Managing HMI utilities for control systems

by
Ekaterina Soukhikh

Master's Thesis in
Information and Communication Technology

Agder University College
Faculty of Engineering and Science

Grimstad
Norway

31 May 2007

Abstract

The supervisor of this project, Origo Engineering AS delivers complete control and automation systems for the oil industry and smelting plants. Each hardware and software package is custom-made. A part of the functionality is implemented by scripts – small programs that are coded and run directly on a Human Machine Interface (HMI) server. This method has some challenges, which are listed and further described in this report. Those challenges increase the time and cost of producing the control and automation system.

The purpose of this master thesis is to design a solution that will solve those challenges. The solution system is called Utility Framework Server. The intentions and design of a Utility Framework Server, an application for managing utilities for control systems that can replace the scripts that are placed on the HMI server, and an implementation of a prototype for it, are further described in the report.

The research project has investigated possible technologies for the solution, and how they can participate in improving the current challenges. Service orientation and component-oriented programming were chosen as the methods for the solution. The reasons for this choice are also presented in the report.

The architecture has been defined based on analysis of the specifications for the system. The prototype for the defined architecture has been implemented using Windows Communication Foundation and .NET. Some clients that can use the services presented by the server have also been implemented for testing the prototype.

Preface

This thesis was written in cooperation with the company Origo in Kristiansand, as a part of the Master degree in information and communication technology at Agder University College. The work was carried out in the period between January 2007 and May 2007.

I would like to thank my supervisors, Andreas Prinz at Agder University College and Trond Friisø from Origo, for valuable help during the entire process of this thesis. I would also like to thank Sven Harald Nilsen for help with the practical part of the project, and Pål Espen Nilsen, Jan Pettersen Nytun and Morten Goodwin Olsen for helpful feedback on this report.

Supervisors:
Trond Friisø
Andreas Prinz

Contact person for Origo:
Sven Harald Nilsen

Author:

____
Ekaterina Soukhikh

Table of contents

Figure List

Table List

Example List

1 Introduction

The external supervisor of this project is Origo Engineering AS. Origo specializes in delivering hardware and software solutions for the oil industry and melting plants. Each delivered control and automation system is customized for the purchaser. The customizing process can be time-consuming. The proposal of this master thesis came from Sven Harald Nilsen. He is one of those who are currently working with programming and software development at Origo. Sven Harald Nilsen came up with the idea to improve the efficiency of customizing HMI systems.

1.1 Problem statement

Control and automation systems consist of a hierarchy where the lowest level consists of sensors and actuators. These communicate with controllers, which are often Programmable Logical Controls (PLCs). At the next level (top level) is the Human Machine Interface (HMI) system. It communicates with the controllers, processes data, and presents data to the user (operator).

The communication between the HMI and the controllers is both for collecting data and for commands from the operator, for example, open valves etc. The HMI system has a server - client architecture. For each customer the HMI system has to be tailor-made. Today a lot of the functionality for processing and presentation of data is made with the help of a number of scripts in the HMI systems.

This works fine, but there are three challenges that occur:

1. The scripts are difficult to reuse in different deliveries, resulting in unnecessary resources being used on SW development. It is also difficult to present the scripts as products, which should be feasible.
2. The scripts are dependent on the HMI system. This is a drawback if the customer demands a particular HMI system, or if replacing the HMI system would be necessary in the future.
3. Due to insufficient error handling, errors in the scripts might cause the entire HMI system to crash.

Origo Engineering AS is interested in a solution that could answer these three challenges. The information problem is then specified as “Develop a solution that makes it easy and stable to include and handle different utility programs on the HMI (Human Machine Interface) server system”.

1.2 Work description

The first step of the project is to analyze the challenges with customizing the HMI systems. The way to improve this process by eliminating the challenges has to be found. This part of the work was divided into 3 tasks:

- Investigate methods for improving reusability of HMI utilities
- Look at possibilities for reducing dependency between HMI utilities and the HMI system
- Consider methods for error handling

This research was done in cooperation with Origo and became the foundation for setting up the requirements for the system that could improve the existing situation.

The next step was to review modern software technologies that claim to solve previously mentioned or similar problems. Service orientation (SO) and component-oriented programming (COP) were chosen as the solution methods.

The results in the reviews were used to design “an ideal solution” – a system that implements all the requirements and therefore resolves the current challenges.

The ideal solution, the inputs from Origo and the time scale of the project were then used to decide what kind of prototype that was going to be developed. The purpose of creating the prototype was to prove the concepts presented in the ideal solution. The programming framework Microsoft .NET 3.0 and its new tool for developing distributed systems that follow SO principles - Windows Communication Foundation (WCF) were chosen as the tools for implementation. How they enforce the principles of SO and COP was investigated, and therefore also how they contribute to solving the current challenges.

1.3 Importance of the project

Origo Engineering AS is eager to get good and functional results from this cooperation and is willing to help during research and development. Since they are truly interested in finding an efficient and practical solution to the particular problems they are experiencing today, cooperation with external supervisor seems to be valuable for both parts. Since the projects task is to develop a prototype for a bigger solution, it has to be well documented, so that Origo could use the achieved results for further development and improvement. Using principles of component oriented programming and programming tools that are chosen by the supervisor will also contribute to making the results from this project easier to utilize in future development. A new framework solution is predicted to increase the efficiency, flexibility and robustness of the existing system.

1.4 Delimitations

The task is not to create a fully functional Utility Framework for Origo Engineering AS. It is rather to investigate the possibilities and potentials in existing technologies, and to make a proposal for a solution for one. A prototype will be created to, in practice, demonstrate the basic principles of the solution idea and how it will improve the efficiency of the existing software.

1.5 Motivation

There are several reasons that became the motivation for choosing this particular project. Among these are:

- This is the research project where a practical output beside the theoretical researches is expected, what appeals to the author of this thesis.
- Server-client based application programming by using SOA principles is a quickly developing field that has become an increasingly popular research area during the latest years.
- Working with .NET and C# appears interesting and challenging. The framework and programming language are popular in today's software development, and are useful to learn.
- Software programming for HMI systems is an interesting field for development.

1.6 Report outline

Report outline section describes an outline of the report. Below is a short summary of the contents of each chapter.

Introduction chapter briefly introduces the project and describes the problem assignment, challenges that are going to be solved in the project, work description and motivation. Chapter two contains some general information about research topic. Chapter three "Problem research and state of the art" elaborates on the challenges that have caused the task of this project and describes the choice of methods and tools that were used to solve them.

Chapter four "Design of the Utility Framework Server" describes the Utility Framework Server as an application that is going to solve the challenges with customizing the HMIs. Chapter five "Prototype of the Utility Framework Server" describes the implementation of a prototype. In the sixth chapter is the discussion of the findings from the previous chapter. Chapter seven contains a summary of the project conclusions and suggestions for further work.

2 Background

The background chapter is divided into two parts. The first part describes the technologies and programming techniques that are general for software development. The other part concerns the theory and definitions that are special for designing software products for the industry.

2.1 General software

The technologies used in this project are Microsoft .NET Framework and the C# programming language. The .NET Framework was recently released in a new version, 3.0. This version is going to be investigated in order to find out if it has something particularly useful for this projects task. After researching and reading on this subject on the internet, it was decided to try using Windows Communication Foundation (WCF) for server-client connections. This new technology seems promising and appealing. On the other hand, this choice has caused some challenges during the work, as it was difficult to find proper documentation about WCF, due to the fact that it is a relatively new technology.

C# and .NET were the tools that Origo desired to be used in project. C# was chosen for the development because it is one of the languages used by Origo for creating applications for the Windows platform. .NET is the natural environment for C# programming.

2.1.1 C# programming language

C# was created in the late nineties and the first complete version was released in the middle of 2000 as a part of the .NET Framework. C# is a relatively new language and is often being compared to Java, one of the other often used object-oriented programming languages. They do have some similarities, for example, they are both modern languages derived from C and C++. They both have garbage collection (GC). GC is an automatic memory management. The garbage collector attempts to “remove garbage”, or to free memory that is used by objects that will never again be accessed or changed by the application. This list can be continued, and with that many similarities, the need for C# as a stand-alone programming language can be questioned. So what was the idea behind its creation? To be platform independent was one of the main design goals back creating the Java language; and its developers succeeded in it. The independency was achieved by using the Java Runtime Machine for running the applications, implemented in java. Even if it successfully addressed many issues surrounding portability, it also caused some shortcomings in Java. One of them is cross-language interoperability, also called mixed-language programming. It is, shortly explained, the ability for the code produced in one language to work easily with the code produced in another language.

Cross-language interoperability is often needed for producing large distributed software systems. It is no secret that different programming languages are best suited for different tasks, and at the same time, programmers' preferences in languages also vary. Service oriented architecture, which is becoming popular in modern software development and will get its attention later in this project, implies allowing programmers to use the language of their choice for implementations. Component-oriented programming insists on reusability of once created components. If one component can be reused on a wider variety of platforms and languages, it makes it more valuable. In the light of those new tendencies, Java has some limitations [10].

Another missing element in Java is full integration with the Windows platform. Java programs can be executed in a Windows environment (where java virtual machine has been installed), but Java and Windows are not closely coupled.

