CODIFICATIONS, seralizedData.Length, rDouble.GetType().FullName);

 if (oDouble == rDouble)
 {
  Assert.True(true);
  Console.WriteLine(CoderTestsConstants.OK_STATE);
  Console.WriteLine(results.ToString());
 }
 else
 {
  Console.WriteLine(CoderTestsConstants.KO_STATE);
  Assert.True(false);
 }
}
```

A.1.6.3   Nuget Server

According to the proposed design MAREA services should be pushed as packages in an own NuGet server (like the one hosted at nuget.org). The steps to configure a NuGet server are presented bellow.

- **Create a ASP.NET Empty Web Application:** Go to the File | New | Project menu option which will bring up the new project dialog and select ASP.NET Empty Web Application.

- **Install the NuGet.Server Package:** Make Right click on the created ASP.NET Empty Web Application and select the option Manage NuGet Packages (figure A.1). Search and install the package Nuget.Server in the Manage NuGet Packages dialog box (figure A.2).

With this last step the NuGet.Server package has just converted the ASP.NET Empty Web Application into a site that is ready to serve up the package feed. To start the NuGet server build an run the ASP.NET Web Application project.

The configuration of the NuGet server can be easily modified through Web.config file. The most important parameters are:

- **packagesPath:** Specifies a custom (absolute or virtual) path for packages folder.

- **requireApiKey:** Determines if an access key is required to push/delete packages from the server.

- **apiKey:** Sets the value of the key to allow people to push/delete packages from the server.

```
<appSettings>
  <!--
          Determines if an Api Key is required to push\delete packages from the server.
  -->
  <add key="requireApiKey" value="true" />
  <!--
          Set the value here to allow people to push/delete packages from the server.
          NOTE: This is a shared key (password) for all users.
  -->
  <add key="apiKey" value="3237c377-4253-4b48-91ec-7b5457df28d5" />
  <!--
          Change the path to the packages folder. Default is ~/Packages.
          This can be a virtual or physical path.
      -->
  <add key="packagesPath" value="C:\MareaPackages" />
</appSettings>
```

Figure A.7: NuGet server application settings from Web.config file

The next step is configure the new package source in Visual Studio by clicking in the button Settings of the dial box Manage Nuget Packages. To add the new source add a name and specify the URL of the NuGet server like is shown the figure A.9. Note that the URL is http://domain/nuget/ and depends on how the site has been deployed.
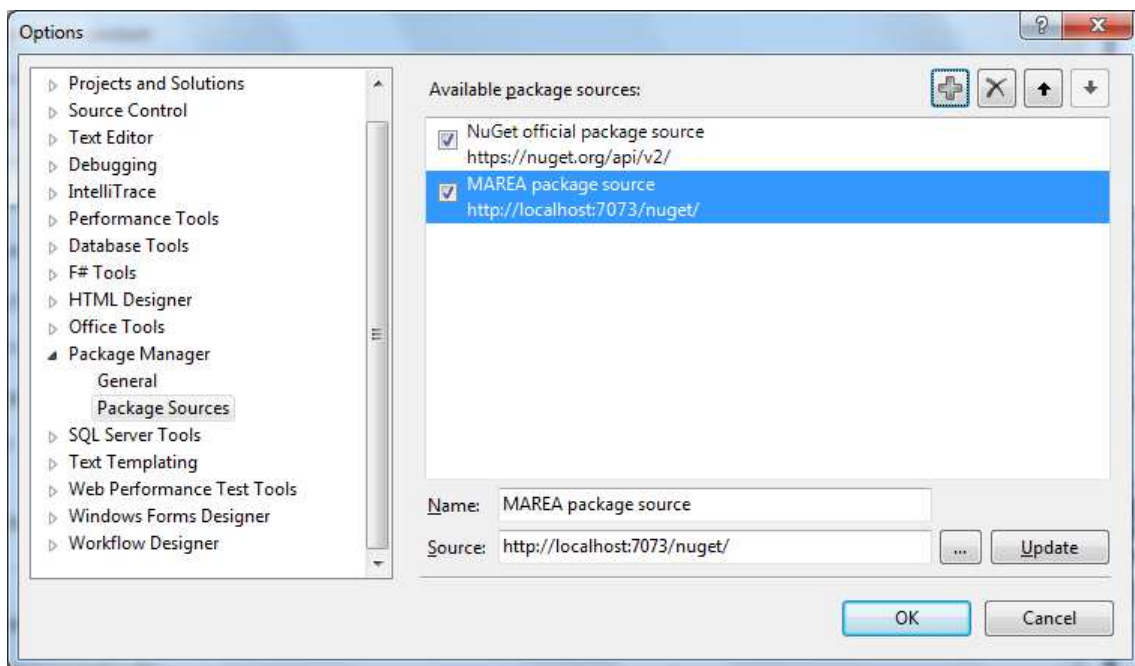
Figure A.8: Visual Studio package source configuration

The URL http://domain/nuget/Packages lists the name and description of the packages that have been uploaded to the server.
