NTNU
Norwegian University of Science and Technology

State-of-the-art Study and Design of a Small Footprint Version of the COOS Plugin Framework

Kashif Nizam Khan

Master in Security and Mobile Computing

Submission date:	June 2010
Supervisor:	Peter Herrmann, ITEM
Co-supervisor:	Samset Haldor, Telenor

Norwegian University of Science and Technology
Department of Telematics

Problem Description

Telenor Objects is a new business unit in the Telenor Group. Telenor Objects delivers managed services for connected objects, with the aim to increase the number of devices connected to the network infrastructure so that customers can benefit from real time information on their assets. The Connected Objects Operating System (COOS) is a platform built by Telenor Objects especially for developing Machine-to-Machine (M2M) communication and value-adding services. COOS is a modular and flexible platform, and includes a plugin framework offered to device and service developers for easy connecting services and devices to the platform.

The current version of the COOS plugin framework is based on Java Standard Edition and OSGI, with some support for development on J2ME. Telenor Objects wish to offer a small footprint version of the plugin framework for Window-based mobile/limited devices typically running Windows CE or Windows Mobile.
The assignment is to design and develop a prototype of a small footprint version of the COOS plugin framework. The work will consist of the following tasks:

Study the Connected Objects concept, and the COOS platform

Short state-of-the-art study on mobile/limited/embedded devices and the OS and programming support, main focus on Windows-based devices.

Define the requirements for the plugin framework

Design of a small footprint version Windows-based plugin framework prototype

Implement a prototype with the goal to connect to a COOS node and send and receive messages; either on simulator or on an actual device depending on time available.

Assignment given: 15. January 2010
Supervisor: Peter Herrmann, ITEM

Abstract

GSM and UMTS technologies have already gained a huge market penetration resulting in millions of customers. Machine-to-Machine (M2M) Communication is promising to be the next big technology that is going to hit the mass market with numerous essential services. Telemetry systems, which were thought once as the domain of big industrial companies, are now being available to larger and wider customers because of the advances in M2M communication. Thanks to mobile technologies, millions of small handheld devices are now available in the mass market which can be used to communicate real time information to the customers. Telenor Objects (a small business unit of Telenor Group) has defined a new Connected Object Operating system (COOS) which aims to provide a common platform for the devices to communicate real time data and to provide value added services to the customers. COOS is a modular and flexible platform, and includes a plugin framework offered to device and service developers for easy connecting services and devices to the platform. The current version of COOS plugin framework is based on Java Standard Edition and OSGI, with some support for development on J2ME. This thesis research work aims to provide a brief overview of the Connected Object concept and the COOS platform architecture. The main goal of this thesis is to design a small footprint version of the COOS plugin framework for Windows-based handheld devices. It will also provide a state-of- the art study on mobile device programming focusing on Windows-based services. This thesis research can serve as a starting document to provide a full functioning plugin framework for Windows-based devices and services.

Acknowledgements

It is a very special moment for me to submit my Master's thesis which eventually will end my Master's study. At this moment, I want to express my sheer gratitude to the Almighty and some special people without whom this thesis would not have been possible.

The list starts with my supervisors Professor Peter Herrmann of NTNU, Professor Tuomas Aura of AYY and my instructor at Telenor R & I, Mr. Haldor Samset. Their continuous guidance and support worked as a true inspiration for me throughout my thesis. Their able supervision has guided me through this thesis with great support. I would also like to mention the name of a very special person here: Mr. Knut Eilif Husa of Tellu AS, who is one of the designers and developers of COOS. Without his continuous help, guidance and valuable comments this thesis would not have been possible. Special thanks to Mr. Haldor for providing me an office place at Telenor R & I, Trondheim and the COOS team at Telenor R & I, Trondheim for being helpful and answering my questions.

Finally, I would like to thank some very special persons for whom I have been able to survive this solitary period: my greatest parents, my beloved wife Jinat Rehana, my family and all my dearest friends for their continuous love, affection and inspiration.

Trondheim, June 30th 2010

Kashif Nizam Khan

Abbreviations and Acronyms

3G	3rd Generation
AF	Actor Frame
AMR	Automatic Meter Reading
API	Application Programming Interface
CDC	Connected Device Configuration
CDMA	Code Division Multiplication Access
CLDC	Connected Limited Device Configuration
CO	Connected Object
COOS	Connected Objects Operating System
DICO	Deployed Infrastructure for Connected Objects
GPRS	General Packet Radio Service
GPS	Global Positioning System
GSM	Global Systems for Mobile
GUI	Graphical User Interfaces
IoT	Internet of Things
IP	Internet Protocol
J2ME	Java MicroEdition
JIT	Just-in-Time
JVM	Java Virtual Machine
LCM	Lifecycle Manager
LTE	Long Term Evolution
M2M	Machine-to-Machine Communications
OS	Operating System
OSGi	Open Service Gateway initiative
QoS	Quality of Service
PDA	Personal Digital Assistant
PL	Programming Language
R & I	Research and Innovation
RFID	Radio Frequency Identification
SDK	Software Development Kit

TCP	Transmission Control Protocol
UIQ	User Interface Quartz
UMTS	Universal Mobile Telecommunication System
URI	Uniform Resource Identifier
UUID	Universal Unique Identifier
WiMAX	Worldwide Interoperability for Microwave Access

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Human thirst for innovation started from the invention of wheels. Since then, we are always in quest of innovations that will simplify our day to day works. Advances in technology are now driving people to imagine the unthinkable and to see the unforeseen. The need to stay connected and share information led us to the invention of the internet. Since then, the internet has been the most effective and essential technology for the mankind. Today we cannot imagine our daily routines without the 'Google' or the 'Facebook', let alone millions of transactions and information sharing throughout the network globe. Human interactions and communication through the internet has outreached all other services delivered by technological boons.

However the predictable way of communication is changing. Imagination has taken us one further step ahead where machines are communicating and interacting with each other and sharing useful information. People are now talking about the 'internet of things' or simply network of objects [19, 27]. Household appliances, small handheld devices or pacemakers equipped with sensors are communicating over the internet, sharing useful information and in some cases taking crucial and complex decisions without human interventions. We can sense that Machine-to-Machine (M2M) communication is the new big leap in this era of technological advancements [17].

These networks comprising of such objects can generally deal with huge amount of data and can make quick decisions and swift responses if needed.

Simple surveillance and monitoring tasks normally takes up a good amount of time and a lot of data processing [21]. The idea in M2M communication is to let simple sensor devices perform these simple surveillance and remote sensing tasks and to aggregate the data into a data repository without human intervention. Sensor networks has been one of the most popular research areas in the computing and engineering society and it is expected to be the next big thing for the futuristic society.

Although, M2M communication has gained significant popularity in research arena, it has not succeeded to get the market penetration that was expected [35, 30]. There are several reasons for that. Currently, M2M communication is lacking the proper technologies and platform to share the data which is acquired from the environment. One reason for that is the confusion of the investors to invest in this sector. It seems that the potential outcome and the targeted customers are not well defined for M2M communications. Although, there have been some initiatives to provide a well defined platform and services for M2M communication, these initiatives lack proper cooperation and collaboration between the initiatives. As a result, the resulting technological environment is confusing and almost no standardization of such technology exists. The lack of standardization is hindering the deployment of reliable services for M2M communication [30, 38].

Different objects can communicate over different networks using diversified communication technologies. Moreover, the number of potential subscribers and the number of sensing devices is expected to be massive in numbers. As a result, it is very essential to provide a standardized platform for M2M communication. Telenor Research and Innovation (R & I) has taken an appreciable initiative to provide a standardized service platform architecture for what they call the 'Connected Objects'[26]. This initiative is aimed to provide a platform for M2M services and communications over any type of networks, employing any type of routing mechanism and using any type of smart devices.

Telenor R & I has defined this platform as 'Connected Object Operating System' or in short COOS. A small business unit named Telenor Objects is dealing with the development of COOS architecture [9]. Developing support applications for M2M has to deal with certain challenges like the vast differences in the capabilities of the underlying devices (or objects) and providing critical system properties like scalability and flexibility. The COOS service platform architecture offers some very useful and crucial properties like adaptability, scalability, device independence and flexibility. Large portion of the COOS platform (COOS Basic and Plugin Framework) is released as open source [3].

The COOS platform provides a plugin framework for objects to get connected to COOS. An object becomes a connected object when it is connected with the COOS through the plugin framework [15]. In simple words, the plugin framework is a gateway or entrance into COOS for external entities or objects. The current version of the COOS plugin framework is based on Java Standard Edition and OSGI, with some support for development on J2ME. Telenor Objects wish to offer a small footprint version of the plugin framework for Window-based mobile/limited devices typically running Windows CE or Windows Mobile. This thesis work is intended to design and develop a prototype of such a small footprint version of the plugin framework. The work will consist of the following tasks:

- Study the Connected Objects concept, and the COOS platform
- Short state-of-the-art study on mobile/limited/embedded devices and the OS and programming support, main focus on Windows-based devices.
- Define the requirements for the plugin framework
- Design of a small footprint version Windows-based plugin framework prototype
- Implement a prototype with the goal to connect to a COOS node and send and receive messages; either on simulator or on an actual device depending on time available.

1.2 Methodology

This thesis work intends to perform a state-of-the-art study on M2M communication, mobile/embedded device programming and to illuminate the working functionalism of the COOS platform. The final goal of this research work is to produce a small light weight version of the COOS plugin framework in .NET platform. The basic idea here is to get a grasp on the theory of M2M communication and produce a working documentation for further research on COOS plugin framework.

Although COOS platform is very subtle and flexible in its functionalism and possesses a very effective design methodology, the working version is not documented yet. We have tried to envision the whole concept step by step or we should say class by class from the code. Obviously the specifications presented in [15, 28] and the wiki pages ([39]) were the main source or design map during this journey of exploration.

Although M2M communication is a hot topic in the research arena, it is still an immature and less explored research field. This is one of the key reasons for starting this research by visualizing the current status of M2M and predicting its future. Although originally our main idea was to concentrate towards the COOS platform only, it soon became apparent that the credibility of a common platform like COOS can only be established if the platform is utilized by a significant number of machines or services. So we started our work by looking at the big picture at first. Chapter 2 presents the essential concepts related to M2M communications and some example scenarios offered by M2M services. It will also be interesting to investigate the market scenario for M2M communication as at the end of the day the success of a technology depends on its acceptability in the mass market.

COOS platform ensures the connectivity for the handheld embedded devices and smart phones and all other machines or objects able to send and receive data over the internet. As we target to produce a plugin framework in .NET platform, it is also necessary to find out the available platforms and programming languages for embedded/mobile device programming specially for windows enabled device. In this regard, we will also try to produce a state-of-the-art study on the current technologies regarding mobile device programming.

To understand the functionality of the COOS plugin framework, it is very important to grasp the basic definitions, components and principles of the working mechanism in COOS. Being a versatile project, COOS specifications covers almost all the ins and outs of the working mechanism. In doing so, these specifications do not highlight more on a specific part like the plugin framework. So, we will try to fetch out the important basics that are compulsory to visualize the working principles of the plugin framework.

As we are trying to produce a small prototype of a plugin framework that already exists, it is very important to derive the new design following the same workflow from the existing design. Our focus will be to induce the new model from the existing one. At first we will try to focus on a design that contains minimal functionalities that must be present in a plugin framework. As this work aims to produce a small working prototype, our goal will be to design the framework in a flexible way so that future research works can add up new functionalities easily. As we also aim to produce a working version of the prototype, we have chosen Visual C# as the programming language. The final outcome of this thesis intends to provide a precise documentation for designing a small footprint version of the COOS plugin framework and a working simulator which is able to connect and communicate with the COOS platform.

1.3 Scope

From the research findings related to M2M communication, it is apparent that this field has enormous possibilities in which many potential aspects are not dealt yet. Although this thesis includes a brief overview of M2M communication scenario, it is not directed towards producing a survey on this topic. The main focus will revolve around finding out the possibilities and problems related to the field of information and communication technology.

COOS platform comes to rectify the need of a common platform for enormous diversified devices and objects. It also presents an opportunity for developing and integrating new potential services. It offers several Application Programming Interfaces (APIs) which can be used as building blocks for new services. It means COOS has many different and diversified aspects. For this obvious reason and for the vast perimeter of the COOS system, it is not possible to discuss all the aspects of COOS in a Master's thesis. Our main focus will be towards the communication and transport mechanisms of COOS. We will revolve around the functionalities which are related to the plugin framework. Advanced COOS features like module specifications or model driven service development are out of the scope of this thesis.

1.4 Thesis Outline

This thesis is split into 8 chapters. In Chapter 2 there will be an overview of the M2M communication and the *Internet of Things*. As M2M communication is a vast topic which covers a broad area of the information and communication technology, it is not possible to cover all the aspects of this diversified technology in a singal chapter. Chapter 2 will therefore focus only towards the important definitions, available technologies to support the evolution of M2M communication, some example architectures and services and most importantly the market scenario revealing the current status of M2M communication in the global market.

Chapter 3 discusses about the available OSs and programming language support for developing applications for the mobile/embedded devices. One of basic objective of this thesis is to implement a simulator of a small working version of the plugin framework and if time constraints allow we will also test the simulator on an actual device. The purpose of this chapter is to gather the necessary information available for embedded/mobile device programming.

In Chapter 4 we illustrate the working principle and architecture of COOS. This chapter contains the essential information regarding the Connected Object platform and it will help the reader to understand the basics of the COOS and more importantly the plugin framework. For obvious reasons, the discussions of this chapter are directed more towards understanding the plugin framework in COOS and its transport mechanisms. It also contains some important definitions and terms that will be used frequently in the upcoming chapters.

Chapter 5 presents the detailed architecture and working mechanism of the plugin framework. It also includes the features of the plugin framework and the design of the framework in JAVA. The information presented in this chapter represents the way we have visualized the plugin framework looking into the code base of COOS.

Chapter 6 documents the basic requirements that a plugin framework design must address. It contains all the detailed information for constructing a plugin framework and we have used this chapter as the base for our design which is presented in Chapter 7.

In Chapter 7 we present our main contribution which is the design methodology and the implementation mechanism of a small footprint version of COOS. By 'foorprint' we mean that this version of the plugin framework is a small lightweight prototype of the COOS plugin framework and it includes only the basic functionalities.

Lastly, Chapter 8 ends the thesis with a short discussion on the key findings of this thesis and a brief discussion on the possibilities to extend this work in future research.

Chapter 2

Machine-to-Machine Communication

The internet has become an unimaginable store of data where every second hundreds of terabytes of data is flowing around the globe. It has become a challenge to process all the real time data, identify information from the data, signify the importance of data and take necessary decisions based on the information. As a result, the focus is rapidly changing towards a communication paradigm where machines or objects are able to sense data, communicate the data between them and in some cases process the information from the data. We are slowly shifting towards a technological society where virtually everything will be connected to every other thing. We are talking about the Internet of Things [19, 27]. This chapter presents an introductory discussion on M2M communication. It also includes a brief discussion on the status of the M2M communication and the Internet of Things.

2.1 What is M2M Communication?

We have seen how General Packet Radio Service (GPRS) , Global Systems for Mobile (GSM) and now 3rd Generation (3G) mobile technologies along with the internet, has affected the global communication. These efficient technologies are offering connectivity to the mass people on very cheap costs. M2M communication is promising to be the next big technology if it is introduced efficiently on a global scale. Although, this thesis is explicitly using M2M as Machine-to-Machine communication, the word 'M2M' may be used as Machine-to-Machine, Machine-to-Man or Man-to-Machine communications in various contexts.

In a general sense, Machine-to-Machine communication is defined as the automated exchange of information between two machines without any human intervention at either end of the system [17]. M2M communication precisely defines a network of intelligent assets which are capable of sensing, communicating and processing information. The assets can be sensors, mobiles, printers, faxes, wireless displays, cars and truck fleets, fitness monitors and what not. Such assets can monitor different physical conditions like temperature, speed, light, heart rate and movement. By sensing and sharing such type of information, machines can react and make decisions on a particular situation without human intervention [22].

M2M communication prefers wireless technologies to maintain connectivity. The choice of technology varies from infrared, bluetooth, zigbee to GSM, Code Division Multiplication Access (CDMA), Universal Mobile Telecommunication System(UMTS), Long Term Evolution (LTE), WiFi and WiMAX, depending on the application of the M2M service. As a result, M2M can offer diversified services to the end users which are discussed in later sections.