In order to fulfill these needs, Microsoft developed C#. The C# language was not created to be better than Java and was not meant to replace Java. The two languages were both designed carefully, but with different purposes in mind. If one language has a feature another lacks, it is the result of design decisions that were made intentionally in a development process.

C# was specially developed for use in the .NET Framework and it depends on libraries that the framework defines. Even though C# can be separated from the .NET Framework, they are tightly bound together. Because of this, it is important to have a general understanding of .NET and why it is significant to C# [11].

2.1.2 Microsoft .NET

The .NET Framework is a programming environment that supports development and execution of distributed component-based applications. It allows different languages to work together in creating parts for one application. The framework's class library simplifies programming development in, for example, such fields as user interface, data access, web application development, network communications and others. .NET also provides security, portability and a common programming model for the Windows platform.

The two parts of the .NET Framework are most important: Common Language Runtime (CLR) and .NET Framework Class Library. CLR, as a running tool, is responsible for executing applications written in programming languages that .NET supports (for example, C++, C#, J#). A special ability of CLR is that it also provides portability, allowing mixed-language programming (executes applications where components / parts are written in different languages) and provides security.

The .NET Framework Class Library provides a program an access to a runtime environment. A class library is a collection of already constructed classes and methods for performing different logic operations, as for example for I/O operations (as showing information on and getting it from a console window). The .NET 2.0 Framework's Class Library is rich and has several built-in functions for different operations.

A limitation is that the framework concentrates on developing applications for Windows based platforms. Portability is also conditional. As Java virtual machine needs to be installed on a PC for running Java-applications, an application written in for example C# will automatically use the .NET Framework class library, will be portable to all .NET environments.

2.1.3 .NET Framework 3.0

.NET 3.0 from Microsoft is a new version of the .NET Framework. The .NET 3.0 Framework came out in alpha-version in November 2007 and has four new parts in addition to the .NET 2.0 Framework. The beta-version of these four parts was earlier presented as "WinFX". The 3.0 Framework is included in Microsoft's newest addition to the Windows family – Windows Vista. It will also be included in the next server solution from Windows, which is now known by the codename Longhorn. Figure 1 presents the whole .NET 3.0 stack as it exists in Windows Vista [13].

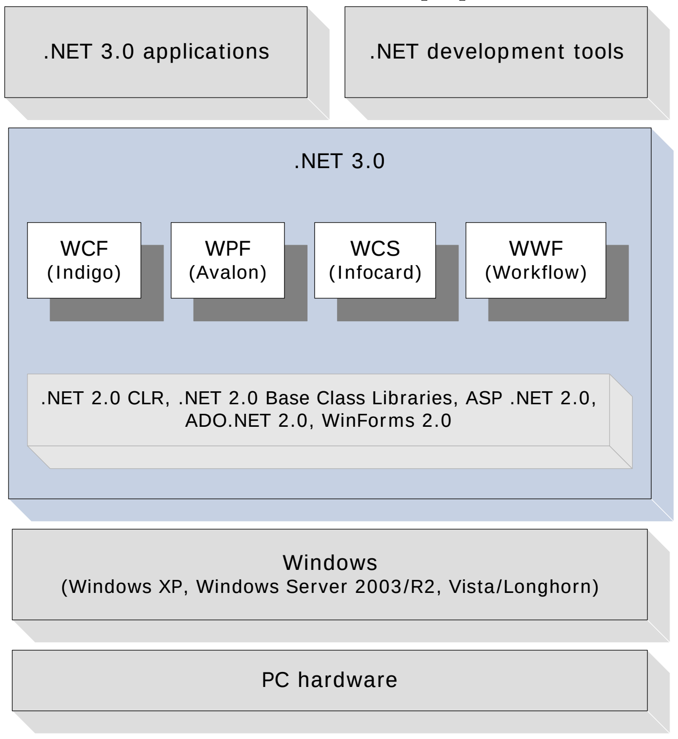

.NET 3.0 applications
.NET development tools
.NET 3.0
WCF (Indigo)
WPF (Avalon)
WCS (Infocard)
WWF (Workflow)
.NET 2.0 CLR, .NET 2.0 Base Class Libraries, ASP .NET 2.0, ADO.NET 2.0, WinForms 2.0
Windows (Windows XP, Windows Server 2003/R2, Vista/Longhorn)
PC hardware

Figure 1 - .NET 3.0 Framework Stack

The four new parts are:

- Windows Presentation Foundation

WPF is formerly known by the codename Avalon, it is a tool for creating a user interface (UI) that is based on .NET, XML and vector graphic technologies, and uses 3D hardware acceleration.

- Windows Workflow Foundation

(WWF) allows for the building of task automation and integrated transactions using workflows.

- Windows CardSpace

(WCS), formerly code-named InfoCard; a software component for securely storing a person's digital identity and providing a unified interface for choosing the identity for a particular transaction, such as logging in to a website.

- Windows Communication Foundation

Formerly known by the code-name Indigo, WCF is a service-oriented messaging system that allows programs to interoperate locally or remotely similar to web services.

A number of previous technologies from Microsoft can be used for implementing of distributed systems, as ASP.NET web services, Web Services Enhancements, Microsoft Message Queuing, Enterprise Services/COM+ and .NET Remoting. Those and a number of new technologies are now gathered in WCF. This way, WCF gives developers one programming model where all communication is collected.

2.1.4 Introduction of Windows Communication Foundation

Windows Communication Foundation (WCF) is Microsoft's new programming model for communication inside one and between a numbers of systems. WCF is one of the 4 parts in the new 3.0 .NET Framework.

The WCF architecture of a distributed application consists of 4 parts (3 without the client application that can use the services):

- Service

The service is a part of a business functionality or application behavior that you wish to provide to the external world.

- Host

A host environment is an application domain and process in which the service runs.

- Endpoint

An endpoint is a window through which a WCF service would communicate with the outside world. A WCF service should expose at least one end point or more.

- Client

A client is any unit that utilizes the service as part of its operations. A client typically uses a channel created by WCF to interact with the service [20].

The endpoint itself consists of 3 elements:

- Address
- Binding
- Contract

Address describes where the service can be found, binding describes communication method and the contract element describes available functionality. Those three are called the ABC's of WCF. Figure 2 illustrates how the connection between client (host) and service is happening. All three elements are described later in the report.

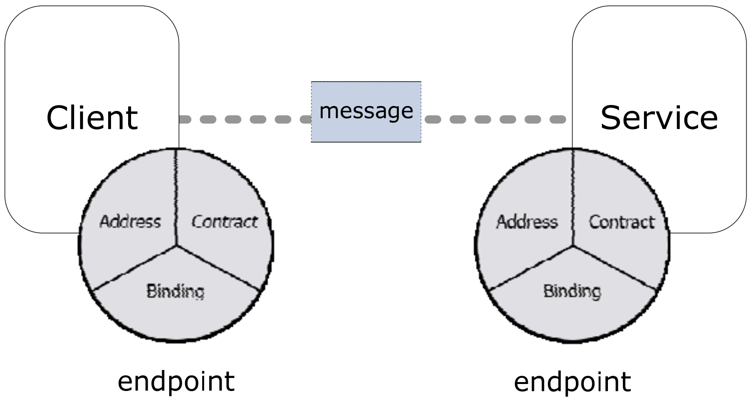


Figure 2 - WCF's ABC

2.1.4.1 Address

Every service has a unique address. The base address for a service has to be defined in the configuration file App.config as is showed in Example 1.

```
<host>
      <baseAddresses>
            <add baseAddress="net.tcp://localhost:9000"/>
            <add baseAddress="http://localhost:8000"/>
      </baseAddresses>
</host>
```

Example 1 - Base address

This unique address consists of a service address : an IP address, the servers network name or "localhost" if clients and services are on the same machine, a port number and definition of a transport protocol. WCF now supports HTTP, TCP, P2P, IPC and MSMQ protocols. The base address always use the same format. Example 2 presents this format.

```
[transport]://[machine or domain][:optional port]
```

Example 2 - Base address format

In the Example 1 the service has two endpoints: one for HTTP and one for TCP connection. As it was mentioned before, multiple endpoints of one service is allowed, and it can be usefull to implement a host in a HTTP mode at the beginning (more about it in the Appendix 1). More information about the transport protocols can be found in the Bindings paragraph later in the report.

2.1.4.2 Binding

A newly created service has multiple possibilities related to communication patterns. For example, messages can be synchronous (server waits for reply) and asynchronous (messages thrown and forgot), queued or not, long-lasting or unstable. Different protocols can also be used for connections, such as HTTP, TCP, P2P, IPC (named pipes) or MSMQ. Note that P2P and IPC are strictly not protocols, but are systems of communication techniques, each one created for own purposes. A peer-to-peer (or P2P) is a computer network especially set up for file sharing. Inter-Process Communication (or IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. Anyhow this difference is not critical for this project, and they will be further referred as “protocols”.

Securing messages in WCF is optional. To simplify the process of choosing between many different communication alternatives, WCF groups a set of such communication choices into bindings [20].

A binding on WCF is a consistent set of choices within communication: transport protocol, message encoding, communication pattern, reliability, security and so on. The main goal of binding is to extract all these communication aspects from the application’s main programming logic, and place them in the communication configuration component.