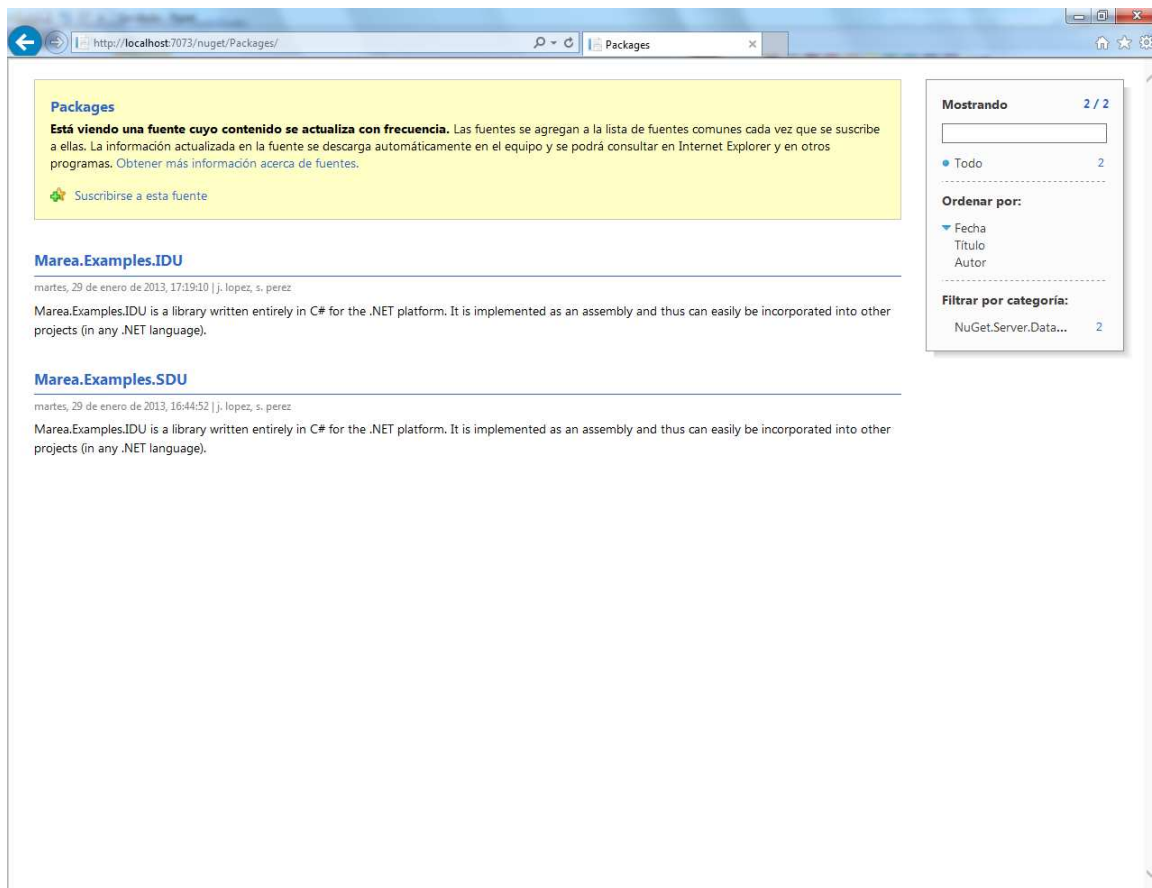
Figure A.9: View of OData over ATOM feed of MAREA packages

A.1.6.4 Thorn

Thorn is a command line utility accelerator for .NET applications. Thorn essentially provides a lightweight routing and dispatch layer to code. Such an environment encourages a style of development which relies on application-aware utilities, and also serves as a good springboard for experimentation, and ensures that one-offs that work out are already built in a repeatable, deployable, reusable fashion.

A.1.6.5 StringTemplate

StringTemplate is a java template engine (with ports for C#, Python) for generating source code, web pages, emails, or any other formatted text output. StringTemplate is particularly good at code generators, multiple site skins, and internationalization / localization. StringTemplate also powers ANTLR.