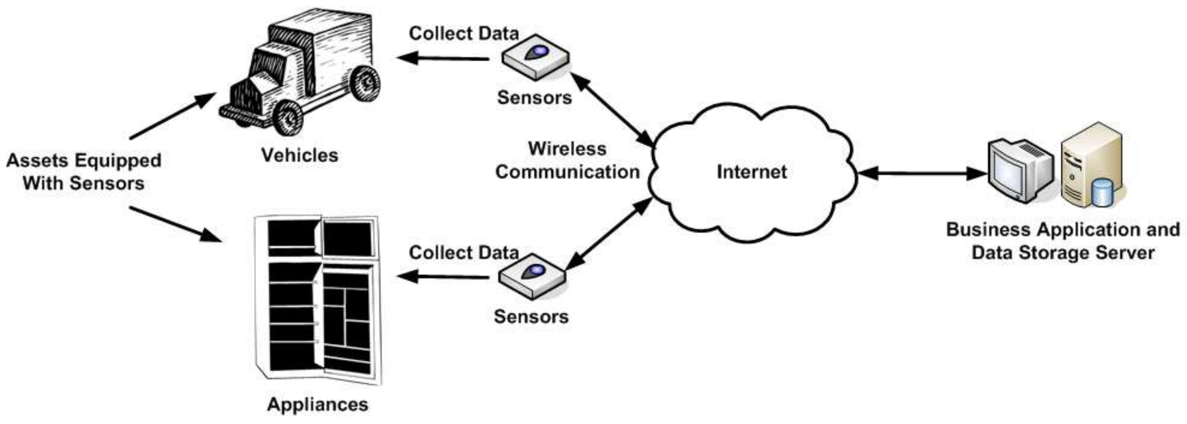


Figure 2.1: M2M Communication Scenario

Figure 2.1 presents a general M2M communication scenario. The assets (vehicles, appliances etc) are equipped with sensors. The sensors are continuously collecting data from the assets. The type of data sensed depends on the specific service of the M2M communication scenario. For example, in case of vehicle, the sensors may sense the battery condition of the vehicle, the current fuel level from the fuel tank or the status of the engine. This information is then transmitted over the internet to the business application server which receives the data, processes it and takes necessary action (for example alerting the driver about bad engine condition) based on the data received. It is not necessary that the data is transmitted over the internet every time. For example, a self diagnostic system for a vehicle may gather the data from the sensors via short-range wireless technologies (bluetooth, zigbee etc) and warn the driver about alarming conditions. This figure is just an illustration of how things happen in M2M communication.

2.2 Internet of Things

With the advent of wireless communication technologies, we are now able to communicate anywhere, anytime with anyone. Such ubiquity is taken us one step ahead, where we are now talking about the connectivity anywhere, anytime with *anything.* It simply means that the dimension of connectivity to anyone is now expanded to allow the communication between anything (Figure 2.2).

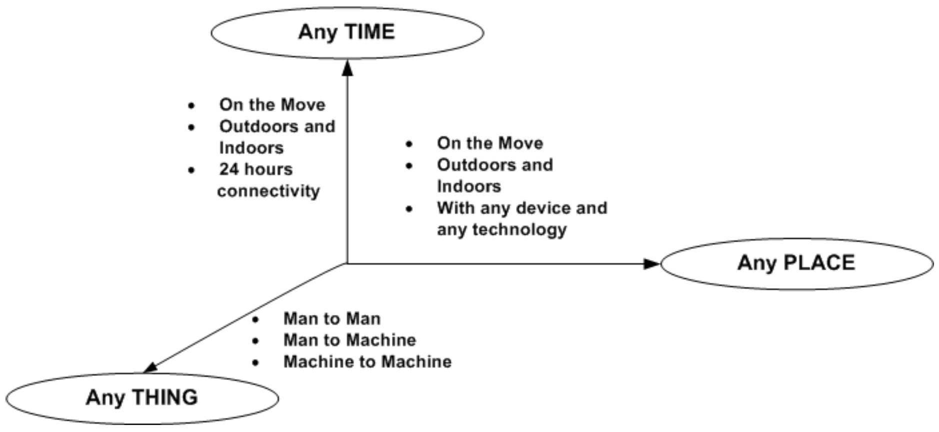


Figure 2.2: Internet of Things- A New Dimension

The idea of *Internet of Things* is composed of two individual concepts: *Internet* and *Things*. *Internet* can be defined as:
"The network of interconnected computer networks which is based on the standard communication protocol, TCP/IP", while *Things* are
"Physical objects that can be uniquely identified digitally by some means". So,

we can define *Internet of Things* semantically as
"The network of interconnected objects or things that are uniquely identifiable and can communicate using a standard communication protocol".
This definition of Internet of Things (IoT) is still very abstract. [40] defines IoT as
"Things having identities and virtual personalities operating in smart spaces having intelligent interfaces to connect and communicate within social, environmental and user contexts".

The concept of IoT, specially the set of functionalities and actions that the connected objects should or can perform, is still undergoing an enormous research. In a more general sense, if an object is equipped with the knowledge of its own properties like creation, transformation, actions, use, recycling or some other useful properties, the object can interact the with the physical world [40]. The current internet can be thought of as a collection of homogeneous devices (PCs, mobiles, Personal Digital Assistants (PDAs) etc) as they more or less exhibit the same communication properties and purposes. However, it will be a great challenge for IoT to provide seamless connectivity among diversified heterogeneous devices. These things or objects can be meant for completely different environment and can have completely different properties.

IoT can bring a technological revolution if it is supported by dynamic and prominent technologies. The most important concern here is to identify each device uniquely. IPv6 shows a good example of unique addressability in the internet where all the connected devices have a uniquely identifiable address. IoT can use a similar structure to create an address gamut where all the physical objects and things will be uniquely identifiable [38]. In such a way, all the objects can identify each other, interact with each other, exchange or relay information and actively participate in the processing of information.

Radio Frequency Identification (RFID) is a very simple and cost-effective way of item identification. RFID systems can be seen as a next-generation technology for bar-codes. However, it offers much more dynamic functionalities other than item identification. For example, RFID systems can track items in real-time with the help of Global Positioning System (GPS), which provides essential information about the objects like physical location or status. Sensor technologies are also emerging in this regard to collect and transmit the physical status or data from the objects or things.

The idea of IoT is very simple yet very innovative. IoT paves the way for new pioneering services that can exploit the connectivity and accessibility of everything from anywhere. It can greatly simplify our social life, increase the quality and efficiency of communication and broaden our network of interaction. Still it is unimaginable to foresee the types of applications IoT will offer. IoT is envisioning about almost endless possibilities.

2.3 Sensor Networks

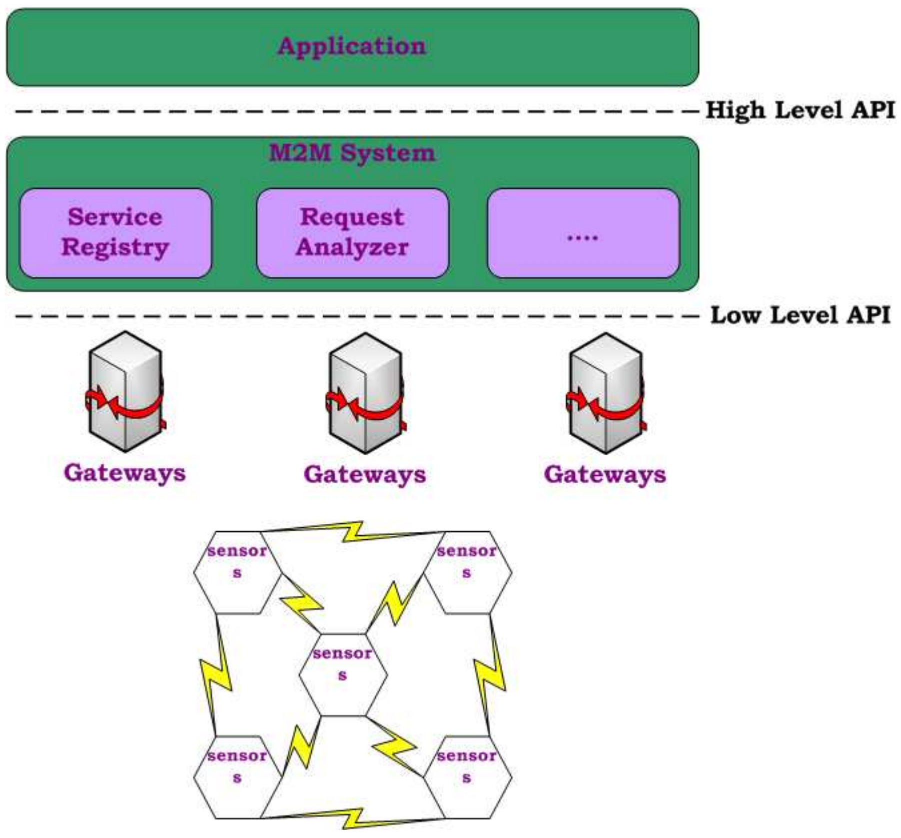


Figure 2.3: Sensor Networks - An example Architecture

In most scenarios, M2M vision consists of several sensor networks which are collecting data from the environment and communicating the data to the application servers through the internet. The basic idea is that, objects or things are outfitted with sensors. The sensors sense or monitor the data of interest and communicate it to remote users or applications of interest. Figure 2.3 presents a typical architecture of sensor networks presented in [31].

In the lowest level of this simple scenario, the sensors are deployed in the environment to monitor the behavior of objects or the environment and collect the data. Data from sensors are aggregated at the wireless sensor network gateways. Gateways generally address the queries from the M2M system, select the set of sensors that match the appropriate query and process the data collected from the sensors according to the query. Gateways also interact with the M2M system with the help of an API. M2M system acts as a middleman between the applications and the wireless sensor networks. It performs several essential operations such as: service registration, request analysis, service publication functions etc. Based on the request of the application, it generates an appropriate query, selects the proper gateways and communicates information between the application and the sensors. On the highest level of this architecture lie the users or the applications that use the wireless sensor networks to gather information of interest.

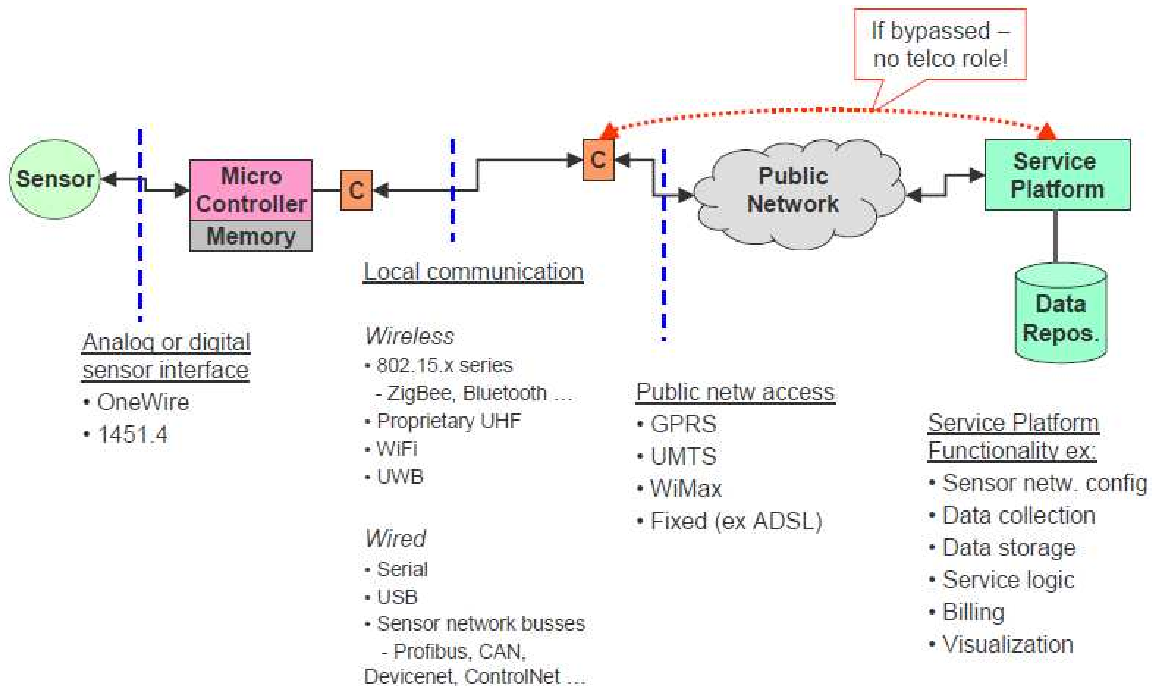


Figure 2.4: Detailed Architecture of a General Sensor Network[24]

Figure2.4 presents a detailed architecture of a generic sensor network involving all the major communicating parties namely: Sensors, Local Communication Network, Access Network, Transport Network and Service platform. Figure2.4 also presents a nice overview of the different technologies that can be used by these parties. The local communication can be wired or wireless. However, wireless communication is more convenient in this regard and hence preferred over the wired communications. Bluetooth, Zigbee or WiFi are example technologies preferred for wireless communications in local area. With the current advancement in telecommunication technologies, GSM, UMTS or WiMAX will be more suitable for the interface between the local area network and the service platform. This figure also points out that, in certain installations public networks can be totally bypassed if there is a closed private network or remote management available. In such case, there is no need to use the public telecommunication networks. However, this is a special case and requires expensive installations.

2.4 M2M Services and Applications

This section presents some examples of the services M2M can provide us. Let's start with an example of a retailer service. In this scenario, the retailer supplies the goods to the retails store and the biggest challenge he faces is to track the items while they are supplied to the stores [17]. The M2M communication offers a very simple way of monitoring the products as they are transported along the distribution channel from the retailer's storage area to the retail stores.

- The items are attached with a RFID tag in the store house. This RFID tag is able to sense some important information from the products such as the temperature.
- While being loaded into the lorry, the items pass through a RFID reader panel so that the exact information of all the items loaded, can be stored.
- The movement of the lorry is tracked by the GPS system that continuously reports its location to a central server. The mobile technologies like GSM, or UMTS can be used to send the continuous location information.
- The store manager can track the progress of the product delivery continuously. In such a way, the store manager can also monitor the prod-

ucts' status and assess any stock level problems or shortage of stocks in early stages.

- Upon arriving at the retail stores, the goods are again unloaded by passing through a RFID reader panel. Now, the store keeper can monitor the delivery and check the status of goods. For example, he can check the temperature of the goods to examine whether the goods were transported exceeding the temperature limit or not.

This is a simple example that shows how M2M communications can improve the quality of the process management. [24] presents some possibilities for M2M services. We will discuss one potential M2M services in the following section.

2.4.1 Personal Health Management

The personal health management uses sensing devices to monitor and collect data about the personal health and stores it in a database. It offers services regarding the access of the healthcare data to the interested stakeholders. The basic idea is to deploy sensors in patients' environment and monitor the healthcare related information from the environment or directly from the patient. The data can be communicated to an application server via the public transport network preferably using a telecommunication network. The application server stores the data and offers the access of the information to interested business logics. The information provided here may be regarding the personal health of a specific patient or the overall statistical data. This overall scenario is depicted in Figure 2.5.

In the context of M2M service, this personal health management system can be very effective. For example, the healthcare information can be monitored remotely by the medical application servers and inform the responsible medical personnel about the current situation of a patient. It can be more effective in disaster scenarios like earthquakes. For example, it can help the rescue workers to locate the patient. It can also help the medical personnel about the status of the patient and they can arrange medical assistance beforehand. All these possibilities makes the health care management system much more efficient and effective and improves the quality of service.

This is a very simple example which illustrates the never ending potential of the M2M services. Service like environmental monitoring, animals and

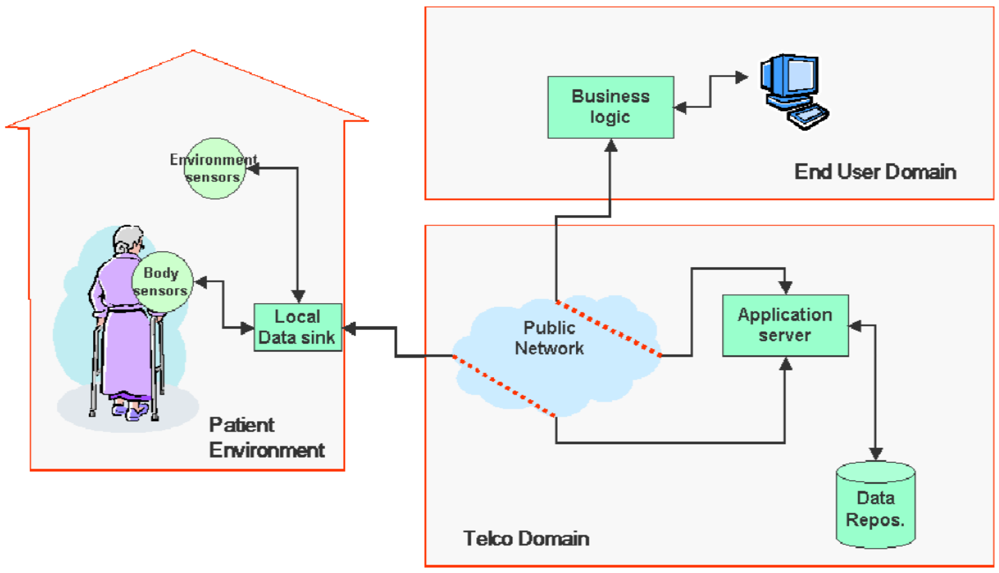


Figure 2.5: Personal Health Management using Sensor Network[24]

fish farm monitoring, fire warning system or remote home monitoring system have already started to create new business opportunities. Such services can create a very positive impact on our day to day lives. Companies like Telefonica or Siemens have already started to offer home automation and remote home management systems and devices. Figure 2.6 presents some of the possible M2M services. M2M communications have a huge potential for diversified business ventures. Much of the M2M research effort is now directed towards exploring the mobile M2M communication possibilities specially the mobile M2M services. [41, 37, 18, 32, 20, 29, 36, 23, 33] are examples of such research which has produced appreciable results. So, more innovative and efficient M2M services are going to be invented in near future.