WCF offers 9 standard bindings and they are all further described in Appendix 2. Two of them that are relevant for the prototype implementation are:

- Basic Binding

Implemented by BasicHttpBinding class, this is the binding for a classic web service; it inherits principles from earlier ASP’s web services (ASMX). Using this binding allows old clients to use new web services.

- TCP binding

Offered by NetTcpBinding, it allows machines to communicate over the internet by using the TCP protocol. It is possible to configure the communication that uses TCP binding in WCF to support optional reliability, transactions and security. This type of binding implies that both clients and services have to use WCF.

When designing the Utility Server solution, TCP binding was chosen as the main binding. In practice, the service and clients will be placed on the same network, and they can reach each other thought it. Exposing services to internet is therefore unnecessary. HTTP binding was created as another choice, just in case (for example, for easier checking if the services are up and running). As both the service and clients are new (there are no services or clients already existing), and the flexibility of communication is important, TCP binding looked like the best choice for the task. This type of binding allows configuring its reliability to a higher degree, and it can definitely be practical in the future work with the Utility Server Framework. There is more about binding's reliability in Appendix 2.

2.1.4.3 Contracts

Contracts in WCF define how a client can use a service. There are 3 types of services in WCF: Service, Data and Message. The service contract describes what operations the service can perform. Data contracts illustrate data structures that can be used for these operations. The message contract can be used for managing the structure of SOAP messages that the service and client will exchange.

Service contracts describe what a service can do out of the list of service operations. A service operation is defined by an operation contract. An operation contract is similar to a method. Just like methods, they can receive parameters and return values. Example 3 illustrates contract descriptions in WCF.

```
[ServiceContract(CallbackContract=typeof(IMessagingServiceCallback))]
    public interface IMessagingService
    {
        [OperationContract]
        string SendMessage(string message);
        [OperationContract]
        int RegisterClient(IsOneWay = true);
    ...
    }
```

Example 3 - Contracts

Even thought WCF is a new concept, it relies and resembles in some degree of the other models of the distributed systems, which are common in software development. On the other hand, definitions, standards and software technologies that are used in industry or particularly in creating the control and automation systems are more special and peculiar. The next section presents some background theory in this area.

2.2 Software in control and automation systems

The development of information technology has caused modifications in automation and control systems. During the last few years, they have changed from centralization to distribution, from closure to openness. The appearance of the networks and low-cost but powerful personal computers caused transformations in a world of industry. Personal computers are now widely used for visualization, data access, process control and several other automation and control solutions. This development has brought along some new definitions, technologies and standards, and some of these will be discussed in this chapter.

The external supervisor of this project, Origo Engineering AS, works on delivering a complete equipment system for control and surveillance of processes for the oil industry and smelting plants. As a hardware/software supplier, they also want to contribute by putting integrated operations into practice and thus improving the safety of the offshore production on the Norwegian continental shelf. Origo is already working on implementing secure and reliable Information and Communication Technology (ICT) systems; and they are ready to make some changes, taking new standards for integrated operations into consideration. This project assignment isn't directly connected to integrated operations as this part is given by the supervisor. It concerns the improvement of existing solutions in order to achieve better reusability and faultlessness. The Integrated Operations (IO) is then one of the motivations for this project.

2.2.1 Integrated operations

The oil and gas industry depends on reliability of ICT systems no less than the rest of the modern society. Also, hardware and software computer technologies developed dramatically during the last few years. These two reasons contribute to an ever growing popularity of the subject of integrated operations. Norwegian oil companies look at IO as one of the first priorities in a strategy for future development.

Integrated operations are expected to contribute to increasing exploitation and production, in reducing expenses and in improving the safety in the offshore industry. White paper nr 38 [1] from the Norwegian Parliament defines integrated operations as "Use of information technologies for achieving better decisions, remote control of equipment and procedures, and to move the functions and personnel onshore". It is a rather general request for oil companies to invest in an improvement of existing computer technologies and in relocating its personnel onshore. The result is that the amount of money spent for those purposes is increasing and the anticipations about IO are big.

2.2.2 Human machine interface

When the personal computers began to be widely used in heavy industries, among other things for visualization of production data, the term Human Machine Interface (HMI) has appeared. HMI is one of many types of user interfaces, the other definition often used in the context of computer systems and electronic devices. The user interface of a mechanical system, a vehicle or an industrial installation is often called the HMI. [2]

As the user interface represents some computer device or a program, HMI represents industrial processes and devices (as for example, motors, pumps, valves and dumpers). HMI gathers information from them by using, for example, Programmable Logic Controllers (PLCs). A PLC is a microprocessor used for automation of industrial processes.

Special types of communication mechanisms are used for communication between HMI and PLC. A number of standards are used for controlling this communication. As for example, the special type of Ethernet called Industrial Ethernet (IEEE 802.3u) can be used between HMI and PLCs. Field-busses are often used for communication between PLCs and industrial devices (the same motors, valves, etc); one example is Profibus (IEC 61158/EN 50170). AS-Interface (EN 500295) can be used for gathering data from sensors and indicators. All three are international standards of network and communication interfaces, and more of them exist.

In the past, any integration between HMI and PLC systems was typically designed manually, because of a lack of standards for a common framework. This approach had several disadvantages that were described in an article [3]:

- Each software application had to include a driver for a particular hardware device
- There were conflicts between drivers of various manufacturers, caused by the fact that some hardware features are not supported by all driver developers.
- A change in the hardware's capabilities could cause functionality failures of some drivers.
- Conflicts when accessing the HW device could occur, because two different SW packages cannot use the same device at the same time since they each contain an independent driver.

All these disadvantages caused an evident need of a standardized solution, a framework that enables data from PLCs to be shared and transformed into usable information in a quick and efficient way. "This framework should be robust enough so that, as new situations arise, information can be exchanged and analyzed in ways not previously anticipated" [4]. To solve the problems a new communication standard - OLE for Process Control (OPC) was created. OLE is a shortening for Object Linking and Embedding.

2.2.3 OLE for Process Control

The OPC technology was developed as an industrial standard. OPC defines an open interface over which PLC and HMI components are able to exchange data.

OLE/COM part stands for Microsoft's object technology that aims at integrating applications with a great deal of compatibility between the various applications. [4]
OPC expands OLE by including structure definitions, interfaces and techniques for a more efficient data transfer. OPC uses all the advantages of OLE, but adapts it to the requirements of the process control industry [6]. In other words, OPC makes it easier for a user to control and monitor hardware devices on a Windows OS platform, regardless of the device vendor.

The usefulness of the OPC standard is discussed and proved in another article [7] which concludes that: "Because of the distributed architecture based on COM/DCOM is well supported by the Windows platform, and the OPC specification is mainly based on COM/DCOM technology, the distributed data integration using OPC technology is a suitable and acceptable choice. It is easy to design and implement". Among other characteristics flexibility, upgradeability, openness and efficiency are mentioned, and are all definitely important qualities. It is no less important that the OPC improves the reliability and accuracy of a HMI system, because it is sometimes absolutely necessary that the data from PLCs are exact and up-to-date. This is especially the case when it comes to dangerous environments in some industries. As an example, a dangerously high gas concentration on an oil platform has to be detected and warned about at an early stage in order to prevent accidents.

These are the challenges that Origo Engineering AS has to work with, as they are developing control and automation systems for critical environments. They have chosen to use Cimplicity HMI software for developing tailor-made systems for the customers. Cimplicity HMI is a part of GE Fanuc Automation's "Process Execution and Supervisory Control" family of Intelligent Production Management solutions. It is designed using the standards and 32-bit code from Microsoft.

2.2.4 Cimplicity

The Cimplicity HMI system is used for communication with the controllers, processing data from sensors and devices, and presenting data to the user (operator). The presentation of data is done in text, graphical or even alarm form (for example, phone message or email warnings). The communication between the HMI and the controllers is both for collecting data and for commands from the operator [3]. The HMI system has a server-client architecture and can use different protocols for communicating with PLCs. Cimplicity can, for example, communicate with PLCs produced by GE Fanuc with the use of a proprietary protocol called GE-SRTP. The previously mentioned OPC standard can also be used for communicating with PLCs. Figure 3 demonstrates the structure of the Cimplicity system [9].

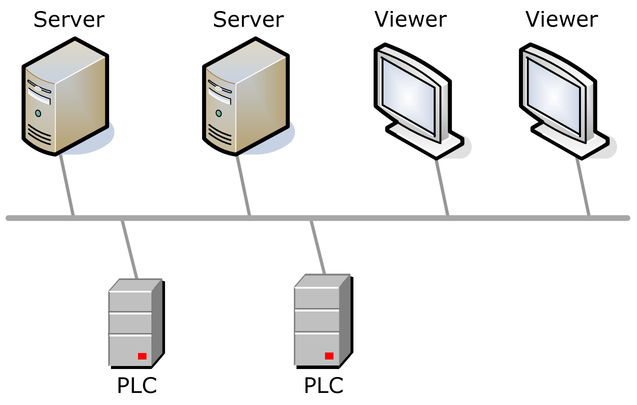


Figure 3 - Cimplicity

The Cimplicity system consists of two main parts; Servers and viewers. Servers collect and distribute system data from PLCs. They seamlessly share data while providing users with a real-time view of the processes being monitored. Viewers allow users to view and interact with the data distributed by the servers, and also to perform control actions.