A template engine is simply a code generator that emits text using templates, which are really just "documents with holes" in them where you can stick values called attributes. An attribute is either a program object such as a string or VarSymbol object, a template instance, or sequence of attributes including other sequences. Template engines are domain-specific languages for generating structured text. StringTemplate breaks up

your template into chunks of text and attribute expressions, which are by default enclosed in angle brackets <attribute-expression> (but you can use whatever single character start and stop delimiters you want). StringTemplate ignores everything outside of attribute expressions, treating it as just text to spit out.

A.2 Source control tools

Source control is defined as the management of changes to documents, computer programs, large web sites, and other collections of information.

A.2.1 Git

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

The major difference between Git and any other version control system (like Subversion) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems think of the information they keep as a set of files and the changes made to each file over time.

Git does not think of or store its data this way. Instead, Git thinks of its data more like a set of snapshots of a mini filesystem. Every time the user commits, or saves the state of a project in Git, it basically takes a picture of what all the files look like at that moment and stores a reference to that snapshot.

Git has three main states that your files can reside in: committed, modified, and staged. Committed means that the data is safely stored in users local database. Modified means that the programmer has changed the file but have not committed it his database yet. Staged means user has marked a modified file in its current version to go into his next commit snapshot.

The basic Git workflow goes something like this:

- The user modifies files in his working directory.

- The user stages the files, adding snapshots of them to his staging area.

- The user does a commit, which takes the files as they are in the staging area and stores that snapshot permanently to his Git directory.

A.2.2 GitHub

GitHub is a web-based hosting service for software development projects that use the Git revision control system. GitHub is one of largest open source community which offers both

paid plans for private repositories, and free accounts for open source projects.

Git makes the code accessible and transparent for developers. Anything push by a user to a repository is instantly viewable online so users can share it with people even if they don't use Git. The main page of every project is a list of the files in the project and information about the last time it was committed to so the users can instantly see the code which is the most important about his project.

The site provides social networking functionality such as feeds, followers and the network graph to display how developers work on their versions of a repository. It also provides support to manage and contribute to projects from different devices.

A.2.3 Git Source Control Provider Extension

Git Source Control Provider is a Visual Studio extension that integrates Git with Visual Studio. This extensions becomes integrated with Visual from the update 2 of the Visual Studio 2012.

A.3 Cross platform tools

Cross-platform tools are a class of developer tool that aim to enable a single implementation of application functionality to run across multiple platforms.

A.3.1 Mono

Mono is a software platform designed to allow developers to easily create cross platform applications. It is an open source implementation of Microsoft's .Net Framework based on the ECMA standards for C# and the Common Language Runtime.

Some of the benefits of this platform are: popularity, higher-Level Programming, cross Platform, Common Language Runtime (CLR) support.

A.3.2 AlterNative

AlterNative is a tool for easy port applications from high-level languages (such as .NET) to native languages (such as C++). Most of the actual systems are C++ compatible, thus if the application is ported to this language, it can be executed in several platforms (i.e. smartphones, tablets, embedded systems, computers with different operating systems).

The process of the AlterNative software is divided in three parts: Decompilation, translation and compilation. The effort of this software is not focused on the code translation, but it is focused in maintaining all the features and functionality of the original code.

AlterNative it is a good solution for scenarios with several devices with some computational power able to execute applications compiled for different operating systems, or processors. AlterNative improves the global performance compiling the applications for every device with a native performance and without the need of a Virtual Machine or the support the same operating system or processor.

A.3.3 IOSharp

IOSharp allows standard Micro Framework applications to run on the top of a Linux operating system. Using this port the programmer will be able to deploy MicroFramework applications in Linux-based devices without doing major changes in the code. The only modifications that the programmer needs to apply are the references related to the underlying hardware. In addition, this port uses the GPIO, SPI and I2C from Userspace to be both distribution and architecture independent.

A.4 Continuous Integration tools

Continuous integration is a software engineering practice in which isolated changes are immediately tested and reported on when they are added to a larger code base.

A.4.1 Jenkins

Jenkins is application that monitors executions of repeated jobs, such as building a software project or jobs run by cron. Among those things, current Jenkins focuses on the following two jobs:

- **Building/testing software projects continuously**, just like CruiseControl or DamageControl. In a nutshell, Jenkins provides an easy-to-use so-called continuous integration system, making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build. The automated, continuous build increases the productivity.

- **Monitoring executions of externally-run jobs**, such as cron jobs and procmail jobs, even those that are run on a remote machine. For example, with cron, all you receive is regular e-mails that capture the output, and it is up to you to look at them diligently and notice when it broke. Jenkins keeps those outputs and makes it easy for you to notice when something is wrong.