2.5 M2M Market Scenario

A lot of research has been devoted towards predicting the future marker scenario for M2M communications [35, 30, 17, 24, 21, 16]. Many of these reports picturize the global M2M market scenario. In this section, we will limit our discussion to European markets only.

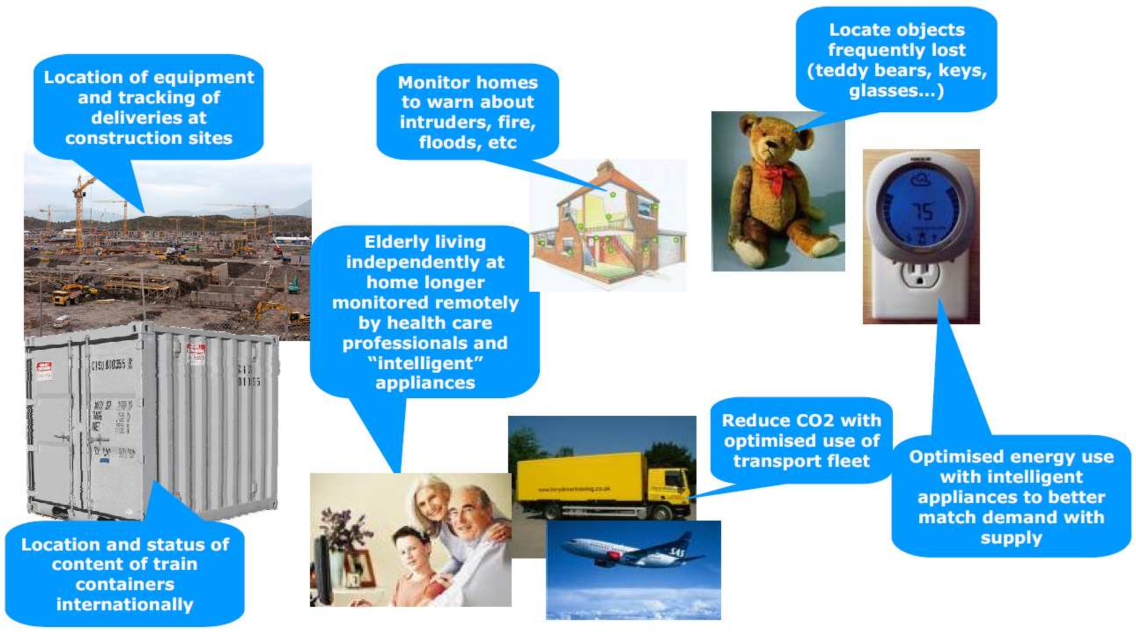

Location of equipment and tracking of deliveries at construction sites
Monitor homes to warn about intruders, fire, floods, etc
Locate objects frequently lost (teddy bears, keys, glasses...)
Elderly living independently at home longer monitored remotely by health care professionals and "intelligent" appliances
Reduce CO2 with optimised use of transport fleet
Optimised energy use with intelligent appliances to better match demand with supply
Location and status of content of train containers internationally

Figure 2.6: Example of Possible M2M Services[24]

Although, M2M communications in form of telemetry and remote sensing are present for quite some years now, the market scenario has not been so impressive. [17] points out some key issues that hinder the seamless growth of M2M market so far. These are:

- Lack of low cost technology for local access media.
- Lack of a single comprehensive application (i.e Google, Facebook).
- Higher costs for integrating M2M solution with existing systems.
- Inability of the vendors or policy makers to describe a proper value chain.

However, with the recent advent in mobile technologies, the mobile M2M market has started to gain some excellent figures in case of market penetration. With the lessons learnt from previous mistakes, policies for M2M communications now focus more on stratergical advantages. The European market has approached the mobile M2M market in a sensible way. One of the key reasons for this is the common mobile telecommunication technology used throughout the Europe which is GSM. Moreover, each European market

Table 2.1: Deployed Wireless M2M Devices in Europe

Segment	2006	2007	2008	2009	2010	2011
Private vehicles	0.6	0.9	1.6	2.3	6.1	15.7
Commercial vehicles	0.4	0.6	0.8	1.2	1.9	2.5
Energy meters	0.8	0.9	1.2	1.2	1.4	1.7
Security alarms	0.5	0.6	0.7	0.9	1.1	1.5
POS-terminals	0.5	0.5	0.5	0.6	0.7	0.8
Other	1.3	1.5	1.7	1.8	2.0	2.2
Total (millions)	4.1	4.9	6.4	7.9	13.2	24.4

targets to monopolize single product solutions and also the value chains are clearly defined.

EU regulations require each home to have Automatic Meter Reading (AMR) system and each vehicle to be fitted with M2M devices. So, the scenario looks very promising for the mobile M2M markets. In a recent research performed in [30], it is found out that in 2008 alone one third of the electronics goods sold were internet enabled which is 27 percent more than the year 2007. In 2011, half of the devices produced will be internet enabled and it will be two third around the year 2014[30].

In 2007 T. Ryberg et. al performed a survey on the European wireless M2M market in [35]. The figures that came out during this research are very promising. According to this research the number of deployed M2M units or devices reached 4.1 millions in the year 2006. It is predicted here that, this number is likely to raise upto more than 24.4 millions around the year 2011. Table 2.1 depicts the number of deployed M2M enabled devices in Europe according to the research performed in [35].

These numbers are highly promising. The mobile network operators have also started to pay attention towards manipulating this massive market. As discussed in the previous section, much research efforts have been directed towards creating useful and consumer friendly M2M applications and services. In this process, Telenor has focused to develop a common platform for the M2M devices to enable them to connect and share data more efficiently. COOS is a significant step towards defining a structured service architecture for M2M communications. M2M communications has created new mass market for the business ventures. It will be interesting to see how this big market is approached strategically.

Chapter 3

Mobile Device Programming

The introduction of wireless communication technologies, small handheld and mobile devices have gained enormous popularity. The ease of access and the need to stay connected anywhere at any time have also paved the way for new emerging mobile services and applications. As a result, a lot of research is currently directed towards improving the efficiency and effectiveness of the mobile technology as well as improving the quality of service. However, such enormous research effort is also backed by generous technological support. This has been possible because of efficient Programming Languages (PLs) and Operating Systems (OSs) which have provided a flexible environment to the application developers. In this chapter, we will mainly try to find out the PL and OS support for embedded/mobile device programming.

3.1 Platforms for Mobile Device Programming

The need for a good and stable platform for mobile device programming was felt from the introduction of small handheld devices specially mobile phones. Small processing power, limited battery life and relatively small graphical displays put a lot of constraint for the application developers. The conventional programming languages which are used to develop programs for normal computers and laptops are not suitable for small handheld devices like smart phones.

Now a days, almost all the popular programming languages provide the support for embedded/mobile device programming. In addition, a good number of new platforms have been developed solely for the mobile device application development.

3.1.1 Java MicroEdition

Java is one of the most popular and flexible programming languages. Java MicroEdition which is well known as J2ME in the developers' arena, provides the same essence of Java with a flexible, stable and reliable environment for mobile application programming [6]. It has been designed keeping the constraints of the embedded devices in mind. Java ME is not merely a programming language. It is a combination of the technology and its specifications. With the help of this platform, it is easy to produce a Java runtime environment fulfilling the requirement of limited memory and power. J2ME provides configurations with the essential libraries and virtual machine and APIs [6].

With the current advancement in mobile technologies, now there is also a large gap between the capabilities of various mobile phones. Some of the mobile devices now possess the capability of extra processing power and memory than the other limited ones. Keeping this difference in mind, J2ME provides two different types of configurations for mobile device programming: *Connected Limited Device Configuration (CLDC)* and the *Connected Device Configuration (CDC)* [6].

The basic features of J2ME includes cross-platform compatibility, better user-experience, security and safety, rich class libraries for easy application development and last but certainly not the least, is to allow the basics of Java technology so that a skilled Java developer can easily grasp its basics.

3.1.2 Symbian and Maemo

Symbian and Maemo are two very famous OSs for mobile device programming. These two OSs are mainly developed and maintained by Nokia with the help of some other open source communities such as Debian and GNOME [7, 8]. Although these two OSs were initially designed and developed keeping specially Nokia mobile devices in mind, both of them have gained much popularity resulting in some other manufacturers producing Symbian or Maemo based devices.

Symbian OS is a very organized and subtle software system. Symbian uses C++ as the native programming language for developing applications. It is a very smartly built software environment which allows the developers to produce flexible, efficient and high performance software products for mobile/handheld devices [34]. Initially mobile devices had very little graphical support with the inputs mainly controlled by keypads. Symbian OS provides a very efficient way of talking to the hardware. It provides both the low-level APIs as well as the high-level APIs for developing technological components for different types of devices. Till now, the most popular Symbian OS phones that have conquered the market are Nokia 80 series, Nokia 60 series and User Interface Quartz (UIQ) series phones [8]. Symbian provides developer tools and free emulators for the developer to design new software products. By maintaining an open source community, it also provides numerous useful and efficient APIs and source codes for mobile application development.

Maemo is a similar type of open source project like Nokia and it is also a great initiative taken by Nokia to provide a proficient platform for advanced smart phones like Nokia N810 internet tablet [7]. Like Symbian, Maemo also provides a Software Development Kit (SDK) for developing application on top of the platform. Maemo OS is essentially based and inspired by Linux OS. This OS not only provides some very useful components to the developers, it also allows the easy integration of components from various online repositories. Developers can choose the programming languages like C, Java, Python or Rubi for developing applications in Maemo. However, C and Java are the two most popular choices among the Maemo developers. In case of Java development in Maemo, a Java Virtual Machine (JVM) named Jalimo is available [5]. It has a very rich class library. With the support of Debian and GNOME, Maemo offers a scalable, reliable and stable platform for mobile application development.

3.1.3 Windows Mobile Programming

Windows based mobile devices have gained much popularity among the mobile users and as such windows enabled embedded devices have a good share of the products in the market. Microsoft provides a set of OSs for mobile/embedded device programming known as *Windows Embedded*. Among others, *Windows Embedded CE* is arguably the most popular OS available for windows enabled embedded devices. It is also sometimes called as *Windows Mobile* OS. It was previously known as *Pocket PC*.

Windows CE is a highly customizable real-time operating system specially designed for embedded devices or smart phones that have constrained power and memory. It provides tools and emulators that allow the developers to build custom environment for specific devices. One of the interesting features of Windows CE is that it resembles a small Windows 32-bit OS [12]. As such it supports most of the Win32 APIs and it also supports the x86 and ARM like processor architectures [13]. Microsoft has developed a number of popular mobile OSs based on the Windows CE kernel. Among those, Pocket PC, Windows Mobile and Smartphone are widely used. Microsoft also provides numerous useful tools and plugins for easy integration and development over the Windows CE. Visual Studio 2005 and 2008 support the development of Windows CE applications based on emulators. A .NET framework is available for windows based embedded device programming which is based on the Just-in-Time (JIT) compiler. The .NET framework is equipped with a large library and run-time management system.

3.1.3.1 Programming Languages for Windows Mobile Programming

There is also a good set of choices for developers when it comes to the selection of suitable programming language for windows enabled embedded/-mobile device programming. Visual C/C++ is often preferred as the programming language in this case because with thess languages, it is possible to talk directly with the hardware without any intermediary layers [2]. With Visual C++ it is possible to develop faster, light-weight and flexible mobile applications. Visual C++ can communicate with the Win32 APIs very efficiently and it is much easier to interact with the conventional Windows Desktop APIs using Visual C++. That is why Visual C++ is sometimes referred to as the *native language* for windows mobile application development. However, when it comes to debugging and error-handling, Visual C++ environment may be much more challenging than other Visual languages like Visual C# [2]. It is also a bit complex to design high level interfaces for mobile application using Visual C++.

Visual Basic .NET and Visual C# are also very popular programming languages for mobile application development specially for windows enabled devices. These two programming languages are very much preferred for the new programmers because these are easier to learn. A lot of underlying plumbing is performed by the .NET compact framework, so it is very easy for a developer to build application on top of the framework. It is also very flexible when it comes to designing nice high-level user interfaces. .NET framework greatly simplifies the task of the developer as there are many useful components already available for application development [2]. It is also much easier to convert a standard Visual C# or Visual Basic Code for windows mobiles compared to other programming languages. As these two languages do not talk to the hardwares directly but with the help of intermediary layers, they are also sometimes referred as *managed languages*.

Apart from these popular languages Java is also used for windows mobile development. J2ME is a popular language for windows mobile application development for traditional Java developers. For mobile web application development Jscript or ASP.NET is commonly used [2].

3.1.3.2 Other Platforms for Smartphone Programming

Recently a good number of research has been performed for developing simple, flexible and efficient OS for mobile device programming. Among them Android, iPhone and BlackBerry have gained much popularity. Android is an open source project which is led and maintained by *Google* [1]. Android enabled phones have gained significant popularity very recently and a recent research shows that Android enabled phones rank in the second position in US among the consumers for the first half of 2010 [1]. Android platform is mainly based on Java libraries and applications and linux kernel. Android offers the essence of traditional Object Oriented platform for the mobile application developers. Its popularity is based on the flexibility of the platform and the large and rich repositories of open source components and applications.

iPhone OS is the OS used for some of the most popular Apple devices namely *iPhone*, *iPad* and *iPod Touch*. Unlike other mobile device OSs, iPhone OS is a proprietary software system solely managed by Apple. It is based on Mac operating system. Apple provides a SDK for third party software developers and maintains an application store for the consumers. The SDK contains almost 1500 APIs for the developers and offers a large collection of library classes for Application development [4]. It is a bit complex to develop applications for iPhone OS as the traditional Mac based applications are not supported for iPhones.

Chapter 4

Connected Objects

As discussed in Chapter 2, M2M communication is offering a rapidly growing market. The number of devices eager to connect and share information is rapidly increasing day by day. One of the major issues for developing a general platform for M2M communication is the diversity in the capabilities and functionalities of vastly different devices. The second problem that comes across M2M communication is the absence of a general communication platform that supports properties like scalability or flexibility. It means that this hugely promising M2M communication lacks a general platform which offers a general solution for the different devices or object to stay connected and share information. COOS is a very innovative and efficient initiative of Telenor R & I which offers a scalable, reliable, flexible and secure platform for M2M communications. This chapter presents an overview of the COOS architecture and its basic functionalities.

4.1 Architecture of Connected Objects Service Platform

Figure 4.1 presents a general overview of the network architecture proposed by COOS. COOS offers a simple and flexible service platform for M2M communication. This CO(Connected Object) service platform is reachable through the internet as it is connected to the backbone IP network. The COs that wish to communicate with each other or with the CO service platform, are similarly connected to the internet [28]. The COs may also connect with each other using the transport networks or gateways. In this case, the COs

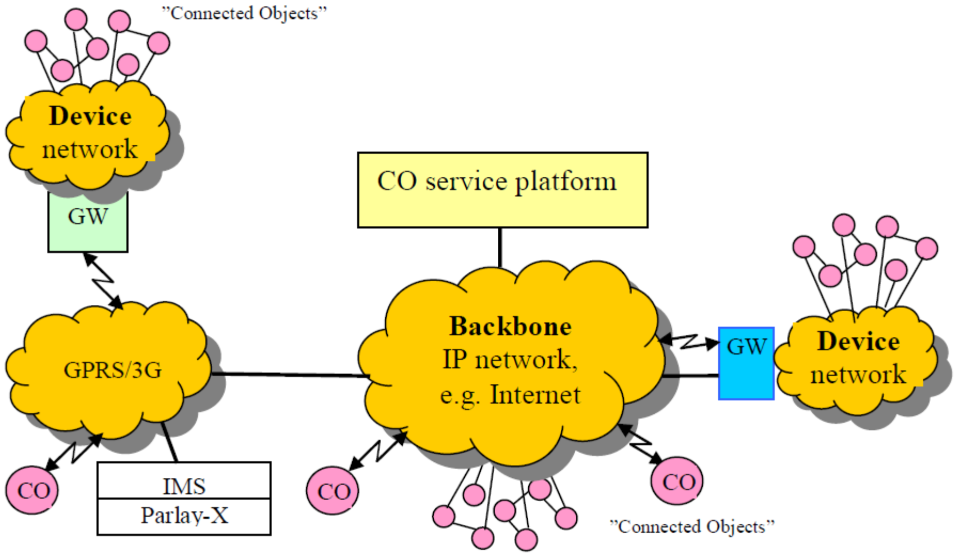


Figure 4.1: General Architecture [28]

may use the local communication network or 3rd party mobile telecommunication networks as the transport network. Figure 4.1 also points out the topology of the CO network. The COs may be connected directly one-by-one with the backbone network or they may be connected in groups with the backbone network directly or via gateways. So, the basic concept of COOS is to mediate the communication between different networks and objects and provide a simplified and general service platform.