The information architecture for the Process Industry involves the following 3 levels, as they are described in [8]: Business, Process and Field Management. Consistent communication between all 3 of them is a motivation for a reliable automation and control system. And the Cimplicity HMI claims to simplify and improve each level. On a business level it simplifies report generating, and then accomplishes integration of information collected from processes into a business management system, for example, for further managing of financial aspects.

It also improves supervisory control of processes (level 2), not only showing relevant and instant information, but also allows storing it in a structured form (as in a database). Cimplicity HMI gathers data in an object oriented form, but can then further connect with and store the information in relational databases like Microsoft Access, Microsoft SQL Server, Oracle and others.

At the Field management level, Cimplicity HMI allows connecting with hundreds of devices from different manufacturers, and then enlarges the amount and wealth of information that is available for collection and monitoring.

The feature that makes the system flexible is the OPC allowing Cimplicity HMI to be integrated with other systems. Two OPC components are available for such purposes:

- OPC Client capabilities are built into HMI Servers and allow for an easy integration of third-party device communications drivers (for example, PLCs from other vendors than GE Fanuc)

- OPC Server provides the same open integration capabilities to third-party software packages.

The piece that still needs to be purchased (it is not the part of the Cimplicity) is an OPC Device Communication server. One of the possibilities is the OPC Server for device communication from MatriconOPC. MatrikonOPC also produces OPC servers and clients that can be used instead of those that come with Cimplicity. MatrikonOPC is one of the world's largest OPC developers and more information about it and its products can be found in [28].

As it was mentioned in the problem description, HMI servers are customized for customers with the help of an amount of scripts. Cimplicity has built-in tools that allow creating scripts and programming applications with a Visual Basic style language. This language has over 600 basic functions that directly integrate with Cimplicity points, alarms and the error logger. Scripting is provided to extend the HMI capabilities for tailoring individual applications according to specific needs. Scripts can be executed based on process events, time or in a specified order [9].

Also included is a tool that allows defining when to start the execution of the script, responding to process events. An event can be defined as a changing point (for example, a limit in a temperature or a gas level), as an alarm state, or based on the time of day. One event may invoke multiple actions and one action may be invoked by many events.

The Basic Control Engine monitors for events and executes the scripts. This is Cimplicity's own running tool. Based on a multi-threaded scripting design, it can invoke and execute multiple programs concurrently. Queued Script Execution extends the functionality of the Basic Control Engine. It allows for setting up scripts in a queue, in the order of which they will be executed later.

These tools are being used by Origo in the development of scripts, according to the customers' needs. The scripts are usually small programs with limited functionality. Examples of scripts that are used today provided by Origo are: Gasstrend, Network monitor, Report Generator, Client - Server monitor, SAP Integration, C&E monitor, E-mail/SMS notification, etc. Figure 4 illustrates the scripts on the HMI server.

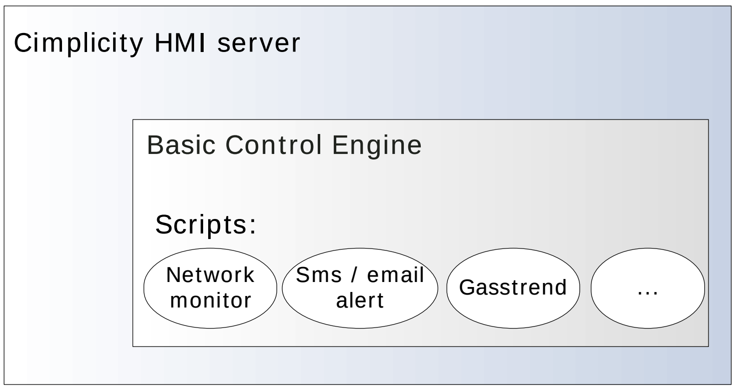


Figure 4 - Scripts on the HMI server

Origo’s final product is a complete automation and control system, that includes both hardware and software parts that are ready to use.

3 Problem research and state of the art

Cimlicity HMI system is tailor-made for each customer. The main part of this is done by creating the amount of small programs with limited functionality. These programs are being called utilities later in the report. As it was illustrated in Figure 4, an example of such a utility can be a program that pings or checks networks status of PLCs, and returns the table with results – a Network monitor. The other example is the program that alarms operators with a message on the phone or e-mail if the temperature level measured on indicators is getting higher than usual. Origo offers a number of different utilities of this kind, and each customer can choose between those he finds useful. The utilities are then added to the complete solution of the HMI system.

3.1 Challenges today

Three challenges with this method were mentioned in the task description. Here they will be further analysed. The first two challenges, reusability and dependency are relatively close in relation to their solution, and will be described together.

3.1.1 Reusability and dependency

The situation today is that developers have to rewrite very similar scripts for creating similar utilities again and again according to customers' specifications. Different customers order different sets of utilities. The sets are assembled from the number of available scripts. Those utilities are then programmed as scripts directly on the HMI server. They can't be reused because they have to be tailored for each particular HMI server. Copy and paste method saves some time, but some changes still have to be made manually. The copy and paste method causes small and annoying mistakes in scripts. At the end, a lack of reusability results in a waste of valuable time.

The other problem is dependency. The utilities are developed uniquely for each HMI server and depend on their HMI for running. They have to be coded from the beginning, for example when a customer gets a new HMI server, even if the requirements stay the same.

Such waste of time and efforts doesn't seem to be the most efficient way of working. It seems logical that it should be possible to reuse the utilities. In order to achieve the reusability, they have to become independent of the HMI server. The independency implies that utilities are not coded directly on HMI and don't use HMI's control engine for running. The utilities have to be re-created as independent small programs with a similar functionality that the scripts from the HMI server have. The utilities still have to communicate with the HMI in order to get data from PLCs. And it has to be possible to create different "sets" of utilities, according to a customer's demands.

This brings us to the idea of a Utility Framework Server – a complete solution for the challenges with scripting on the HMI. The hardware part is obvious – the Utility Framework Server is a personal computer with an installed operating system, and placed on the same network as the HMI and PLCs (as shown on Figure 3). As for the software part, the Utility Framework Server is a stand-alone application with these main purposes:

- To act like a server for managing clients - utility programs.
- To communicate with HMI server in order to exchange the information with PLCs.

The Utility Framework Server should also solve the three challenges listed in the project description. As for the part about communicating with HMI server, the use of third-party software is anticipated, as for example an OPC server.

3.1.2 Error handling

The third challenge lies in the error handling mechanisms. The main cause of the problem is still the same – scripting on the HMI. As the utilities are unique for each HMI server, they don't have a standard error handling mechanism. The utilities are placed and run directly on the HMI, and a crash in the utility can cause a crash of the HMI server. As the crash of a HMI server is highly undesirable, methods for generalizing error handling have to be explored. The method used today is suggested by the producers of the Cimplicity HMI software, and implies doubling the hardware resources. Figure 5 demonstrates the technique as it is described in the developer manual [9].

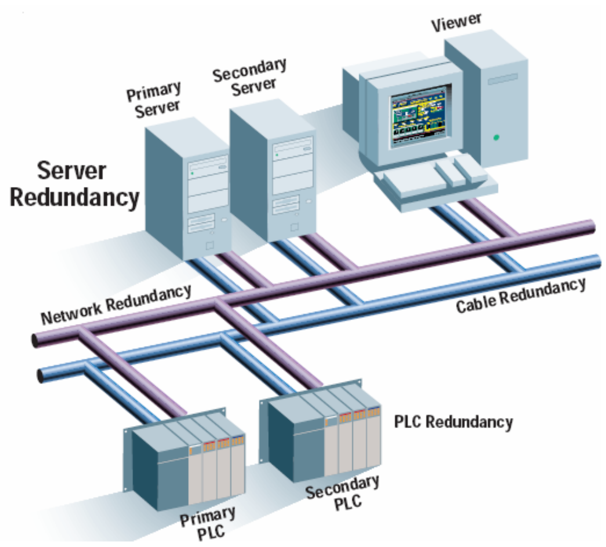


Figure 5 - Error handling with redundancy

The method is first and foremost used for preventing hardware errors. Instead of one hardware / software package, containing server, network and PLC, two similar packages are delivered to the client, one as primary and the other as secondary.
Cimplicity projects are configured so that a secondary system is ready to take over operations automatically when the primary system fails. It provides an automatic switchover from the primary system to a backup, in the event of a hardware failure or an application crash. All the main HMI functions are transferred during the failure, so critical data acquisition, alarming, logging and security operations continue without interruption.

This technique works in practice (as attended, mainly for preventing the hardware errors), but due to its cost, it cannot be called efficient. Doubling of hardware resources increases the price considerably. The other issue is that if the scripts are absolutely identical on both servers (primary and secondary), they can probably cause errors at the same time and under the same conditions. In this case, it doesn't make sense to use the redundancy as a software errors preventing method.

3.2 *Modern software technologies*

The three challenges – reusability, dependency and error handling - with the scripting on the HMI are likely common for today's software. In fact, the service-oriented philosophy was created in order to find the solution for these and similar challenges. Service orientation is a relatively new definition in programming and will be presented later in this section. At first, the problems with modern software systems and the development process of the solutions are presented. The question is how are they relevant for the three challenges that are a part of the problem description of this project?