Jenkins has been configured to build MAREA when a change is pushed to GitHub, and to run automatically the implemented tests.

Figure A.10: Jenkins project configuration

Figure A.11: Jenkins project general view

A.5 Material

The equipment used to perform the different tests is a PC DELL OPTILEX 755 with following characteristics:

- **Processor:** Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz.

- **RAM:** 4096MB (2x2GB) 667MHz DDR2.

- **OS:** Windows 7 Enterprise X64.

The .NET Stopwatch diagnostic class and the Visual Studio Profiler have been used to accurately measure the elapsed time, and to collect memory allocation data respectively.
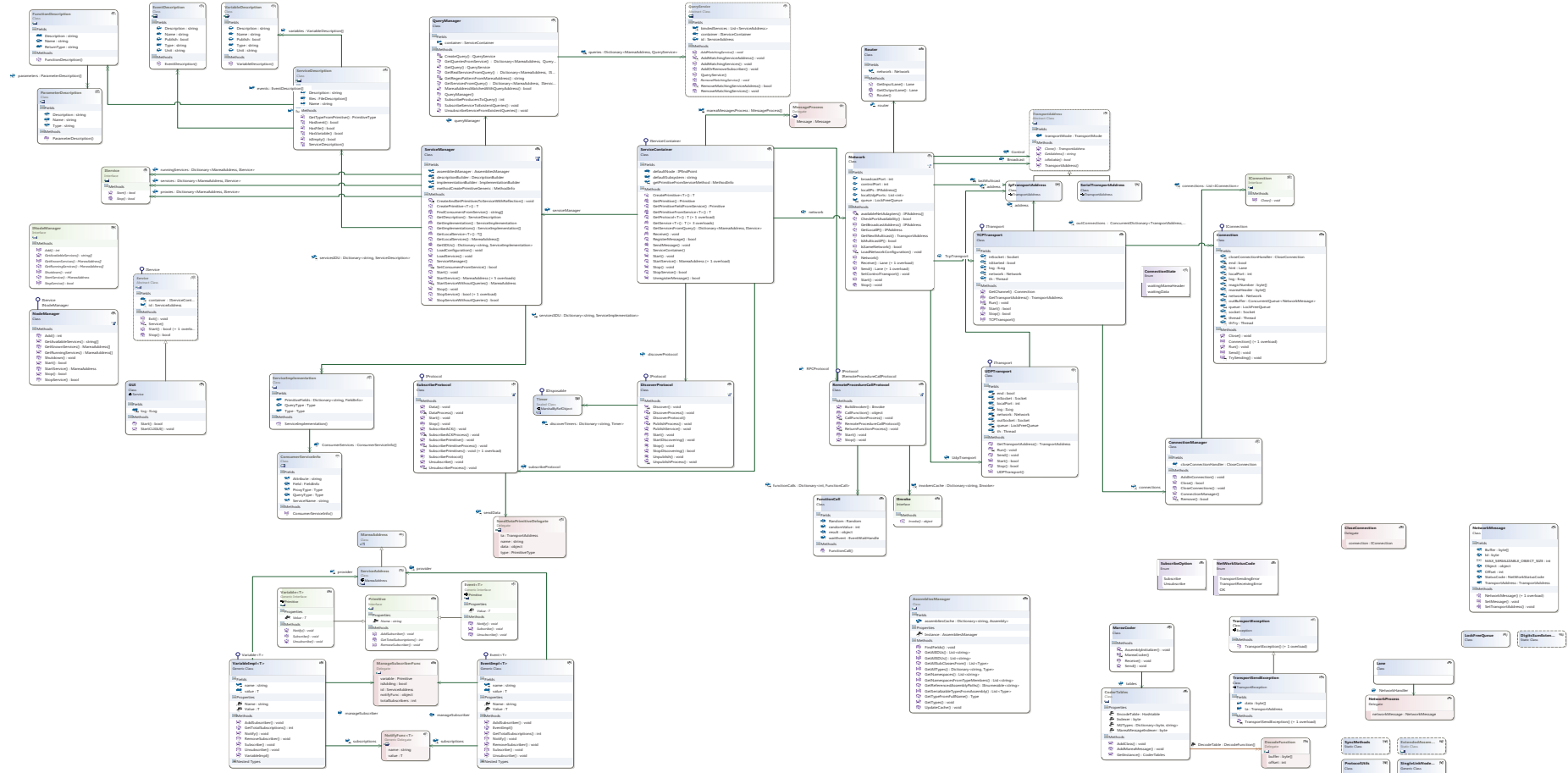
APPENDIX B. CLASS DIAGRAM

Figure B.1: MAREA core class diagram