The CO project aims to provide a generic plugin framework for M2M communication. The COOS project is an open source project and the developers can develop intended services according to the specifications provided with COOS. The up to date technical information is maintained in a wiki page [39]. The specifications are provided in [15, 28]. These documents are distributed through the Telenor's research web page and are accessible to all. We will start looking into some more details of the COOS architecture in the following sections, starting with some important definitions in the next section.

4.1.1 Important Definitions

This section contains some important definitions in the context of this thesis. For a detailed overview of all the definitions regarding COOS [26, 25, 39, 15, 28] are recommended readings.

4.1.1.1 Object

Object can be defined as an entity that is able to send and receive data. It can be a device or a service. Generally, an object can be a client, a server or both server and client depending on its required functionality [28, 26]. As a server, it can offer services to other objects and as a client it can request services from other objects. In this scenario, objects talk to each other through well defined interfaces. Depending on the actual communication scenario, an object can be a remote device, a sensor, a RFID tag or any kind of service.

4.1.1.2 Edge

An edge is a gateway between an object and the CO service platform. In other words, an edge is the connection between an object and the platform. The external gateways, objects or the local transport system uses the edge to reach and communicate with the platform [28, 26].

4.1.1.3 Plugin

Plugin is any object the can be attached into the messaging bus.

4.1.1.4 Module

Module refers to a set of functions in COOS. It is the smallest set of operations that perform a specific task. Messaging is an example of COOS module [26].

4.1.1.5 Application Programming Interface (API)

In the context of COOS, API refers to a set of rules that a software process must comply in order to communicate with another software process. In this framework, a software process can be an object, an operating system process or simply a protocol stack [28]. As far as information is communicated with a software process, the rules stated in the API must be fulfilled. In many contexts, API refers to the operational interfaces between different functional components in the COOS system.

4.1.1.6 Connected Object (CO)

An object becomes a Connected Object (CO) when it gets connected with the platform and communicates with the platform through an edge [28].

4.1.1.7 Customer Service

Customer Service is the service delivered by the CO service platform to specific customers. Customers need to register for a specific service with the COOS system in order to receive the offered services. It is also a specific role or model of object.

4.1.1.8 Gateway

In the context of coos, gateway is a device that connects the local communication network with the access network. Mobile devices or RFID readers are examples of gateways [28].

4.1.2 Detailed Architecture of COOS

The general Architecture of COOS is very straightforward. At the very lowest level, COOS has some modules. As defined earlier, modules offer specific functionalities. Modules can interact with each other through a messaging system. This messaging system is asynchronous [26]. The messages have a specific format that wraps the raw data to make the message carrier and content independent. Modules are deployed as OSGI bundles inside an OSGI container. Figure 4.2 depicts the general concept of modules.

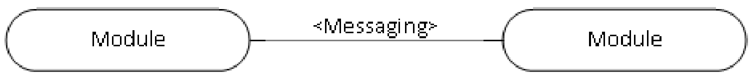


Figure 4.2: Modules in COOS

The OSGI container that contains the OSGI bundles of modules, is known as the *COOS*. COOS can interact with other COOS or objects from the real world. A connected object connect with COOS through the *Plugin Framework* (Figure 4.3). More detailed discussion about the Plugin Framework follows in Chapter 5.

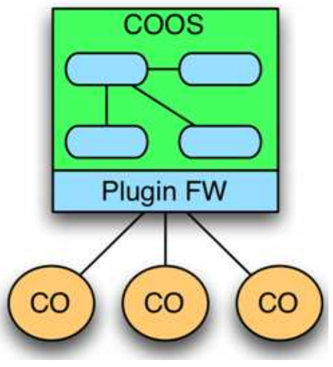


Figure 4.3: COOS [39]

A COOS instance generally runs on a physical machine [26]. When different COOS instances get connected and send and receive data among each other, a CO platform is created. And when such a platform is deployed, it is called Deployed Infrastructure for Connected Objects (DICO) (Figure 4.4). The topology or the structure of the DICO completely depends on the physical installations. Several DICOs can also interconnect with each other and share information. DICOs also offer the flexibility to share its components or modules among each other.

The basic idea behind COOS is to overcome the difficulties of M2M communication like inaccessibility due to lack of standardization. It promises optimal performance with secure data flow between connected objects [26]. By providing a standard platform of distributed APIs, COOS aims to create a global communication system for connected objects. On one hand it offers connectivity by allowing developers to create and distribute modules for their specific tasks, and on the other hand it ensures flexibility by allowing input control of the messages and providing a standard communication protocol. COOS has its own object-naming scheme which uniquely assigns addresses to objects. For the transportation of messages, it uses the standard TCP/IP protocol stack. In this case, customers are allowed to choose the security

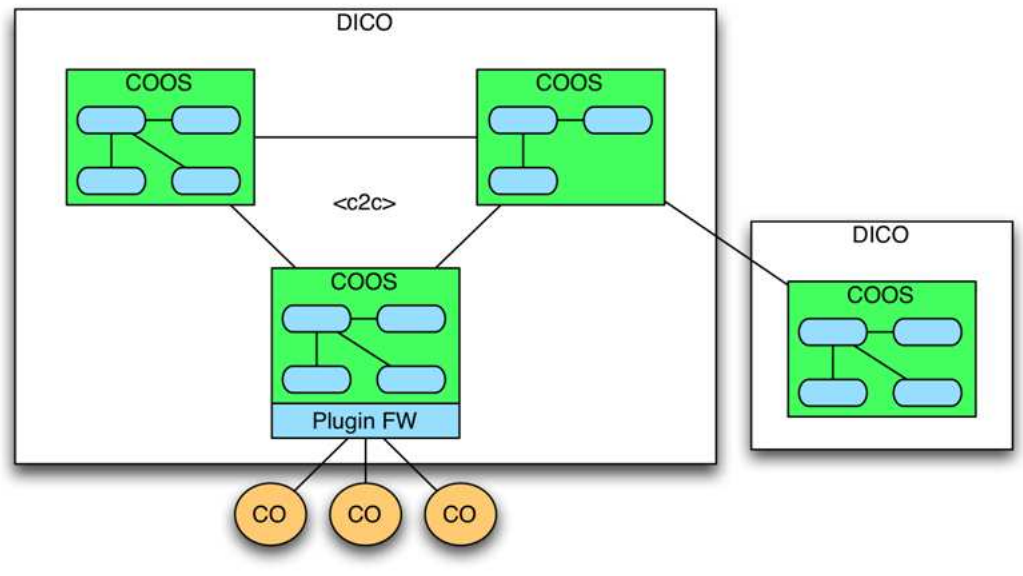


Figure 4.4: Deployed Infrastructure for Connected Objects (DICO) [39]

level for the transportation of messages. As such, it uses IPSec protocol for the authentication of the parties and end-to-end secured transportation of the messages.

As stated earlier, different components in CO service platform interact with each other through different interfaces. Figure 4.5 is a detailed illustration of Figure 4.1 which highlights the different interfaces between different components.

One of the vital points that should be kept in mind while standardizing any communication platform is to clearly define the set of rules specifying how different components should talk with each other. CO platform specifications ([15, 28]) clearly states the requirement and functionalities for the different interfaces shown in Figure 4.5. We will not discuss the details of the requirements here as this is out of the context of this thesis. For detailed reading [15, 28] is prescribed.

The architecture of COOS is based on four crucial design properties or principles. These properties are :

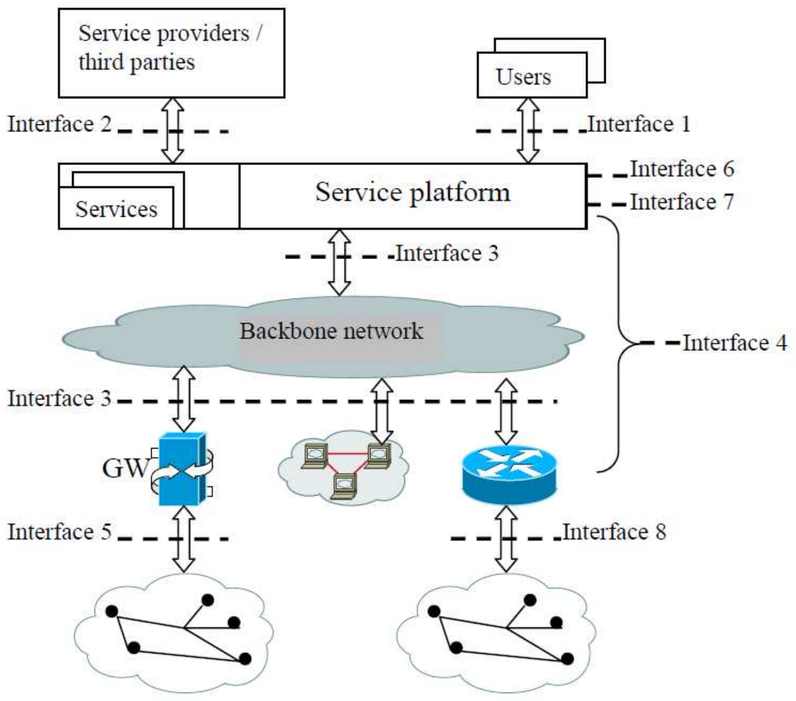


Figure 4.5: Detailed CO Architecture with Different Interfaces [28]

- *Stable* - Ensures all time availability of the platform.
- *Reliable* - Guarantees that the platform does what it is intended to do.
- *Flexible* - Promises easy configurations to add or modify new functionalities and
- *Scalable* - Assures that the platform is capable of taking high or small loads.

As discussed earlier, COOS architecture is designed in a modular way. Figure 4.6 depicts the modularization of COOS. As we can see from the figure, COOS is sub divided into two categories: Basic, Features and Advanced [15]. In this research, we focus mainly on COOS Basic and COOS Features. In the next two sections, we will discuss about these two categories.

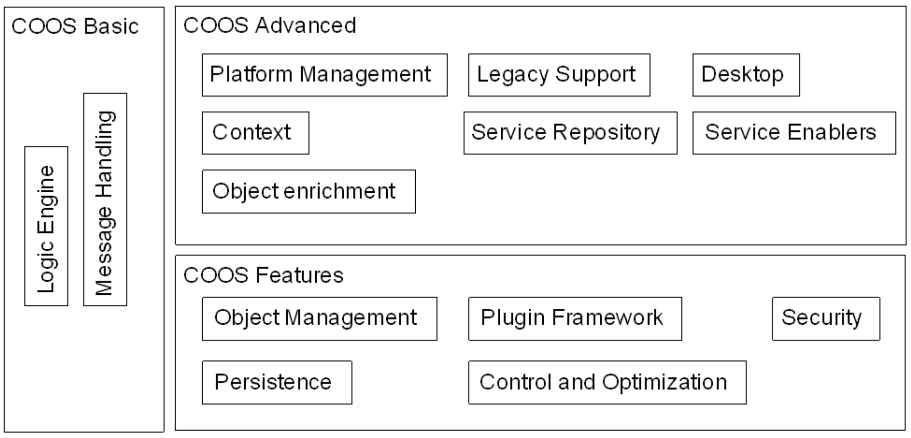


Figure 4.6: COOS Categories [15]

4.2 COOS Basic

Figure 4.6 presents an overview of the functionalities that COOS can offer. Of course not all these functionalities are required in every scenario. However, the COOS basic module needs to be present in all the implementations. In this section, we will mainly discuss the COOS messaging module from COOS Basic.

4.2.1 COOS Messaging

COOS offers a distributed point-to-point messaging service [26]. COOS messaging system is very flexible, highly configurable and it supports diversified routing algorithms. It is based on a process model that supports concurrent execution of processes [26, 15]. Figure 4.7 describes the COOS messaging system.

As we can see from this figure, the objects or rather the components are connected via the router network. Both the connected objects and routers form an overlay network. The figure also points out an interesting property of COOS messaging. If we analyze the router network, we can see that distributed point-to-point routing is offered. However, when we see the big picture, the routing system is centralized from the viewpoint of the objects/components. Each object/component is connected to a router and each

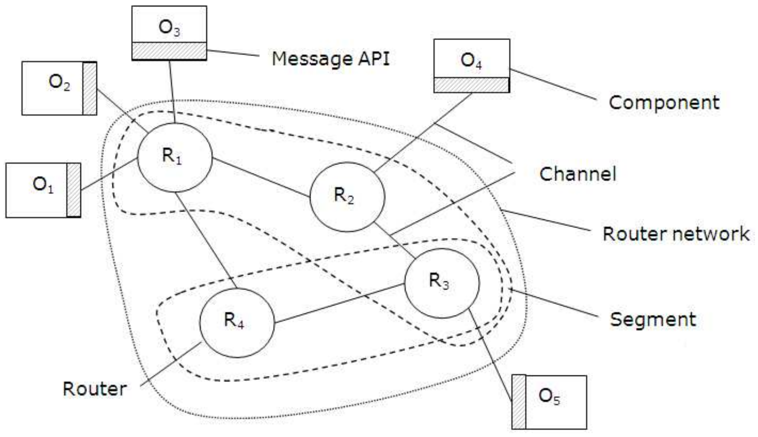


Figure 4.7: COOS Router Network with Connected Components [26]

of them connects with the routers via a Message API. This architecture makes the routing system scalable and fault-tolerant. Components communicate with each other using virtual links. The virtual links are mapped to a transport mechanism by COOS. As COOS routing is configurable, it can run several routing schemes concurrently depending on the topology or the transport requirements of different communications. Currently, COOS supports Link State Routing and Hierarchical Routing. The objects/components interact with each other through bi-directional channels. Depending on the cost of links, a message and its reply may be transported through different channels. There are also transport components in the routing system. These components are responsible for the transportation of the message between the channels and links. Transport components maintain a queue for the message delivery[15].

4.2.2 COOS Naming Service

As discussed earlier, COOS has its own naming service for identifying objects uniquely. When an object connects to the platform via an edge or bus, it is assigned a Universal Unique Identifier (UUID) [15]. The identifier itself contains vital information that helps the routing mechanism. The structure of the UUID is as follows:

UUID = <segmentA><segmentB><segmentC>....<endpointAddress>

As we can see, UUID contains hierarchical segment information and an endpoint address. The segment information contains information which indicates the network segment within which the object resides. The segment information helps the routing mechanism to route the messages at correct address. The endpoint address is a unique identifier that identifies the object within a network segment. One example of UUID is: uuid = coosPing.24eb78adef76892d. Here coosPing is the network segment of the uuid and its address is 24eb78adef76892d.

4.3 Features of COOS

From Figure 4.6 we can see that, COOS has five different modules or features. The features are described as follows:

- *Object Management* - Object Management deals with the management of the objects which comprises of User Manager, Access Control Manager, Lifecycle Manager etc. These are different management modules that perform the necessary object management tasks to ensure the connectivity of the objects. For example, the Lifecycle Manager (LCM) keeps track of the status and states of the objects. Objects register themselves with the LCM during their startup and deregister themselves when they shut down.

- *Persistence* - Persistence deals with data storage in the platform. Currently, COOS supports an Object-Relational Mapping library called Hibernate [15] as the data base. The basic idea is to store the persistence data and make it available for other objects. The objects can utilize the central database using their own proxy database manager.

- *Control and Optimization* - This module is responsible for policy management, load balancing and controlling. It has a Policy Manager that organizes the policies for different users, a Load Balancer which is responsible for load balancing tasks and a Controller which controls the link properties like Quality of Service (QoS), bandwidth, latency e.t.c

- *Security* - This module is accountable for providing the security properties of the platform. Security properties like confidentiality, integrity or non-repudiation are the concerns of this module.

- *Plugin Framework* - Plugin framework is a generic framework that helps objects to build edges to connect to the platform. It is the entry point into the COOS. More detailed discussion on plugin framework follows in Chapter 5.

Chapter 5

Plugin Framework in COOS

Plugin Framework is the gateway towards the COOS system for the objects. As discussed earlier, objects get connected with the COOS system with the help of the edges. Plugin Framework comes to help building the edges towards the platform. Objects may use different technologies or standards. However there should be a generic way for all the objects to get connected with the platform. Plugin Framework promises a generic doorway towards the CO platform which should be independent of technology (Java, .Net, C/C++ etc) [15].

As COOS plugin framework aims to be independent of the technology, it will be easy to integrate objects supporting different standards. Figure 5.1 depicts the role of the plugin framework in COOS. Plugin framework helps to implement edges for objects or components. COOS specifications [15, 28] require that plugin framework should not only allow components that wish to connect to the platform using native COOS protocols, but also to allow other components or objects to connect to the platform which are not using COOS protocols. Of course, this means a well defined protocol needs to be established so that components that are not familiar with the COOS protocol, can understand how to talk with the platform.

COOS specifications [15, 28] discuss very little about the Plugin Framework. As stated earlier, there is also no documentation of the COOS platform which is implemented in JAVA. In this research, we have tried to visualize the design and functionalities of the COOS plugin framework by analyzing the codes. In the next section, we will discuss the features of Plugin Framework.