3.2.1 Traditional ”old fashioned” architecture

Larger systems are designed and implemented as a number of classes that are usually developed based on Object Oriented Programming (OOP) principles. OOP is a programming “philosophy” that was used commonly since 1990 and resulted in an amount of well-designed and well-working solutions. All the same, some inappropriate results can appear in such systems. One of the causes is that components of such a system are tight-attached with each other, as they use and exchange data as objects / classes and often shares data structures. As it was discovered in practice, a large OOP system lacks or has in a lesser way the next features:

- Ability for updating

When parts of the system have to be updated or corrected, it often causes influence on the whole system. The result is that it may be necessary to take the whole system or parts of it out of service. Obviously this is not always possible or desirable, as in oil-industry where software systems have to be working and reliable at all time.

- Reusability

In software development there is often a need for reusing the parts from earlier developed systems, and it is desirable to make systems as generic as possible, particularly with respect to reusing some parts of the code later, in projects with similar requirements. Even so, such systems are often so tight-bounded in itself at the end, that it is difficult to reuse parts without time-consuming adjustment that makes its reusability unprofitable.

- Extensibility / Scalability

Shifting or extending of hardware resources can also be problematic. Big and advanced systems often employ clustering methods. For a number of years it was easier to add more hardware and software parts to the system than to change it as a whole, and it is typical for not just the oil, but several other industries. This causes the need for highly educated expertise, a lot of money and resources to make the switch to other and more modern technologies. With regards to this, developers now have to think about how their systems can be renewed in the future.

- Maintainability

All the three previously explained problems make the systems expensive to maintain, at least because of its complicity. The larger the system is, the more vulnerable it to the challenges.

3.2.2 Interfaces vs. Inheritance

In object-oriented analysis and design, an application is modelled as a complex hierarchy of classes that serve the common goal to achieve the requirements as close as possible. A developer can re-use existing code by inheriting it from an existing base class and specialise its behaviour. To be able to do that, a developer must be familiar with the details of an implementation of the base-class. This form of re-use is called “white-box reuse” and it doesn’t allow for economy of scale of large programs, or easy adoption of third party frameworks.

Component-oriented applications support black-box reuse, which allows reusing existing components without caring about its internals, as long as the component complies with some predefined set of operations or interfaces. The component-oriented frameworks still allow developers to use inheritance in implementations. Nevertheless, the developers are encouraged to concentrate themselves on factoring out the interfaces that can be used as contracts between components and clients, instead of designing complex class hierarchies. Using interfaces instead of inheritance (which is common for OOP) between the classes is one of the main ideas behind Component Oriented Programming (COP). COP can be presented as “OOP with role modelling (interfaces)”. The use of an interface alone is said to solve the challenges in a certain degree. As for example, the use of the interface is said to improve the reusability.

By practicing, it was discovered that it is easier to improve the mentioned challenges with the large software systems by dividing or breaking down the whole bigger system into smaller components. As a result, the system becomes faster in production, more robust and more scalable, and the development and maintenance costs go down. Instead of object-oriented architecture we are now looking at a component-oriented approach that is the key-factor of .NET development.

To describe what component-oriented programming is, it is necessary to define what is meant by component at first. By its definition, the component is simply the part of something; related to software developing, the component of the system is usually the class. Each .NET class is a binary component, which is the definition for component that is used later in this report. The .NET Framework is used as an example here, but is of course not the only framework practicing COP.

Component-oriented programming is different from object-oriented programming; although the two methodologies have things in common (remember its evolutionary nature, as it was mentioned before). To shortly explain it, the object-oriented programming focuses on the relationships between classes that are combined into one large binary executable, while component-oriented programming focuses on interchangeable code modules that work independently and don’t require knowledge of how the other modules work in order to use them.

3.2.3 Component oriented programming

In a traditional object-oriented vision, although the logic is divided between a number of classes, once those are compiled, the result is a monolithic binary code. All the classes share the same physical deployment unit (for example, EXE) process, address space, security privileges, and so on. If several developers work on the same code base, they have to share source files. In an application, a change made to one class, can cause re-linking of the whole unit, and result in a need to exploring and re-testing all of the other classes. A component-oriented application comprises a collection of interacting binary application modules: components and calls that bind them. Figure 6 visualizes the main principle of a component-oriented application [10] [27].

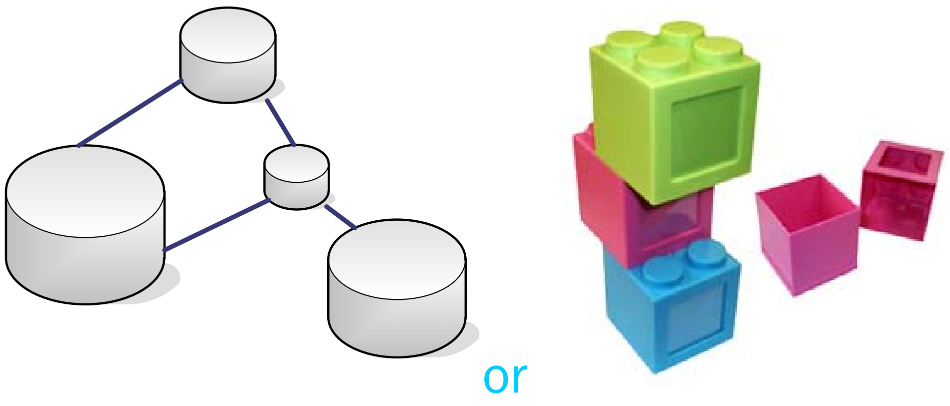


Figure 6 - Component-oriented application

Binary components can have different purposes, some general as communication or file-access components, or developed especially for the exact application. An application implements and executes its logic by gluing together the functionality of its components. Component-enabling technologies as COM, J2EE, CORBA and .NET provide the infrastructure needed to connect binary components in such a way, that the connections are invisible and unimportant for the user. One can imagine a component-based application that has been built from a number of lego-blocks. They can look different and can have different purposes, but all together create the structure that one wanted to achieve. One can also add or remove more of them if needed.

The main question of this section is how COP does contribute in solving the previously mentioned challenges in software systems?

- Ability for updating

Updates made to one component contained to that element only. No existing client of the component requires recompilation or redeployment. Components can even be updated while a client application running, as long as the component is not currently being used. Improvements or fixes of a component will immediately be available to all applications that use this component, whether on the same machine or across a network.

- Reusability

As the functionality is shared between components, re-using is getting easier. Again, like with Lego, one can build different towers and castles re-using the same Lego blocks. Here comes one more difference between component-oriented and object-oriented design, using interfaces instead of inheritance that allows reusing of components without knowing how they do things, but just what they are doing. More about interfaces is later in the report.

- Extensibility / Scalability

A component-oriented application is easier to extend, as well. After getting new requirements, they can be implemented by adding new components, without touching existing components that are not affected by the new requirements.

- Maintainability

These factors reduce the cost of long term maintenance of a component-oriented application. Development time is getting shorter, because of the reusability factor. One can select from a range of available components, either earlier self-constructed systems or from third-party vendors, and thus avoid repeatedly "reinventing the wheel" or doing the same thing over and over again, something that is not just time consuming, but also not so motivating.

The next step towards reliable, scalable and easy maintainable systems has become an appearance of the term "SOA".

3.2.4 Service oriented architecture

Service orientation (SO) is said to be an evolution of OOP. A subtle difference with SO is that it can be used to model not only a particular system architecture but larger-scale constructs such as the way systems interact within the organization, or even how organizations interact with each other in the outside world. Service orientation is becoming more popular today and is predicted to dominate enterprise application integration in the future.

Service oriented architecture (SOA) has as its major goal to achieve the independence between systems and components. SOA focuses most on functionality of a future system, in preference to anything else. Platforms, programming languages, algorithms and protocols take second place. As long as all the parts are able to communicate with each other using some communication standard, they can be as different and as ignorant to each other as they, or a developer, want.
"The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without the service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks". [12]

Programmers tried to achieve SO-like architecture for quite some time now, for example through distributed objects as DCOM or CORBA, and also in first generation web services. That is why SOA, as of today, looks more like evolution of previous technologies, rather than revolution. It is not something completely new; it is in some way a further development of the object oriented architecture. This evolutionary aspect makes moving towards SOA less drastic and makes the learning curve less steep.

It was decided to use WCF in the designing and implementation process, and WCF claims to be a SOA-oriented communication technology. In the next section it will be analyzed how WCF employs the principles of SOA. The most important question, what can WCF do for this particular project?

3.2.5 WCF as SOA-oriented communication model

An application that provides and consumes services is not a new idea. What is new in WCF is a focus on services instead of objects. WCF developers had SOA in mind while designing it and that is where its basic principles are coming from.

- Share schema, not class

Unlike older distributed object technologies, services in WCF interact with their clients only through a XML interface. Behaviours such as passing complete classes or methods across service boundaries are not allowed.

- Boundaries are explicit

A goal of distributed object technologies such as Distributed COM (DCOM) was to make remote objects look as much as possible like local objects. While this approach eased development in some ways by providing a common programming model, it also covered the obvious differences between local objects and remote objects. Services avoid this problem by making interactions between services and their clients more explicit. Hiding distribution is no longer a goal.

- Services are autonomous

A WCF service and its clients agree on the interface between them, but are otherwise independent. They may be written in different languages, use different runtime environments, such as the CLR or the Java Virtual Machine, execute on different platforms, and differ in some other ways, as long as they both support the SOAP message standard.

- Use policy-based compatibility

When possible, determining which options to use between systems should rely on WS-Policy-based mechanisms [14].

Motivation for implying the two last principles was:

- To create the technology that guarantees compatibility between different platforms and contractors, and
- To allow the developers to choose the communication technology.

By focusing on services and implying SOA principles, WCF reminds more about web services, one of the other technologies for creating the distributed systems. According to the W3C [21], a web service is defined as "a software system designed to support interoperable machine-to-machine interaction over a network".

Figure 7 illustrates standards for message exchanging that are typically used in web services [29]. WCF implies the same standards and those will be described later.

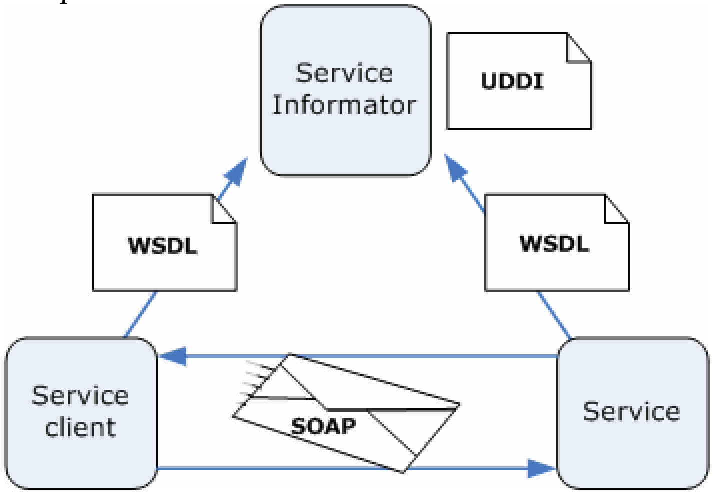


Figure 7 - Standards in web services

WCF employs WSDL standard for describing of services and SOAP messages for communication between a service and its clients. This makes WCF-based applications interoperable with any other process that communicates via SOAP messages. WCF supports WS-I Basic Profile specification, and it lets it communicate with other non-WCF applications, as long as those also support WS-I Basic Profile.

Basic Profile defines if it is SOAP or WSDL or UDDI that is used, as long as a web service follows the Basic Profile specifications.

WSDL – Web Service Description Language specifies how a web services is described, so that the client can communicate with it. This description contains information about what protocol, message format and operations that service supports [15].

UDDI – Universal Description, Discovery and Integration is a "telephone book" for services where web services can be registered, so it became possible to find and use web services that others have made and displays for disposal [16].

In WCF a message is the basic unit of data that is exchanged between server and client. WCF permits the serialization of a message in a variety of formats including binary, text XML or some custom format. SOAP is a protocol for exchanging messages, SOAP is transport neutral, and it means that it is independent from the underlying transport method. Typically HTTP is used, but also plain TCP or other protocols such SMTP or FTP can be used [17].

Beside WS-I Basic Profile, WCF also supports some WS-* specifications that was made by Microsoft, SAP and IBM, and that expands WS-I Basic Profile specifications with new possibilities for security, reliability and transactions. [18] [19]

Compatibility between different constructors is quite important for this project. While platform compatibility is less important, as the most if not all software suppliers for oil-industry use newer versions of Windows based platforms, but the compatibility between constructors is significant. In future, the client for Utility Server system can be created in different programming languages and by different developers. And they still have to be available to consume the same services from Utility Server.

The other problem with current technologies is that developers have to choose connection methods for distributed applications pretty early in the development process, because a choice of design architecture and algorithms are dependent on communication methods. If the communication requirements change, it will not be easy to reuse the old application with new communication methods.

On the other hand, WCF isolates application logic from communication methods. The reason for this is that the communication mechanism is only defined in the configuration file, and WCF allows choosing from different communication methods. So for changing the communication mechanism it is only necessary to change settings in a configuration file.

As for this particular project, it is unknown if the changing of communication methods will be necessary in the future, but extra flexibility is always an advantage software applications.

4 Design of the Utility Framework Server

The idea of the Utility Framework Server was briefly introduced in the previous chapter. Chapter four is dedicated to explaining the design of the new server and how it is going to solve the three challenges with the scripting on the HMI. Figure 8 presents the first model of the Utility Framework Server, main principles of which will be described immediately after.

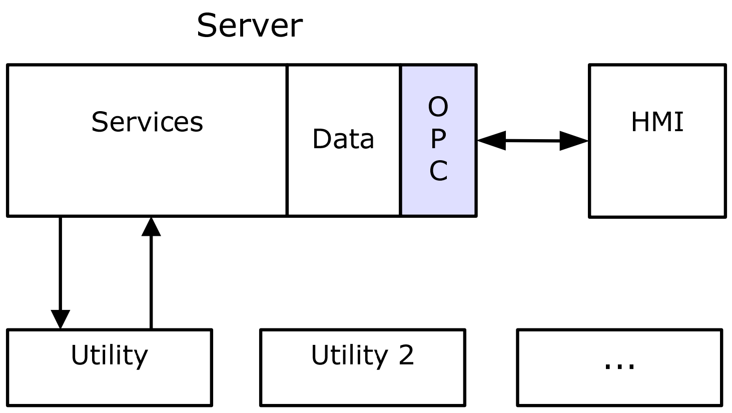


Figure 8 - Utility Framework Server

The new server is going to act as a connecting link between HMI server and utilities – small programs that reproduces functionality of the old scripts. The server will take care and to transfer data between the utilities and the HMI server. The server itself uses OPC standard for connecting to HMI server. The other purpose of the server is to manage the utilities. It has to be possible to, for example, add new and to remove old utilities. This will increase the scalability of the system, and will make it easier to adapt the HMI system to the orders from the customer.

The scripts from HMI will be presented as utilities and act like clients against a new server. The relocation of these from the HMI server on their own server helps with the dependency and partially with the error-handling problem. It also helps to improve the reusability in a long-term view.

All the new recreated utilities must have the same "skeleton" or have to follow the same design standard. This allows that the utilities will be able to use the same services from server, and sends and receives messages in the same format. The communication between the Utility Framework Server and its utilities goes both ways, from server to client, and from client to server. The communication between the clients is not the requirement. In order to allow the other way communication, from client and to the server, all the clients have to provide same services – methods located on clients that will be used by server. In this case the clients have to act like a server.

System requirements present detailed description of the functionality of the Utility Framework Server.

4.1 System Requirements

After analyzing the problem description, the system requirements for the Utility Framework Server were set up. These requirements relate to the software part of the Utility Framework Server, the hardware part is granted by supervisor.

The requirements are divided into 2 categories: functional – those are directly about systems functionality, and non-functional – all kinds of side-requirements. The requirements are set up in a three-structure, as some of them deepen others.

All requirements follow the same description standard:
Requirement <ID>: <short name>, **Priority**: <A- most important; B- important; C - least important>
Description: <What is the requirement about?>

4.1.1 Functional Requirements

↳ **Requirement** R1: Utility Framework, **Priority** A
Description: The Utility Framework is a distributed system for managing utilities.

↳ **Requirement** R1.1: Reusability, **Priority** A
Description: The Utility Framework should solve the reusability problem with HMI.

↳ **Requirement** R1.1.1: Standard for utilities, **Priority** A
Description: The utilities should follow the same "skeleton" in design, probably by using the same interface.

↳ **Requirement** R1.1.2: Standard for communication, **Priority** A
Description: The server and utilities should have a standard interface for connection between them.

↳ **Requirement** R1.1.2.1: Knowledge about services, **Priority** A
Description: The utilities should have knowledge about available services.

↳ **Requirement** R1.1.2.2: Registering of utilities, **Priority** A
Description: The server should make services available for clients (utilities).

↳ **Requirement** R1.1.2.3: Registering of utilities, **Priority** A
Description: The server should have a mechanism for registering of utilities.

↳ **Requirement** R1.2: Dependency, **Priority** A
Description: The Utility Framework should solve the dependency problem with HMI.

↳ **Requirement** R1.2.1: HMI connection, **Priority** B
Description: The Utility Framework should have a method / service for communication with HMI server.

↳ **Requirement** R1.3: Error handling, **Priority** A
Description: The Utility Framework should solve the error handling problem with HMI.

↳ **Requirement** R1.3.1: Error detection, **Priority** B
Description: The Utility Framework should have mechanisms for error detection.

↳ **Requirement** R1.3.2: Error display, **Priority** B
Description: The Utility Framework should have mechanism for showing, logging and reporting detected errors.

↳ **Requirement** R1.3.3: Error handling, **Priority** B
Description: The Utility Framework should be able to react to detected errors.

↳ **Requirement** R1.4: Scalability, **Priority** A
Description: The Utility Framework should be able to handle a number of utilities (more than one).

↳ **Requirement** R1.4.1: Utilities, **Priority**: A
Description: The utilities can be places on the same (local) machine or on the other on network.