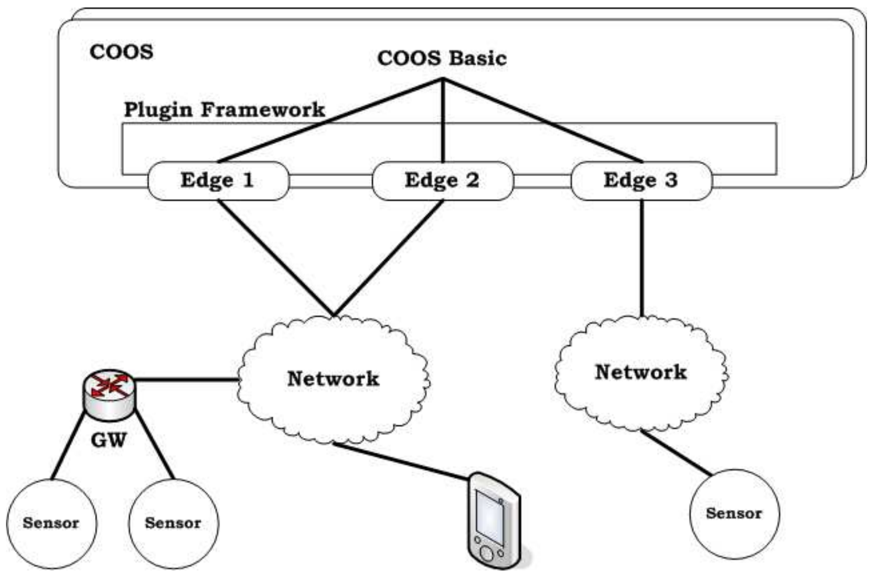


Figure 5.1: Plugin Framework

5.1 Features of COOS Plugin Framework

One of the important functionalities of plugin framework is the management of different plugins or edges. Plugin framework provides a SDK for plugin development and management. Actually, plugin framework provides some interfaces in a plugin package that must be implemented in order to create and install own plugins.

Plugin framework offers consumer producer style for the plugin development [15]. This style is very easy to understand and flexible to implement. Producers produce a message and deliver it to the framework for transportation. The framework transport the message using the message bus and Consumers consume the message from the framework. Consumer-Producer style of message communication is very suitable for functions like writing information to file or sensor communications. The interfaces provided by the SDK also contain default implementations. So the plugin developers need to extend the default implementation and add their specific functionalities of interest. SDK provides four different actors or interfaces. These are discussed as follows:

- *Producer* - As discussed earlier, Producer produces message and passes it to the message bus through the plugin framework. One interesting thing to notice is that, a producer seems like a consumer if seen from a connected object's viewpoint, since it consumes information from the object. However, COOS developers choose the name Producer as it is seen from the Message bus side [15].

- *Consumer* - The function of Consumer is straightforward. It consumes the messages from the Message bus.
- *Endpoint* - Endpoint deals with the registration, authentication and life cycle management of the plugins. Endpoints are also responsible for creating producers and consumers during plugin development.
- *Exchange* - Exchange interface helps the framework to transport messages from and to the message bus. It has some exchange patterns which is used by the framework to synchronize with the bus [15]. In-Out is an example of exchange pattern which tells the framework that the consumer is supposed to consume a message and send a response in reply. In-Only does not require the consumer to send any response. In Optional-Out, Out-Only, Out-In and Out-Optional-In are the remaining exchange patterns which are defined in [39, 15].
- *Interaction Helper* - This is a simple helper module which helps the framework to send and receive messages according to different exchange patterns.

The default implementations provided in the SDK contains lots of underlying functionalities which are required for plugin development. As a result, it becomes very easy for the developers to develop new plugins without worrying much about the framework protocols or functionalities. [39] presents a nice PingPong example which helps to understand this whole scenario in more detail.

5.2 COOS Plugin Framework in JAVA

Currently, COOS has a JAVA version of the Plugin Framework. It is not designed collectively in one class. The functionalities of the plugin framework are distributed in different classes. As there is no specific discussion on the design or the functionalities in the specifications, this research work has been directed towards understanding the working principles of the Plugin Framework by looking at the classes themselves. In this section, we will try to give an abstract view of the functionalities of the plugin framework.

Most of the major functionalities of the plugin framework are implemented in the following classes: *DefaultEndpoint()*, *DefaultMessage()*, *SerializerFactory()*, *PluginChannel()* and *DefaultChannelServer()*. Apart from these classes Plugin framework also uses some other channels for other less important tasks. These aforementioned classes perform the following functionalities:

- *DefaultEndpoint()* - This class performs many important functions and it is the base class not only for the Plugin framework but also for other components of COOS.As mentioned earlier this class is responsible for creating Consumers and Producers for plugins. It also takes care of the login, authentication and lifecycle management of the plugins. Apart from that it helps to set the endpoint UUIDs, informs the log system about the current state of processing, helps to create exchange patterns, initializes the processing of messages, initializes an endpoint and much more. Actually, a lot of cementing has been performed in this class so that creating and maintaining plugins, processing of messages and logging can be performed effectively.

- *SerializerFactory()*- Serialization and Deserialization are two of the most essential functions of the plugin framework. In the context of data communication, serialization deals with converting a message into a sequence of bits suitable to be transmitted over the wire across the network. Deserialization performs the exact opposite role - converts the bits to a message according to a message structure. The *SerializerFactory()* factory in COOS deals with the serialization and deserialization of messages. Currently COOS supports Actor Frame (AF) serialization and JAVA serialization. AF serialization means the serialization has to be implemented by own mechanism rather than using the default serialization methods of programming languages like JAVA or C#. The *SerializerFactory()* initializes the The *ObjectSerializer()*. *ObjectSerializer()* has six helper classes for six different primitives. These are:

  1. *ObjectHelper()* - Chooses the proper helper class for serialization/deserialization from the object primitve.
  2. *ArrayHelper()* - Serializes/Deserializes arrays of data (String array, Byte array, Integer array)
  3. *StringHelper()* - Serializes/Deserializes string type data.
  4. *IntegerHelper()* - Serializes/Deserializes integer type data.
  5. *HashtableHelper()* - Serializes/Deserializes hash tables.

6. *VectorHelper()* - Serializes/Deserializes Vectors(which can contain any Java primitives i.e byte, int etc).

- *DefaultMessage()* - This is the most important component of the plugin framework. This class deals with the processing of incoming or outgoing messages. The transport mechanism of COOS delivers the message to this class for further processing. Similarly, this class delivers the final message to be communicated to the transport system. COOS has a special format for constructing messages which is defined in a special protocol named *Connect* Protocol. *DefaultMessage()* class constructs the message according to the *Connect* protocol. After constructing the message, it pipes the message to the appropriate serialization method. Similarly, for an incoming message it breaks up the message, extracts the header information, cuts down the serialized message and pipes it to the deserialization engine. More discussion on constructing the message follows in Chapter 7.

- *DefaultChannelServer()* - *DefaultChannelServer()* implements the server part of the *Connect* protocol. This class processes messages from the clients or objects and produces the reply message. For example, *DefaultChannelServer()* produces the *Connect Acknowledgement* message in reply of a *Connect* message from an object. It also sets up the channel properties for message transportation.

- *PluginChannel()* - The client part of the *Connect* protocol is implemented in *PluginChannel()* class. It starts the connection and processes the reply messages from the server. It also sets up important connection parameters according to the reply from the server. In case of an outgoing message, for example *Connect* this class builds up the payload of the message and initializes the *DefaultMessage()* which then adds up the headers and pipes it to the serialization mechanism.

It is important to note that, the COOS specifications does not contain any documentation about the *Connect* protocol. This protocol defines the message structure and thus it is very important for the developers to know the rules to construct a message. Otherwise the objects cannot send and receive messages correctly. Moreover, the logging system in COOS is also a bit stiff in case of connection establishment and message passing. It does not produce any response in case a message with an unknown format is passed to it. In such case, it becomes really difficult to figure out what went wrong.

In this chapter, we have discussed about the structure and functionalities of COOS plugin framework in brief. This information is very essential for developing technology independent plugin framework for connected object communication. In the next chapter, we will discuss about the requirements of a small footprint version of COOS plugin framework which is developed in .NET platform using C# as the programming language.

Chapter 6

Requirements of a Small Footprint Version of COOS Plugin Framework

The aim of this research is to produce a small lightweight version of the COOS plugin framework for objects that support the .Net platform. This version of the plugin framework will be used by the application logic running in an object in order to get connected to the platform. We have already seen the properties and functionalities of the COOS plugin framework in JAVA which is discussed in Chapter 5. In this chapter, we discuss the requirements for a small footprint version of COOS plugin framework that will eventually run on .NET platform. There are some requirements which must be met and some other requirements which are optional. The optional requirements enhance the functionality and flexibility of the plugin framework. We list all the requirements in the Table 6.1 according to their priority.

The priorities are classified into two catagories: *Basic* and *Optional*. The *Basic* functionalities must be present in a deployment of a plugin framework and so these requirements must be met. *Optional* requirements are not necessary for each deployment of a plugin framework. However, if these requirements are fulfilled, they make the plugin framework fully compatible with the CO service platform and they greatly enhance the usability of the framework. It might not be feasible to implement all the *Optional* requirements because some of them might exceed the limit of memory usage or processing power of small handheld devices. It will depend on the designer how much extra functionality he wants to deliver keeping the device constraints in mind.

Table 6.1: Requirements of the COOS Plugin Framework

	Requirement	Priority
1	Message Construction	Basic
2	Serialization	Basic
3	Deserialization	Basic
4	Connection Establishment	Basic
5	Message Communication	Basic
6	COOS Plugin Support	Optional
7	COOS Node Support	Optional
8	COOS Configuration Support	Optional
9	Energy Efficiency	Optional
10	Small Size	Basic

6.1 Basic Requirements

As discussed earlier, the basic duty of the plugin framework is to help an object to create edges towards the CO service platform. It is also responsible for sending and receiving the messages on behalf of the object. Apart from these two essential tasks, plugin framework also maintains the connectivity with the CO service platform. Figure 6.1 represents the basic workflow of the plugin framework. A plugin framework must perform these functionalities in order to enable an object's communication with the CO service platform. We will discuss these functionalities and the requirements in details in the following sections.

6.1.1 Constructing a Message

Upon receiving the message payload from the application logic, the plugin framework should reconstruct the message adding extra header information. The message format is defined in the *Connect Protocol.* The format of a message is depicted in Figure 6.2

As we can see, plugin framework needs to add a lot of extra information to aid message transportation and message processing. A short description of the message fields are as follows:

- *Message Length* - The total length of the message in bytes (Integer).

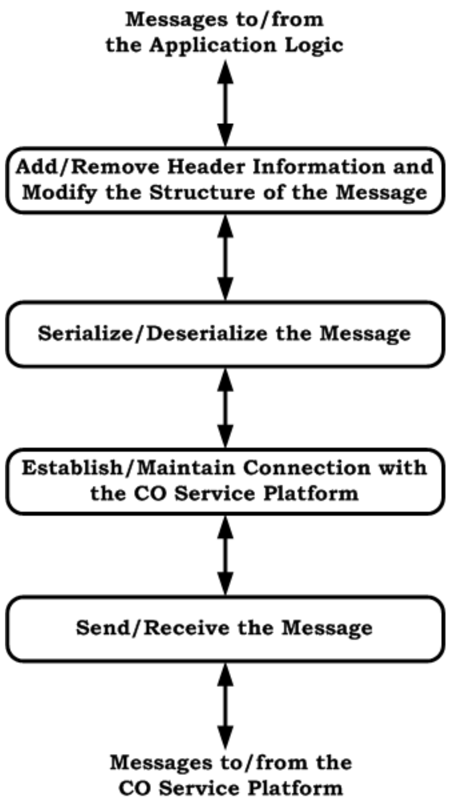

Messages to/from the Application Logic
Add/Remove Header Information and Modify the Structure of the Message
Serialize/Deserialize the Message
Establish/Maintain Connection with the CO Service Platform
Send/Receive the Message
Messages to/from the CO Service Platform

Figure 6.1: Plugin Framework Workflow Diagram

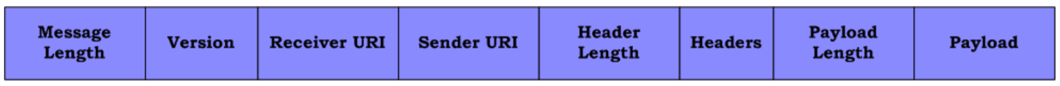


Figure 6.2: Message Format

- *Version* - Version of the plugin framework (1 byte).
- *Receiver Endpont URI* - The Uniform Resource Identifier (URI) or the UUID of the Receiver (String).
- *Sender Endpoint URI* - UUID of the Sender (String).
- *Header Length*- Number of headers to follow (Integer).
- *Headers* - Headers including type of the header and the value of the header (String).

- *Payload Length* - Length of the actual payload message in bytes (Integer).
- *Payload* - Actual payload message (String, Byte or Object).

6.1.2 Serialization/Deserialization

As discussed in Chapter 5, one of the vital duties of plugin framework is to serialize and deserialize the message. Serialization/Deserialization assists message transportation. We already know that COOS supports two types of serialization: Actor Frame or own serialization which is also the default serialization and the JAVA built in serialization. However, in this case, we require the plugin framework to run on .NET platform. So, it is required that own serialization and deserialization method is implemented. While developing the serialization and deserialization mechanism, one should also keep in mind that this mechanism should be compatible with the COOS serialization/deserialization methods. This is simply because the COOS platform has to deserialize the message that is serialized using the own serialization mechanism of the .NET version of the plugin framework.

6.1.3 Establish/Maintain Connection with the CO Service Platform

Plugin Framework should establish and maintain connectivity with the CO service platform. CO service platform always listens on port **15656**. It means that the CO service platform offers services and connectivity through this port. So, the plugin framework should implement some sort of socket programming in order to connect to the platform. It also has to use efficient streaming mechanisms to send and receive message from the sockets. As we know, COOS supports the general TCP/IP protocol, plugin framework should maintain the connectivity using TCP transport mechanisms.

6.1.4 Send and Receive Message

The most important activity of the plugin framework is to send and receive messages on behalf of the connected object. It means the plugin framework has to establish proper links and channels, maintain the connection state and safeguard the message communication from the sender to the receiver. Depending on the exchange pattern of the messages, plugin framework may or may not wait for a reply. Upon receiving the message, it should pipe the message to the deserialization mechanism and inform the object about the content of the message.

6.1.5 Small Size

Size is a very crucial factor while designing a plugin framework for specific devices or objects. Depending on the constraints put on the size of the plugin framework, optional requirements may or may not be offered in a plugin framework design. The constrictions in this case can be the processing capability of the object, the available memory space and the battery life of the object. If there are limitations like these, a plugin framework that supports all the basic and optional features mentioned in Table 6.1 may not be a feasible solution. In our case, we are designing a small footprint version of the plugin framework which will act as a prototype. In situations like this, system designer should put on strict restrictions on size and design the framework in such a way which will be small and efficient at the same time.

6.2 Optional Requirements

The optional requirements stated in Table 6.1 identifies the adaptable features of COOS plugin framework which will enhance the reusability and efficiency of the framework. We will discuss these features in brief in this section.

6.2.1 COOS Plugin Support

As stated in Chapter 5, COOS plugin framework provides an SDK for plugin development and it prefers the consumer producer style of plugin development. So, if the plugin framework provides such kind of plugins with already defined interfaces, it will be very helpful for the developers to use the interfaces and plugins and build their plugins on top of these skeletons. In such case, the developer needs not to worry about the class diagrams or message constructions. All he needs to do is to develop a producer and a consumer according to his own specific requirements and make the necessary method or class invocation. The rest of the job is taken care of by the underlying plugins. This feature will greatly enhance the efficiency of the framework.

6.2.2 COOS Node Support

It is also helpful if the plugin framework support the standard COOS nodes. By standard COOS node we mean a node which is deployed in COOS and possesses all the functionalities which is required from a COOS node. If other than developing own nodes to interact with the CO platform, the developer chooses to develop the nodes according to COOS specifications, it will be interoperable with the COOS system and the communication efficacy will increase.

6.2.3 COOS Configuration Support

If the plugin SDK supports the COOS configurations, it will simplify the object integration with the platform. By COOS configuration we mean, all the necessary protocols and specifications which are required for an object to build edges towards the CO service platform. Of course it is completely upto the system designer what configuration he will include or support in the plugin framework for easy object integration. The end goal is always to allow objects to get connected and make accessible through the platform.

6.2.4 Energy Efficiency

Although we have considered energy efficiency as an optional requirement, it has become apparent that it is one of the most crucial features that a plugin framework should contain. The plugin framework developed for the objects are meant to run on the object's machine. As COOS targets small hand-held devices to provide seamless connectivity, the plugin framework designer should pay serious attention to the limited capabilities of the small handheld embedded devices. Some optional requirements may not be considered to be present in the plugin framework if it is developed for such a device which does not support extra processing or memory space. And the battery life of such a device is also limited. So the energy efficiency of the plugin framework should be considered seriously.