↳ **Requirement** R1.4.2: Concurrency, **Priority**: A
Description: The Utility Framework Server should have some mechanisms for concurrency handling of utilities.

↳ **Requirement** R1.4.3: Flexibility, **Priority**: A
Description: The Utility Framework Server should handle utilities of different kinds (in functionality, developers, etc.).

↳ **Requirement** R1.5: GUI, **Priority**: A
Description: The Utility Framework Server has to have GUI, from which the user could operate the utilities, and also perform some operations with them.

↳ **Requirement** R1.5.1: User input, **Priority**: A
Description: User input should be possible through a mouse.

↳ **Requirement** R1.6: Utility Manager, **Priority**: A
Description: The Utility Framework Server should manage the available utilities.

↳ **Requirement** R1.6.1: Adding utilities, **Priority**: A
Description: It should be possible to add new utilities to the server.

↳ **Requirement** R1.6.2: Removing utilities, **Priority**: A
Description: It should be possible to remove or deactivate some utilities from the server.

↳ **Requirement** R1.6.3: Running, **Priority**: A
Description: It should be possible to run and to stop the utilities from the server.

↳ **Requirement** R2: Utilities, **Priority**: A
Description: The utilities form the part of the Utility Framework Server. The utilities should be able to employ the server's functionality. Later, a couple of examples for utilities have to be created for testing of the prototype.

4.1.2 Non-functional Requirements

Requirement NR1: Documentation, **Priority**: A
Description: Design has to be well-documented, so that Origo could take over further development.

Requirement NR2: Technology, **Priority**: A
Description: Using of .NET technology is required.

Requirement NR3: Programming language, **Priority**: A
Description: Using of C# programming language is required.

Requirement NR4: Copyright, **Priority**: A
Description: No Origo's business information should come out public.

4.2 How does the Utility Framework Server implement the requirements?

The main idea is to relocate some functionality from HMI server, and place it on its own server. This relocation idea itself participates in solving the three challenges that were mentioned before: reusability, dependency and error handling. These challenges are listed as requirements R1.1, R1.2 and R1.3. The chosen solution technology (.NET and WCF) partially implements some other requirements, as it has built-in methods that help to solve those kinds of problems (for example, the requirement R1.1.2). Some implementations have to be made to implement the remaining requirements. Table 1 demonstrates the ideas and methods for solving functional requirements.

Table 1 – Requirements and solutions

Requirement ID	Solutions
R1.1	The functionality from scripts from HMI will be implemented as independent utilities and managed by the Utility Server. Those utilities can be reused in different HMI servers, because Utility Sever will use an OPC standard for communication between those two.
R1.1.1, R3.1	All utilities should follow the same structure. It will be implemented probably by using the same interface for all utilities or just by containing the same class' structure: some variables and methods must be common for all the utilities.
R1.1.2	Communication between all the utilities and the server should be standardized. If it is assumed that communication goes both ways, and by using WCF, it will be implemented by 2 steps: • The server exposes a number of services. All the utilities know about these services and can use them. Although different utility can get a different data from server, and return different results, the type of data must be the same. As in Example 3, the method SendMessage that returns string value is the same and known to all utilities, but the value can differ. • The server use methods exposed by clients. These methods should probably be the same in all the utilities or comes from the list of possible methods that the server knows about and can use.
R1.1.2.1	Mechanisms in WCF for providing the information about the available services to the client should be explored.
R1.1.2.2	The server should be up and running.
R1.1.2.3	The method for registering clients/utilities should be implemented.

R1.2	Physically removing utilities from HMI server obviously reduces dependency. The utilities will no longer be running directly on the HMI; instead they will use its own server. The Utility Framework Server should be able to communicate with different versions of the HMI server, probably again with the help of a third-party software, that uses OPC standard.
R1.2.1	This is not the first priority requirement. It is assumed that this part will be granted by using the third-party software provided by Origo.
R1.3	As the scripts are no longer running directly on HMI server, they will not cause a risk of crashing of it.
R1.3.1, R1.3.2 R1.3.3	Methods for error discovering, handling and showing errors should be implemented in the Utility server.
R1.4, R1.4.2	Should be implemented.
R1.4.1, R1.4.3	Mechanisms for this in WCF should be explored or else implemented. Some functionality comes also out of implementing the requirement R1.1.2.
R1.5, R1.5.1	Should be implemented.
R1.6, R1.6.1, R1.6.2, R1.6.3	Should be implemented.
R2	Some examples of the utilities should be implemented.

Out of system requirements, Utility Framework functionality is constructed. One more thing has to be mention here; it is about the "Utility Framework Server" conception. In fact, this definition varies in the report. In some cases, services, GUI and other components but without the utilities are defined as the Utility Framework Server. In other cases, utilities (or clients of services) are also a part of this definition. This difference in using the term is not critical of understanding the context; but is done for reasons of clarity.

4.3 Use-case model

Use-case diagram is used to describe the actions between the actors. The actors are the parts of a system that were presented before: Utility Client, Framework Utility Services and GUI, and HMI server. Figure 9 presents Utility Framework functionality in a form of the use-case diagram, as an easy and schematic way.

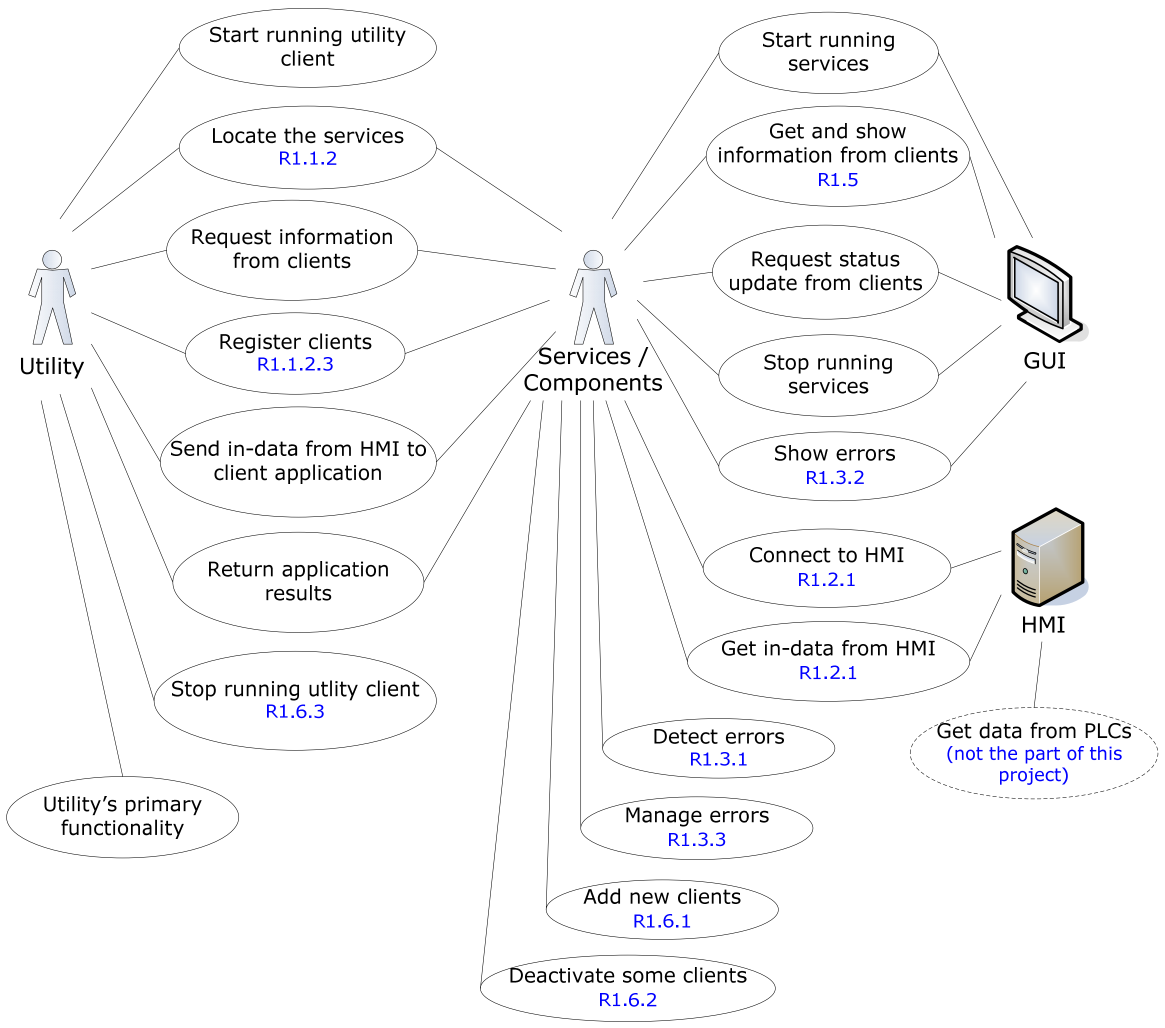

Start running utility client
Locate the services R1.1.2
Request information from clients
Register clients R1.1.2.3
Send in-data from HMI to client application
Return application results
Stop running utlity client R1.6.3
Utility's primary functionality
Utility
Services / Components
Start running services
Get and show information from clients R1.5
Request status update from clients
Stop running services
Show errors R1.3.2
Connect to HMI R1.2.1
Get in-data from HMI R1.2.1
Detect errors R1.3.1
Manage errors R1.3.3
Add new clients R1.6.1
Deactivate some clients R1.6.2
GUI
HMI
Get data from PLCs (not the part of this project)

Figure 9 - Use case diagram

As it showed by diagram, some actions are divided between two actors; and some of actions are performed by actors themselves. Some of the actions came directly from the requirements, and they are marked with the requirement ID. The others implement the side-functionality. For example, actions that helps to solve the requirements or actions that are logically inevitable (as start and stop executing clients, etc.).