Chapter 7

Design and Implementation of a Small Footprint Version of COOS Plugin Framework

One of the essential objectives of this thesis is to produce a small footprint version of the COOS plugin framework that will enable devices (that support .NET Platform) to connect to the CO service platform. So far we have discussed the functionalities of the COOS plugin framework and the requirements of a plugin framework that will aid an object to connect to the platform. In this chapter, we will present the design of our plugin framework. We will also discuss the implementation procedure followed to develop such a plugin framework.

7.1 Design of the Plugin Framework

The design of the small lightweight version of the COOS plugin framework straightly follows from the workflow diagram (Figure 6.2) presented in Chapter 6. It is presented in Figure 7.1. This small lightweight version of the plugin framework performs all the basic functionalities which is required to help and object to build an edge towards the CO service platform and send and receive messages.

In this design, the application logic running on an object needs to invoke the plugin framework. The plugin framework then establishes connection with the platform. Although the design does not suggest any specific transport protocol, standard TCP transport is preferred. COOS also supports secured

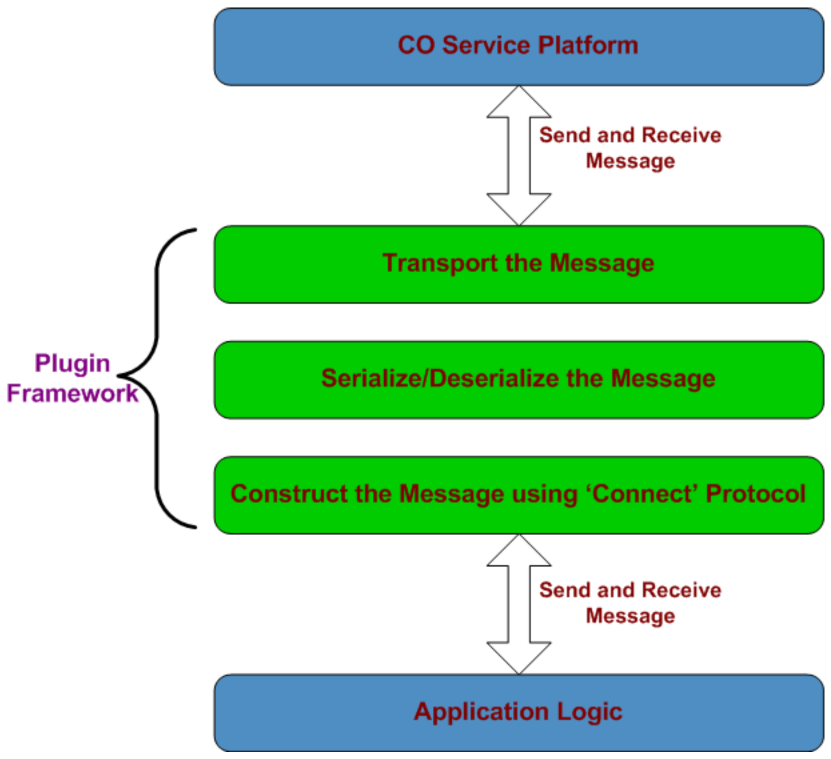


Figure 7.1: Design

TCP transport with end-to-end security like IPSec. However, for this simple lightweight plugin framework we have only considered standard TCP transport for the time being. During the design phase, we tried to develop our own transport mechanism for the client (object) side communication with the platform. However, sooner it became relevant that it is not compulsory to develop own transport mechanisms. Ofcourse, it will provide more flexibility and probably more efficiency. However, it will also introduce complexity in the design and eventually in the implementation phase. And, our goal was to keep the design as simple as possible. So, we kept the design and partial implementation of the transport mechanism aside and went on with the normal socket programming concepts. This is possible because, CO service platform is always listening on port 15656 and waiting for objects to connect to it using legitimate *Connect* protocols.

This plugin framework also constructs the message according to the format defined in the *Connect* protocol. Next section presents an example of how the messages are constructed in the actual implementation. The serialization and deserialization block helps to serialize or deserialize data suitable for transportation over the communication medium and transport block simply communicates messages back and forth between the object and the platform using the standard socket programming and streaming mechanisms. We do not discuss the design again here as the functionalities of each of the blocks are discussed in Chapter 5 and Chapter 6. In the following section, we will discuss the implementation procedure.

7.2 Implementation

The plugin framework is implemented using *Visual C#* in the *Microsoft Visual Studio 2008* environment. C# is an object-oriented language that offers a platform for simple, feasible and flexible programming environment. It is one of the popular initiative of Microsoft .NET. As COOS plugin framework already has a working version in JAVA, the choice of programming language was more towards classical C based language. C# is chosen because it not only offers object-orientedness but also the Visual C# makes it much easier to develop nice and simple Graphical User Interfaces (GUI).

7.2.1 Class Diagram

Figure 7.2 represents the Class Diagram for our implementation. It includes the interfaces, classes and essential methods (methods are shown in a filtered view). As depicted in the figure, our implementation consists of seven classes and two interfaces. The main class which invokes all other classes in this implementation is the *DefaultMessage()* class. In the following subsection we will discuss these classes and their methods in brief.

7.2.1.1 DefaultMessage() Class

This class implements the interface defined as *Message()*. *DefaultMessage()* not only implements the two base methods defined in *Message()* namely, *deserialize()* and *serialize()*, but also implements some other essential methods like *sendAndReceive()*, *sendConnectMessage()* and *sendMessage()*. The functionalities of each of the methods are discussed as follows:

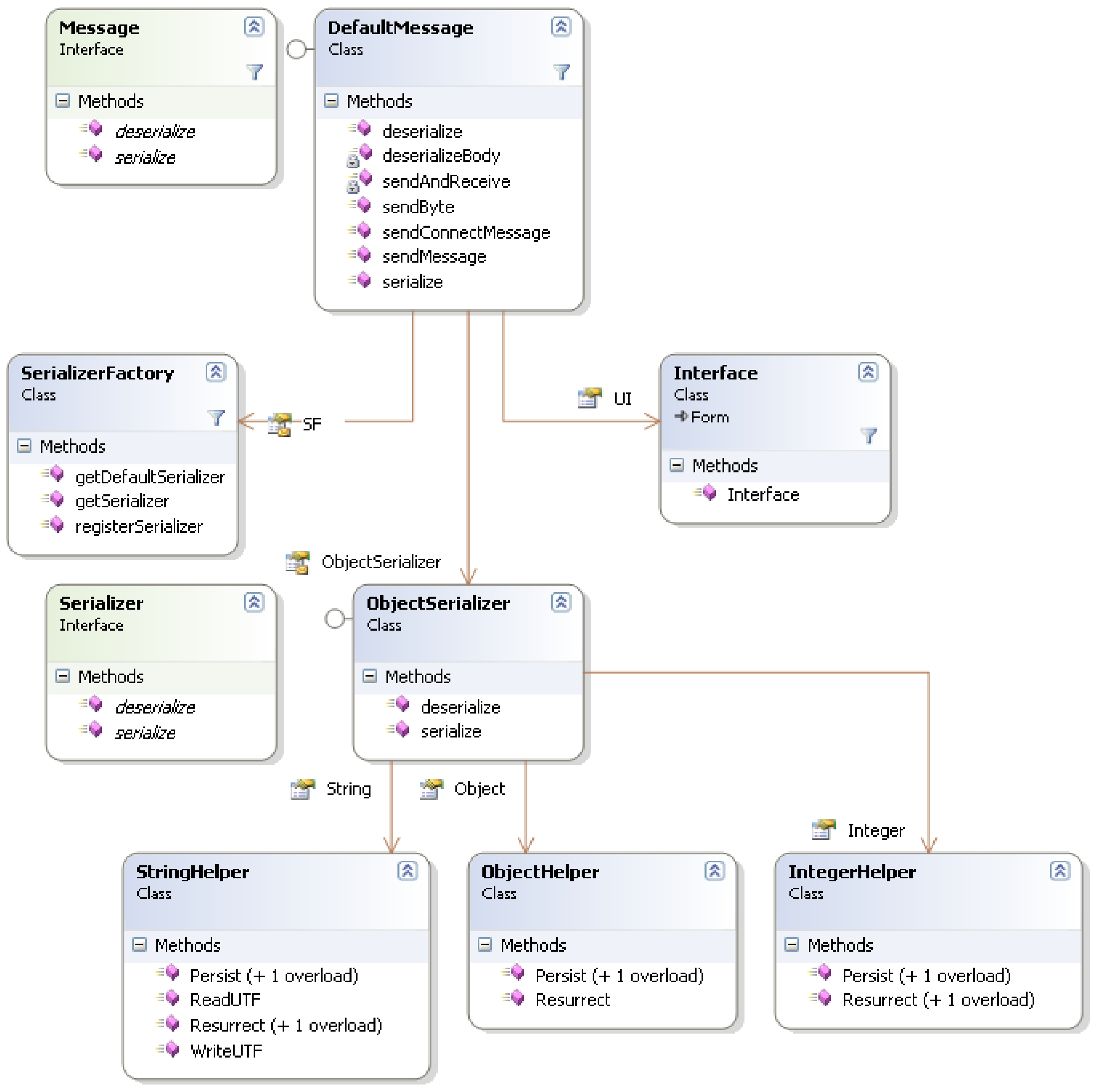


Figure 7.2: Class Diagram

- *deserialize()* - Deserializes the whole message along with the header information. It extracts the body from the message and constructs the header.
- *deserializeBody()* - Deserializes the body of the message only.
- *sendAndReceive()* - As the name suggests, it is responsible for sending and receiving the final message with headers. While sending, this class is invoked with a *byte* array which is the serialized message. Similarly,

while receiving data on behalf of the framework, this class fetches an array of bytes from the network stream and invokes the deserialization engine.

- *sendConnectMessage()* - Constructs the connect message with the necessary headers and an empty body. Message construction is discussed in Section 7.2.3 in more detail.
- *sendMessage()* - Constructs a string message with necessary headers.
- *serialize()* - This method is responsible for serializing the message. It communicates with the class *SerializerFactory()* to determine the serialization type (default or AF) and invokes the serializer engine.

7.2.1.2 SerializerFactory() Class

The serializer factory class is responsible for registering and maintaining the serialization methods. It maintains a hashtable which contains entries for all the supported serialization methods or mechanisms. The method description for this class is as follows:

- *getDefaultSerializer()* - Returns the name of the default serializer mechanism.
- *getSerializer()* - Returns the current serializer mechanism for a session.
- *registerSerializer()* - Registers a serializer mechanism that can be used for serialization and deserialization of a method. Currently, our implementation only supports the own serialization or default serialization mechanism.

7.2.1.3 Interface() Class

This class is responsible for the user interface invocation and connection establishment. Our implementation provides a simple user interface for sending and receiving connect message, string or byte. Connection establishment and user interface design are discussed in detail in Section 7.2.2 and Section 7.2.4 respectively.

7.2.1.4 Object Serializer()

Object Serializer() class implements the *Serializer()* interface. This class is onvoked by the *DefaultMessage()* class with the body of the message. *Object Serializer()* class determines the type of the data and according to the type it invokes one of the three helper classes *StringHelper()*, *IntegerHelper()* and *ObjectHelper()* class. The *deserialize()* and *serialize()* method invoke the serialization and deserialization mechanism for the specific data type. These mechanisms are implemented in the helper classes.
It should be noted that, although our implementation supports the communication of *byte* datatype, we have not implemented any *ByteHelper()* class. The reason is that, after implementing *IntegerHelper()* it became apparent that sending and receiving a byte is not so cumbersome so that it should be implemented in a class. As a result, we have implemented it in a simple method named *sendByte()* in the *DefaultMessage()* class.

7.2.1.5 StringHelper(), ObjectHelper() and IntegerHelper() Class

These helper classes perform the actual serialization and deserialization of the message according to the data type contained in the body. These classes accept two parameters: the data to be serialized/deserialized and an instance of the network stream. As these classes perform more or less the same functionalities they are discussed together here. All of these helper classes implement two very essential methods: *Persist()* and *Resurrect()*.

- *Persist()* - This method is responsible for the serialization of the message. This method takes the message to be serialized, and writes the serialized bytes to the network stream. In case of simple primitives like *Integer*, this method directly writes the serialized data to the stream. However, in case of *String* it uses another method *WriteUTF()* to write the data in Unicode Transformation Format -8 (UTF-8). *WriteUTF()* method is discussed in Section 7.2.3 in more detail.

- *Resurrect()* - This class performs the opposite duties with respect to the *Persist()* method. It receives the serialized data as byte array and deserializes it to a primitive of object, string or integer type. It assumes that the first byte in the message indicates the data type of the primitive contained in the message. In case of string, it uses *ReadUTF()* method to read the data in UTF-8. *ReadUTF()* method is also discussed in Section 7.2.3 in more detail.

7.2.2 Connection Establishment

As mentioned earlier, CO Service Platform listens on port 15656, accepts request for connection and provides services. In order to establish an edge towards the platform, we have used simple socket programming mechanisms of C#. The following line is an example of simple client request for TCP Connection towards a server which is running on localhost and listening on port 15656.

```
TcpClient Client = new TcpClient("localhost", 15656);
```

7.2.3 Serialization/Deserialization and Message Construction

We will discuss the serialization/deserialization ahead of message construction as it includes some important concepts which will simplify the discussion of the message construction. In our implementation, serialization/deserialization and message construction are performed in conjunction. COOS system serializes/deserializes strings using UTF-8. UTF-8 is widely used in case of stream communications. It is very flexible and it is also backward compatible with ASCII.
COOS plugin framework uses the JAVA built in *readUTF()* and *writeUTF()* methods provided by the *DataOutputStream()* class, which read and write data in UTF-8. However, C# does not have such built in methods for writing and reading data in UTF-8 format. So, we have developed own method for writing and reading data in UTF-8 format. The methods are as follows:

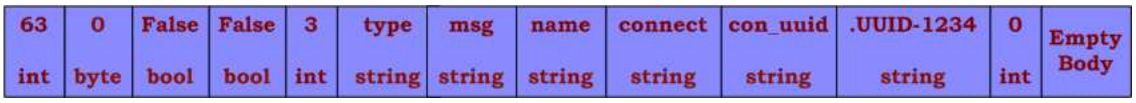
```
public static void WriteUTF(BinaryWriter bw, string str)
        {
            ushort size = (ushort)(str.Length);
            bw.Write((byte)(size >> 8));
            bw.Write((byte)size);
            bw.Write(str.ToCharArray());
        }

public static string ReadUTF(BinaryReader br)
        {
            ushort size = 0;
            size += (ushort)(br.ReadByte() << 8);
            size += br.ReadByte();
            char[] s = br.ReadChars(size);
            return new string(s);
        }
```

The methodology for writing data in UTF-8 format is pretty simple. The *WriteUTF()* method is provided with the *BinaryWriter* instance which writes data streams to the socket connection and the string data. The first three lines of the *WriteUTF()* method writes the size of the data according to the UTF-8 specifications [11, 14]. The last line in this method writes the string data to the stream. The *ReadUTF* method reads the size of the data according to the UTF-8 specification and then reads the data itself.

As stated in Chapter 5, COOS plugin framework basically supports both own serilization and Java serialization methods. For obvious reason, we have used the own serialization mechanism. At the startup, our main objective was to send connect message and send and receive strings. However, our implementation also supports the serialization/deserilization of integers, objects and bytes.

Message construction has been performed according to the message format presented in Figure 6.2. A *Connect* message is required to connect to the CO service platform. The connect message is constructed and serialized as follow:



63	0	False	False	3	type	msg	name	connect	con_uuid	.UUID-1234	0	Empty Body
int	byte	bool	bool	int	string	string	string	string	string	string	int	

Figure 7.3: Connect Message

The first integer part '63' indicates the size of the total message. It is followed by a byte '0' which indicates the version. The two boolean values 'false' indicate that this message does not contain any sender and receiver endpoint URI. Next part is an integer '3' which indicates the number of headers. The headers are all string values. Headers indicate the 'type' of the message which is 'msg', the 'name' of the message which is obviously 'connect' and the connection UUID 'con_uuid' which is '.UUID-1234'. Headers are managed by *Hashtables* with the type of the header as its key and the value of the header as the value of the corresponding key. The headers should be followed by the body of the message. *Connect* message generally has an empty body. This is indicated by an integer '0' which reflects the length of the body.

In case of writing integer values we faced a critical problem. COOS system was unable to read the size of our version of connect message and our plugin framework failed to detect the size of the message coming from the other size. After a little bit of research we found out that the problem was regarding the format of the integer which is handled differently in JAVA and in C#. JAVA platform and other network protocols generally handle integer in *big-endian* format. However, Microsoft .NET platform deals with *little-endian* format. So, we have changed the byte order of all the integer going in and out from our plugin framework, to make the system compatible with the COOS plugin framework. One of the useful C# method that we have used in this regard is *IPAddress.HostToNetworkOrder()*. For example, if we want to write integer '3' to the stream, we write it like this:

```
dout.Write((int)IPAddress.HostToNetworkOrder(3));
```

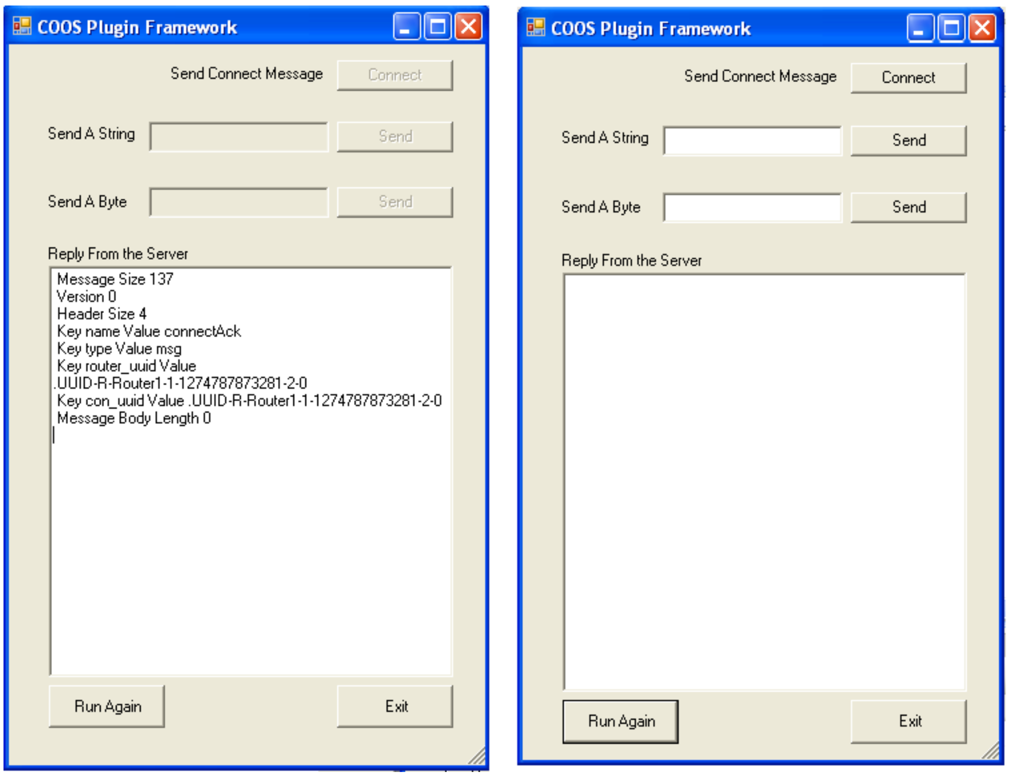
Detailed source codes are included in Appendix A.

7.2.4 User Interface

Initially there was no user interface for our version of the plugin framework. We communicated with the CO service platform using the console interface. In later stages, we have developed a simple user interface for sending and receiving 'connect', 'string' and 'bytes'. It also shows the reply message from the server. The user interface is depicted in Figure 7.4.

The user interface on the right side is the starting interface as we run our plugin framework. As we can see from the figure (Figure 7.4), the plugin framework allows to send and receive connect message or a string or a byte. The user interface on the left hand side depicts the scenario when a connect message has been sent to the server. The reply message is shown in the dialogue box. The server acknowledges the *Connect* message with an *Connection Acknowledgement* message. It also sends back the router information that will be used to send and receive messages for this session.

So, in short, we have developed a simulator for the plugin framework which helps objects to connect to the CO service platform and sends and receives messages. Currently our system only supports the communication of strings and bytes. It is only a small lightweight version of the plugin framework that demonstrates that it is possible to connect objects that support different



```
Message Size 137
Version 0
Header Size 4
Key name Value connectAck
Key type Value msg
Key router_uuid Value
.UUID-R-Router1-1-1274787873281-2-0
Key con_uuid Value .UUID-R-Router1-1-1274787873281-2-0
Message Body Length 0
```

Figure 7.4: User Interface

technologies and standards.

Chapter 8

Conclusion

8.1 Discussion

One of the easy conclusions that comes out from this thesis is that M2M communication has a huge potential as an emerging technology which can very easily conquer the consumer market. However, we have also realized that there are various challenges regarding this field which are yet to be addressed. It is felt that, in the field of communication technology everybody is talking about the huge potential of M2M communication and trying to visualize its potential on their own way. This scarce distribution of interest and realization is lacking the proper collaboration and standardization in this field. Such lack of common standards is very harmful for an emerging technology. Until very recently, nobody has got the clear picture of the possibilities and market scenario of M2M communication. This has been the main reason behind the lack of interest of investing money for the stake holders and customers.

The situation has started to change for some recent appreciable research efforts like COOS. COOS is a very significant effort towards providing a sort of standardization to M2M communication. With such initiatives, customers are getting a clear picture of the whole system. Each party now atleast knows how to approach in this field and what is the possible outcome of their investment. COOS can be considered as one of the first steps towards a standardized system of M2M. As we have seen in Chapter 2, the opportunities for developing new services for M2M are almost endless. History has shown that emerging technologies with such endless possibilities ends in vain in many situations just for the lack of standardization. For this reason, service providers, stake holders, network operators and all the other parties involved in M2M communication scenario should seriously signify their goals and requirements and stick to one common standard. Moreover, the service providers should design exciting services keeping the customer demand in mind. The success of M2M communication greatly depends on one or two stable and exciting applications or services that will hit the consumer market like 'Google' or 'Facebook'. If the service providers can design such service where the *winner takes it all*, M2M communication has enormous possibilities for all the stake holders.

The idea of connecting every object through the internet has also opened new doors of opportunities for telecommunication network operators and service providers. As a result, significant research is now being performed to figure out the mobile M2M possibilities. With already established networks of mobile devices, it does not seem difficult to expand the reach of telecommunication networks to provide connectivity to all the embedded devices. COOS targets the embedded mobile devices and objects for providing a common platform for communication. Chapter 3 presents the available programming languages and OSs for mobile/embedded device programming. One of our initial targets was to implement a working version of plugin framework to run on a windows enabled device. Thats why we have focused more on the opportunities available for windows enabled embedded device programming.

Developing a small footprint version of the COOS plugin framework in .NET platform was a challenging task. Initially, it seemed a very tough to identify the functionalities of the plugin framework by just analyzing a large code base consisting of hundreds of files containing thousands of lines of source codes. However, this challenging task has enabled us to get a better overview and understanding of the functionalities of COOS. In Chapter 5 we have discussed about the plugin framework in detail. Although the COOS specifications ([15, 28]) present the features of the plugin framework in detail, they contain very less discussion on the actual functionalities of the plugin framework and the implementation in Java. As such, the discussion presented in Chapter 5 on the actual implementation, highlights our conceptualization about the actual design.

In Chapter 6 we have presented the general requirements for a plugin framework which will aid an object to connect with the CO service platform. Although, we have implemented the plugin framework in .NET platform using

C# as the programming language, the requirements of a plugin framework presented in Chapter 6 are independent of the underlying technology. These requirements are generic with respect to any platform or device or programming language. Any system designer who wants to design a new plugin framework for any other platform, can design the framework following these requirements.

It is very important to note that, although we have implemented a small footprint version of the COOS plugin framework, this framework is meant to run on the object's device. The COOS plugin framework will still run on the server side. Our version of the plugin framework will run on a client device which supports .NET platform and will interact with the COOS plugin framework in order to establish and maintain the device's connectivity with the CO service platform. The initial goal of this thesis was to implement a small prototype version of the COOS plugin framework. This goal has been achieved and we have shown the simulated results in Chapter 7. Table 6.1 presents the requirements of a general plugin framework. We have not implemented all of the required features in our versions. Our implementation only addresses the basic requirements which are needed to aid an object's connectivity with the platform. Table 8.1 depicts the features which we have implemented. If we compare Table 6.1 and Table 8.1, we can see that our version of the plugin framework implements only the basic requirements. We have not addressed the optional features because of time constraints and also because we never intended to implement a fully functional version of the plugin framework. This of course leaves space for future work.

In Chapter 2 and Chapter 4 we have discussed how the lack of a common platform for connected objects was hindering the growth of M2M communication. COOS is a significant research initiative from Telenor Research Group which not only provides a platform for the objects to communicate with each other but also allows the service providers to develop new exciting services and offer the services to the connected objects through the platform. Being an open source project, COOS will significantly aid the growth of the internet of things and it will take the M2M communication one step ahead. However, during this thesis we have felt that lack of documentation and in some cases lack of protocol specifications will make the system designer's job a bit more complex. It's still early days for COOS and as it is an ongoing research project, we hope that COOS will provide more elaborative specifications and documentations. This thesis can be a good documentation for those who want to design and implement plugin frameworks for their own

Table 8.1: Achieved Requirements

	Requirement	Implemented
1	Message Construction	Yes
2	Serialization	Yes
3	Deserialization	Yes
4	Connection Establishment	Yes
5	Message Communication	Yes
6	COOS Plugin Support	No
7	COOS Node Support	No
8	COOS Configuration Support	No
9	Energy Efficiency	No
10	Small Size	Yes

platform and device.

8.2 Future Work

This thesis leaves space for some exciting future research regarding the plugin framework development for COOS. New questions have kept coming during this thesis. Is it necessary to include all the features required for a plugin framework (presented in Table 6.1) especially when it is meant to run on a embedded device which has limited processing power and battery life? Will it be feasible? Or is it possible to design and implement a platform independent plugin framework which will run on any device? Questions like these are yet to be answered and they are very significant for future research on COOS. It will be also interesting to see the behavior of our small version of the plugin framework on an actual device like *Nordic ID PL3000* which runs on Windows CE 6.0 platform.

We have tested our plugin framework by only sending *String* and *Bytes*. Although, our version supports the communication of *Integer* header properties and *Objects*, we have not tested them yet. In future, it will be interesting to see if it is possible to support the communication of all the primitives which are supported by the COOS plugin framework in Java. Future research may also be directed towards developing much more flexible and efficient plugin framework. This thesis has only shown that it is possible to develop a plugin framework in any platform and interact with CO service platform. So, it will be interesting to see whether it is possible to design a framework which provides all the functionalities like the COOS plugin framework and at the same time it is feasible to implement it on actual devices.

Regarding the M2M communication, future research can be directed towards developing new exciting services. As COOS is providing a common platform for the service providers also, it will be interesting to see how these new services cope up with the COOS platform. Security is another issue which is a crucial property of any system now a days. So, security mechanisms in COOS specifically in plugin framework should be carefully designed. As numerous devices are expected to connect with each other, the scalability of COOS platform will be an important issue to focus. Although COOS promises to be scalable both on high loads and low loads, experiments should be performed practically to see the performance of COOS platform both on high loads and low loads.

To conclude, we can say that through this thesis, we have realized that M2M communication is a thrilling technology with numerous challenges and exciting possibilities. COOS has emerged to provide a common platform for objects or things, to communicate with each other. The idea of connecting all the things with each other and allow them to communicate without any human intervention, which seemed a fairy tell even some years ago now appear very realistic for research initiatives like COOS. Recently Telenor has promised to provide a new platform *Shepherd* for aiding efficient communication of sensor applications [10]. *Shepherd* will use COOS as the key technology. This thesis is a small effort to aid the plugin framework development for COOS and provide a starting documentation for future research on plugin framework development for COOS. We hope that COOS will be a significant step forward towards efficient M2M communication.

Bibliography

[1] Android: Open Source Project . WWW page of Android: http://source.android.com/. [Accessed 28th May 2010.].

[2] Choosing a Programming Language for Windows Mobile Development . WWW page : http://msdn.microsoft.com/en-us/library/bb677133.aspx. [Accessed 28th May 2010.].

[3] COOS Open Source Community. WWW page of COOS Open Source Community: http://telenorobjects.com/shepherd/open-source-community.aspx. [Accessed 28th May 2010.].

[4] iPhone OS . WWW page of iPhone: http://www.apple.com/iphone/preview-iphone-os/. [Accessed 28th May 2010.].

[5] Jalimo. WWW page of Jalimo: https://wiki.evolvis.org/jalimo/index.php/Hauptseite. [Accessed 25th May 2010.].

[6] Java ME Technology. WWW page of J2ME: http://java.sun.com/javame/technology/index.jsp. [Accessed 25th May 2010.].

[7] Maemo. WWW page of Maemo: http://maemo.org/intro/platform/. [Accessed 25th May 2010.].

[8] Symbian. WWW page of Symbian: http://www.symbian.org/about-us/history-symbian. [Accessed 25th May 2010.].

[9] Telenor Objects. WWW page of Telenor Objects: http://telenorobjects.com/about-us.aspx. [Accessed 28th May 2010.].

[10] The Shepherd ő Platform . WWW page of Shepherd: http://telenorobjects.com/shepherd.aspx. [Accessed 28th May 2010.].

[11] UTF-8 and Unicode Standards. WWW page of UTF-8: http://www.utf8.com/. [Accessed 25th May 2010.].

[12] Windows CE Programming For Pocket PC In A Nutshell. WWW page : http://www.ece.northwestern.edu/realtime/windows.pdf [Accessed 28th May 2010.].

[13] Windows Embedded CE Technical Specifications. WWW page of Windows CE: http://www.microsoft.com/windowsembedded/en-us/products/windowsce/technical-specifications.mspxn. [Accessed 25th May 2010.].

[14] F. Yergeau. UTF-8, a Transformation Format of ISO 10646. Tech. rep., Alis Technologies, November 2003.

[15] Arild Herstad, Espen Nersveen, Jan Audestad, Geir Melby and Knut Eilif. Connected Objects Platform Specification (Internal Document). R & I Research Note, Telenor R & I, October 2008.

[16] Brown, A. Wireless Enterprise Stratergies: 2008 Market Outlook. Viewpoint snapshot, Stratergy Analytics, November 2007.

[17] Brown, A., and Moroney, J. A Brave New World in Mobile Machine-to-Machine (M2M) Communications. Forecast and outlook snapshot, Stratergy Analytics, July 2008.

[18] Chen, Y., and Yang, Y. Cellular based machine to machine communication with un-peer2peer protocol stack. In *Vehicular Technology Conference Fall (VTC 2009-Fall), 2009 IEEE 70th* (20-23 2009), pp. 1 –5.

[19] Conti, J. The internet of things. *Communications Engineer 4*, 6 (2006), 20–25.

[20] Curran, I., and Pluta, S. Overview of machine to machine and telematics. In *Water Event, 2008 6th Institution of Engineering and Technology* (22-23 2008), pp. 1 –33.

[21] Dr. Jens Struker, D. G., and Faupel, T. RFID Report 2008 Optimizing Business Processes in Germany. Tech. rep., Department of Telematics, Albert-Ludwig University, 2008.

[22] Enterprise, V. G. Global Machine to Machine Communication. White paper, Vodafone.

[23] Evans, C. Intelligent retail business: Location based services for mobile customers. In *Pervasive Computing and Applications, 2007. ICPCA 2007. 2nd International Conference on* (26-27 2007), pp. 354 –359.

[24] Ferreira, J. C. Sensor Telecos- New Business Oppurtunities. D1-main technology trends, capabilities of devices and service examples, EURESCOM, March 2006.

[25] Grønbæk, I. M2M Architecture with Node and Topology Abstractions. pp. 89–109.

[26] Herstad, A., Nersveen, E., Samset, H., Storsveen, A., Svaet, S., and Husa, K. Connected objects: Building a service platform for M2M. pp. 1 –4.

[27] INFSO, D., and EPoSS. Internet of Things in 2020: A roadmap for the future. Report from the workshop: Beyond rfid - the internet of things, September 2008.

[28] Jan Audestad, Inge Grønbæk, Stein Svaet. Connected Objects Platform Specification, Version 1. R & I Research Report, Telenor R & I, February 2009.

[29] Karim, J., Bin Wan Amat, W., and Razak, A. Car ignition system via mobile phone. In *Future Computer and Communication, 2009. ICFCC 2009. International Conference on* (3-5 2009), pp. 474 –476.

[30] King, P. Digital Media Devices Global Market Report. Research report, Stratergy Analytics, July 2008.

[31] Krco, S., Tsiatsis, V., Matusikova, K., Johansson, M., Cubic, I., and Glitho, R. Mobile network supported wireless sensor network services. In *Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE Internatonal Conference on* (8-11 2007), pp. 1 –3.

[32] Laguna, M. A., Finat, J., and González, J. A. Mobile health monitoring and smart sensors: a product line approach. In *EATIS '09: Proceedings of the 2009 Euro American Conference on Telematics and Information Systems* (New York, NY, USA, 2009), ACM, pp. 1–8.

[33] Martsola, M., Kiravuo, T., and Lindqvist, J. Machine to machine communication in cellular networks. In *Mobile Technology, Applications and Systems, 2005 2nd International Conference on* (15-17 2005), pp. 6 pp. –6.

[34] Richard Harrison and Mark Shackman. *Symbian OS C++ for Mobile Phones.* John Wiley and Sons, 2007.

[35] Ryberg, T. The European Wireless M2M Market. Research report, M2M Research Series, May 2007.

[36] Siwen, L., and Yunhong, L. Design and implementation of home automation system. In *Information Science and Engieering, 2008. ISISE '08. International Symposium on* (20-22 2008), vol. 2, pp. 633 –636.

[37] Song, J. Y., and Kim, D. H. u-manufacturing model: application system using rfid/usn, mobile and internet technology. In *Advanced Communication Technology, 2008. ICACT 2008. 10th International Conference on* (17-20 2008), vol. 1, pp. 79 –83.

[38] Sverre Bye Grimsmo. Reliability issues when providing M2M Services in the Internet of Things. Master's thesis, Norwegian University of Science and Technology, NTNU, 2009.

[39] Telenor R & I. Connected Objects WIKI. WWW page of the Connected Objects WIKI: http://telenorobjects.onjira.com/wiki/display/coos/Home. [Accessed 15th March 2010.].

[40] Union, I. T. The Internet of Things. Executive summary, ITU.

[41] Wang, W., Srinivasan, V., and Chua, K.-C. Extending the lifetime of wireless sensor networks through mobile relays. *Networking, IEEE/ACM Transactions on 16*, 5 (oct. 2008), 1108 –1120.

Appendix A

Source Codes and Sample Output

A.1 Source Codes

In this section, we include the important portions of source code of one of the most important classes of our implementation: *DefaultMessage()*.

A.1.1 DefaultMessage()

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Collections;
using System.Net;
using System.Net.Sockets;

namespace test
{
  public class DefaultMessage : Message
  {
   String TRACE_ROUTE = "traceRoute";
   String TRACE = "trace";
   String PRIORITY = "priority";
   String QOS_CLASS = "QoS";
   String MESSAGE_NAME = "name";
   String SECURITYTOKEN = "sectoken";
   String EXCHANGE_PREFIX = "coosx";
   String EXCHANGE_ID = "xId";
   String EXCHANGE_PATTERN = "xpattern";
   String DEFAULT_MESSAGE_NAME = "";
   String TIME_STAMP = "ts";
   String GUARANTEED_DELIVERY = "gd";
   String TRANSACTION_ID = "tId";
   String SENDER_ENDPOINT_NAME = "senderEPName";
   String RECEIVER_ENDPOINT_NAME = "receiverEPName";
   // message type header parameter
   String TYPE = "type";
   // message type values
   String TYPE_MSG = "msg";
   String TYPE_ERROR = "error";
   String TYPE_ANALYZE = "analyze";
   String TYPE_ROUTING_INFO = "routingInfo";
   String TYPE_ALIAS = "alias";
   String ERROR_REASON = "errorReason";
   String ERROR_NO_ROUTE = "noRoute";
   String ERROR_NO_ALIAS = "noAlias";
   String ERROR_TOO_MANY_HOPS = "tooManyHops";
   String ERROR_NO_RECEIVER = "noReciver";
   // message hops field
   String HOPS = "hops";
   // message segment field
   String SEGMENT = "seg";
   // Message content type header parameter
    String CONTENT_TYPE = "contentType";
   // Property content type
   String CONTENT_TYPE_PROPERTY = "property";
   // String content type
   String CONTENT_TYPE_STRING = "string";
   // Byte array content type
   String CONTENT_TYPE_BYTES = "bytes";
   // Object content type
   String CONTENT_TYPE_OBJECT = "object";
   // body serialization method header parameter
```

```
String SERIALIZATION_METHOD = "ser";
// serialization method ActorFrame, not dependant on java SE but
// serialization must be implemented
String SERIALIZATION_METHOD_AF = "af";
// serialization method Java, dependant on Java SE
String SERIALIZATION_METHOD_JAVA = "java";
// serialization method default
String SERIALIZATION_METHOD_DEFAULT = "def";
/// </summary>
protected String receiverEndpointUri;
protected String senderEndpointUri;
protected Hashtable headers = new Hashtable();
protected Object body;
protected byte[] serializedbody;
public static TcpClient client;
public static MemoryStream bout;
public static BinaryWriter w;

public static StringBuilder reply = new StringBuilder();
  Interface inter = new Interface();
  inter.ShowDialog();

 }

 public static void sendConnectMessage()
  {
   dout.Write((bool)false);
   MemoryStream bout = new MemoryStream();
   BinaryWriter dout = new BinaryWriter(bout);
   dout.Write((byte)0);
   dout.Write((bool)false);
   dout.Write((bool)false);
   dout.Write((int)IPAddress.HostToNetworkOrder(3));
   DefaultMessage.WriteUTF(dout, "type");
   DefaultMessage.WriteUTF(dout, "msg");
   DefaultMessage.WriteUTF(dout, "name");
   DefaultMessage.WriteUTF(dout, "connect");
   DefaultMessage.WriteUTF(dout, "con_uuid");
   DefaultMessage.WriteUTF(dout, ".UUID-1234");
   dout.Write((int)IPAddress.HostToNetworkOrder(0));
   MemoryStream bouth = new MemoryStream();
```

```
        BinaryWriter douth = new BinaryWriter(bouth);
        byte[] payload = bout.ToArray();
        int length = payload.Length;
        douth.Write((int)(IPAddress.HostToNetworkOrder(length)));
        douth.Write(payload);
        byte[] serial = bouth.ToArray();
        douth.Flush();
        sendAndReceive(serial);
        }
      public static void sendMessage(string mess)
        {
        MemoryStream bout = new MemoryStream();
        BinaryWriter dout = new BinaryWriter(bout);
        dout.Write((byte)0);
        dout.Write((bool)true);
        DefaultMessage.WriteUTF(dout, "coos://myownartifact");
        dout.Write((bool)true);
        DefaultMessage.WriteUTF(dout, "coos://Ping");
        dout.Write((int)IPAddress.HostToNetworkOrder(6));
        DefaultMessage.WriteUTF(dout, "type");
        DefaultMessage.WriteUTF(dout, "msg");
        DefaultMessage.WriteUTF(dout, "name");
        DefaultMessage.WriteUTF(dout, "test");
        DefaultMessage.WriteUTF(dout, "xId");
        DefaultMessage.WriteUTF(dout, "1234");
        DefaultMessage.WriteUTF(dout, "xpattern");
        DefaultMessage.WriteUTF(dout, "OutIn");
        DefaultMessage.WriteUTF(dout, "contentType");
        DefaultMessage.WriteUTF(dout, "string");
        DefaultMessage.WriteUTF(dout, "ser");
        DefaultMessage.WriteUTF(dout, "def");
        int length = mess.Length;
        dout.Write((Int16)IPAddress.HostToNetworkOrder(length));
        DefaultMessage.WriteUTF(dout,mess);
        MemoryStream bouth = new MemoryStream();
        BinaryWriter douth = new BinaryWriter(bouth);
        byte[] payload = bout.ToArray();
        int length = payload.Length;
        douth.Write((int)(IPAddress.HostToNetworkOrder(length)));
        douth.Write(payload);
        byte[] serial = bouth.ToArray();
        douth.Flush();
        sendAndReceive(serial);
        }
      public static void sendByte(byte b)
       {
        MemoryStream bout = new MemoryStream();
        BinaryWriter dout = new BinaryWriter(bout);
        dout.Write((byte)0);
        dout.Write((bool)true);
        DefaultMessage.WriteUTF(dout, "coos://myownartifact");
        dout.Write((bool)true);
        DefaultMessage.WriteUTF(dout, "coos://Ping");
        dout.Write((int)IPAddress.HostToNetworkOrder(5));
        DefaultMessage.WriteUTF(dout, "type");
        DefaultMessage.WriteUTF(dout, "msg");
        DefaultMessage.WriteUTF(dout, "name");
        DefaultMessage.WriteUTF(dout, "Ping?");
        DefaultMessage.WriteUTF(dout, "xId");
        DefaultMessage.WriteUTF(dout, "1234");
        DefaultMessage.WriteUTF(dout, "xPattern");
        DefaultMessage.WriteUTF(dout, "OutOnly");
        DefaultMessage.WriteUTF(dout, "contentType");
        DefaultMessage.WriteUTF(dout, "byte");
        dout.Write((int)IPAddress.HostToNetworkOrder(1));
        dout.Write((byte)b);
        MemoryStream bouth = new MemoryStream();
        BinaryWriter douth = new BinaryWriter(bouth);
        byte[] payload = bout.ToArray();
        int length = payload.Length;
        douth.Write((int)(IPAddress.HostToNetworkOrder(length)));
        douth.Write(payload);
        byte[] serial = bouth.ToArray();
        douth.Flush();
        sendAndReceive(serial);
        }
      static void sendAndReceive(byte[] serial)
       {
        try
        {
        Console.WriteLine("Connected to the Server ....");
        c.Write(serial, 0, serial.Length);
        c.Flush();
        Console.WriteLine("Data sent to the Server .....");
        try
        {
         byte[] serial1 = new byte[150];
         DefaultMessage msg1 = new DefaultMessage(reader);
        }
        catch (EndOfStreamException e)
        {
         Console.WriteLine(e.Message);
        }
       }
       catch (SocketException e)
        {
        Console.WriteLine(e.Message);
        }
       }
     public DefaultMessage()
     {
      setHeader(MESSAGE_NAME, DEFAULT_MESSAGE_NAME);
      setHeader(TYPE, TYPE_MSG);
     }
     public DefaultMessage(String signalName)
     {
      setHeader(MESSAGE_NAME, signalName);
      setHeader(TYPE, TYPE_MSG);
     }
     public DefaultMessage(String signalName, String type)
     {
      setHeader(MESSAGE_NAME, signalName);
      setHeader(TYPE, type);
     }
     public DefaultMessage(BinaryReader din)
     {
      deserialize(din);
     }
     public String getReceiverEndpointUri()
      {
      return receiverEndpointUri;
      }
```

```
public Message setReceiverEndpointUri(String receiverEndpointUri
{
 this.receiverEndpointUri = receiverEndpointUri;
 return this;
}

....
....
....

public void deserialize(BinaryReader dinR)
{
 int k=0;
 Console.WriteLine("Deserialization Starting......");
 NetworkStream ms = (NetworkStream)dinR.BaseStream;
 int j = IPAddress.HostToNetworkOrder(dinR.ReadInt32());
 reply.Append(" Message Size ");
 reply.Append(j);
 reply.Append(System.Environment.NewLine);
 byte version = dinR.ReadByte();
 reply.Append(" Version ");
 reply.Append(version);
 reply.Append(System.Environment.NewLine);
 Console.WriteLine(" Payload Size "+ j);
 Console.WriteLine(" Version " + version);
 if (dinR.ReadBoolean())
 {
  dinR.ReadByte();
  receiverEndpointUri = dinR.ReadString();
  reply.Append(" Receiver EndPoint URI ");
  reply.Append(receiverEndpointUri);
  reply.Append(System.Environment.NewLine);
  }
 if (dinR.ReadBoolean())
 {
  dinR.ReadByte();
  senderEndpointUri = dinR.ReadString();
  reply.Append(" Sender EndPoint URI ");
  reply.Append(senderEndpointUri);
  reply.Append(System.Environment.NewLine);
 }
```

```
    int headerSize = IPAddress.HostToNetworkOrder( dinR.ReadInt32()
    Console.WriteLine(" Header Size "+ headerSize);
    reply.Append(" Header Size ");
    reply.Append(headerSize);
    reply.Append(System.Environment.NewLine);
    for (int i = 0; i < headerSize; i++)
    {
     dinR.ReadByte();
     String key = dinR.ReadString();
     Console.WriteLine(" Key " + key);
     reply.Append(" Key ");
     reply.Append(key);
     dinR.ReadByte();
     String value = dinR.ReadString();
     Console.WriteLine(" Value "+value);
     reply.Append(" Value ");
     reply.Append(value);
     if(!(headers.Contains(key)))
     headers.Add(key, value);
     reply.Append(System.Environment.NewLine);
    }
    serializedbody = new byte[IPAddress.HostToNetworkOrder(dinR.Rea
    Console.WriteLine("Message Body Length "+ serializedbody.Length
    reply.Append(" Message Body Length ");
    reply.Append(serializedbody.Length);
    reply.Append(System.Environment.NewLine);
    if (serializedbody.Length == 0)
    {
      return;
    }
    dinR.Read (serializedbody, 0, serializedbody.Length);
    }

   private void deserializeBody()
   {
   if (body == null && serializedbody != null && serializedbody.Len
   {
    String serMethod = headers[SERIALIZATION_METHOD].ToString();
    if (serMethod != null)
    {
     Serializer serializer = SerializerFactory.getSerializer(serMet
     body = serializer.deserialize(serializedbody);
    }
    else
    {
     throw new Exception("No serialization method indicated in mess
    }
   }
  }
  public byte[] serialize()
  {
   MemoryStream bout = new MemoryStream();
   BinaryWriter dout = new BinaryWriter(bout);
   dout.Write((Byte)1);
   // The addresses
   dout.Write((Boolean)(receiverEndpointUri != null));
   if (receiverEndpointUri != null)
   {
    WriteUTF(dout, receiverEndpointUri);
   }
   dout.Write((Boolean)(senderEndpointUri != null));
   if (senderEndpointUri != null)
   {
   WriteUTF(dout, senderEndpointUri);
   }
  // The body
   if (body != null && serializedbody == null)
   {
   //headers.Add(CONTENT_TYPE, CONTENT_TYPE_STRING);
   String serMethod = "def";
   if (serMethod != null)
   {
    Serializer serializer = SerializerFactory.getSerializer(serMeth
    if (serializer != null)
    {
     serializedbody = serializer.serialize(body);
    }
    else
    {
     throw new Exception("Serialization method not registered: " +
    }
   }
   else
   {
    try
    {
     Serializer serializer = SerializerFactory.getDefaultSerializer
     serializedbody = serializer.serialize(body);
     headers.Add(SERIALIZATION_METHOD, SERIALIZATION_METHOD_DEFAULT
    }
    catch (Exception e)
    {
     Serializer serializer = SerializerFactory.getSerializer(SERIALI
     if (serializer != null)
     {
     serializedbody = serializer.serialize(body);
     headers.Add(SERIALIZATION_METHOD, SERIALIZATION_METHOD_JAVA);
     }
    else
    {
     throw new Exception("Serialization failed");
    }
   }
  }
  // The headers
  dout.Write((int)headers.Count);
  dout.Flush();
  IDictionaryEnumerator en = headers.GetEnumerator();
  while (en.MoveNext())
  {
   dout.Write((String)en.Key);
   dout.Write((String)en.Value);
  }
   dout.Flush();
   // The body
   if (serializedbody != null)
   {
    dout.Write((int)serializedbody.Length);
    dout.Write(serializedbody);
   }
   else
   {
    dout.Write((int)0);
    }
    String temp = dout.ToString();
    Console.WriteLine(temp);
    return bout.ToArray();
  }
    ....
    ....
    ....
}
```

A.2 Sending and Receiving a String

Input : Hello COOS

Output at Client Side:

```
Message Size 137
Version 0
Header Size 4
Key name Value connectAck
Key type Value msg
Key router_uuid Value .UUID-R-Router1-1-1276723963015-0-0
Key con_uuid Value .UUID-R-Router1-1-1276723963015-0-0
Message Body Length 0
Message Size 209
Version 0
Receiver EndPoint URI coos://Ping
Sender EndPoint URI coos://.UUID-localhost-1-1276723963015-3-0
Header Size 7
Key hops Value 1
Key receiverEPName Value Ping
Key name Value test Received With Body Hello COOS
Key xId Value 1234
Key type Value msg
Key xpattern Value InOut
Key senderEPName Value myownartifact
Message Body Length 0
```

Output at Server Side

```
[Thread−4] INFO org.coos.messaging.transport.TCPTransport −
 Reader started on :/ 127.0.0.1:15656

[Thread−4] INFO org.coos.messaging.transport.TCPTransport −
Writer started on :/ 127.0.0.1:15656

[Thread−7] DEBUG org.coos.messaging.transport.DefaultChannel
Server − Allocatingnew channel!

[Thread−7] DEBUG org.coos.messaging.routing.DefaultRouter −
coos1: Adding link:Link to org.coos.messaging.transport
.TCPTransport@1a12495, destUUID: .UUID−1234, aliases:[]

[Thread−7] INFO org.coos.messaging.processor.Logger −
DefaulLoggerName, ReceiverEndpoint: coos://myownartifact,
SenderEndpoint: coos://Ping, Message name: test ,Message
type: msg, isOutLink: false

[pool−1−thread−1] DEBUG myowngroup.impl.myownartifact
Endpoint −Endpoint: coos://myownartifact, Processing
incoming exchange: ExchangeID: 1234

[pool−1−thread−1] DEBUG myowngroup.impl.myownartifact
Endpoint −Endpoint: coos://myownartifact,
Processing outgoing exchange:ExchangeID: 1234

[pool−1−thread−1] INFO org.coos.messaging.processor.Logger −
DefaulLoggerName, ReceiverEndpoint: coos://Ping,
SenderEndpoint: coos://.UUID−localhost−1−1276723963015−3−0,
Message name: test Received With Body Hello COOS,
Message type: msg, isOutLink: true

[Thread−7] INFO org.coos.messaging.transport.TCPTransport −
Connection closing EOF

[Thread−7] INFO org.coos.messaging.transport.TCPTransport −
Closing transport: null:15656
Received msg: test from: coos://Ping with body: Hello COOS
```