4.4 Component model

As it was decided to use component-oriented programming, the whole system are divided and presented as a number of blocks – components. I want to specify that the components are not the layers of the system – they are "independent" parts. Each component has its own role, which is implemented by the number of functions. Also in order to follow SOA and COP principles, each component (as class) implements an interface. In the model the utilities are placed outside, it is done to underline that they can be placed physically "outside" the server system, and by meaning of that, on the other machine on a network. They can also be placed on a local machine – in this meaning, the same machine that the server. Figure 10 shows the component structure of the Utility Framework Server.

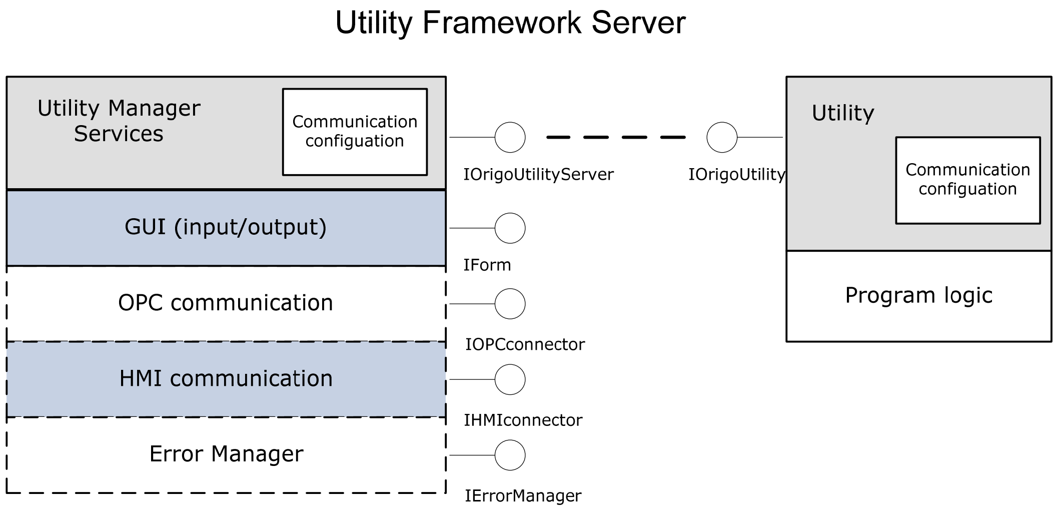


Figure 10 - OUF in components

Each block in the figure represents a component. Each component is responsible for each own part of the system requirements. Utility Manager is responsible for managing the utilities. GUI component is responsible for the graphical presentation of the server. OPC and HMI communication parts are as names implies responsible for communication with HMI and OPC systems. Error Manager is responsible for error detecting, logging and displaying. The Utility part forms the part that is identical for all the utilities. Program logic that varies for each utility comes in addition to the Utility component. The relations between the components and the requirements are demonstrated in Table 2.

Table 2 - Components and requirements

Component	Requirements
Utility	R1.1.1, R3.1
Communication configuration, client side	R1.1.2.1
Communication configuration, server side	R1.1.2
Utilities Services	All the requirements about managing the utilities (register, add, delete osv.) and R1.4 with all the sub requirements
GUI	R1.5 and the sub requirement R1.5.1
OPC and HMI communication	R1.2.1
Error manager	R1.3 and all the sub requirements

Domain Specific Language (DSL) class diagram is created based on functionality described in use-case diagram and components figure. The whole model is presented and further described in Figure 23 in Appendix 4.

As it was specified earlier, the utilities are the weaker coupled components of the Utility Framework. In order to loosen the utilities, WCF is used for communication between the server and the clients - utilities. Section 4.5 describes how it is done.

4.5 The Utilities

For implementing the server-client communication in WCF, it is required to create a local stand-in for the service, called a proxy. The term "proxy" defines a software design pattern. Shortly, a proxy is a class that acts as an interface for something else. This "something else" can be anything: a network connection, a file, a large object in memory or some other resource that is expensive or impossible to duplicate. In WCF a proxy is being created for each and every service call made.

The proxy is connected to a particular endpoint on the target service. The client then invokes the service's operations via its proxy, and simulates the server. Figure 11 shows how this looks and has been reproduced from [14].

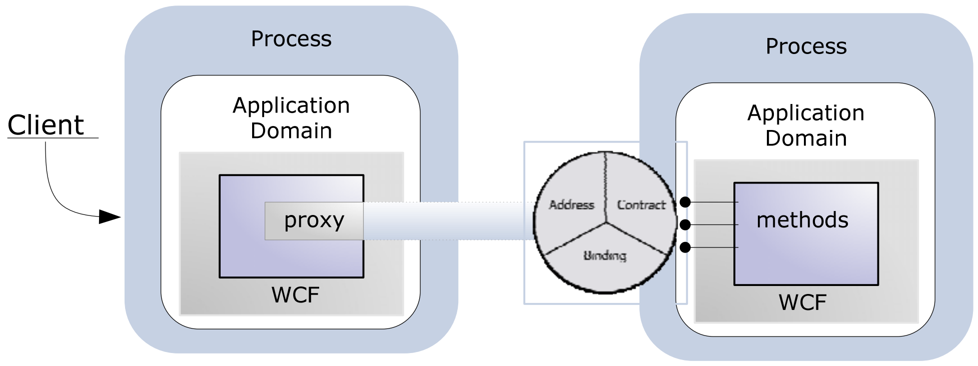


Figure 11 – Proxy

Creating a proxy requires knowing precisely what contracts are exposed by the target endpoint. Those contracts definitions are used to generate the proxy. In WCF, this process is performed by a tool called svcutil. If the service is implemented using WCF, svcutil can access the service's dynamic-link library (DLL) to learn about the contract and generate a proxy. DLL is Microsoft's implementation of the shared library concept. If only the service's WSDL definition is available, svcutil can read this to produce a proxy. If only the service itself is available, svcutil can access it directly using either WS-MetadataExchange or a simple HTTP GET to acquire the service's WSDL interface definition, then generate the proxy. When the proxy is generated, the client can create a new instance of the proxy, and then invoke the services by using it.

One more thing remains to be specified by the client: the exact endpoint it wishes to invoke operations on. Like a service, the client must specify the endpoint's contract, its binding, and its address, and this is typically done in a config file. In fact, if enough information is available, svcutil will automatically generate an appropriate client configuration file for the target service. A practical guide about how to generate the proxy and the configuration files for clients in Visual Studio or by using svcutil can be found in Appendix 3.

The fact that the communication between the clients and the server has to go both ways was made early in the design process. The possibilities for implementing this in WCF have to be investigated. The methods that are placed on a client side and are used by a server are referred as callbacks in WCF, and this is the term that will be used later in the report. A callback allows the service to call back to the client [20]. Figure 12 illustrates the services on a server and callbacks on a client side.

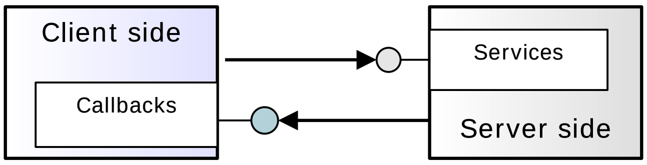


Figure 12 – Services and callbacks

Although callback objects are not full services (for example, one cannot initiate a new communication channel with a callback object), for the purposes of implementation they can be thought of as a kind of services. An example of a callback is when the server contacts the clients to inform them about some data updates.

The client is one who is going to host a callback object and expose a callback endpoint. All the client need to do for hosting a callback object is to instantiate the callback object and construct a context around it as it showed in Example 4.

```
class MyCallback : IMyContractCallback
{
      public void OnCallback()
      { … }
}

IMyContractCallback callback = new MyCallback();
InstanceContext context = new InstanceContext(callback);
```

Example 4 - Callback setup

The *InstanceContext* class (*ServiceModel* namespace from WCF, supported in .NET 3.0 Framework) provides a constructor that takes the service instance to the host. Any public static members of this type are thread safe. Any instance members are not guaranteed to be thread safe. The callback methods are being executed on a thread from the thread pool, it is therefore important to provide the thread safety in the callback methods and in the object that provides it. The use of synchronization objects and locks to access the member variable of the client is required.

5 Utility Server Prototype

After further discussing the model and architecture of Utility Framework Server prototype, a prototype model was created. This first implementation has limited functionality. The components that are drown with stroke lines will not be implemented because of the time limitations (see Figure 10). This means that the prototype will not have any communication with HMI or OPC, but so far works as a stand-alone application for managing the utilities. An error manager component will not be implemented either. Therefore the prototype only implements the requirements, which were assigned to the three other components – Utility Manager Services, GUI and Utility. The first prototype model is presented in Figure 13.

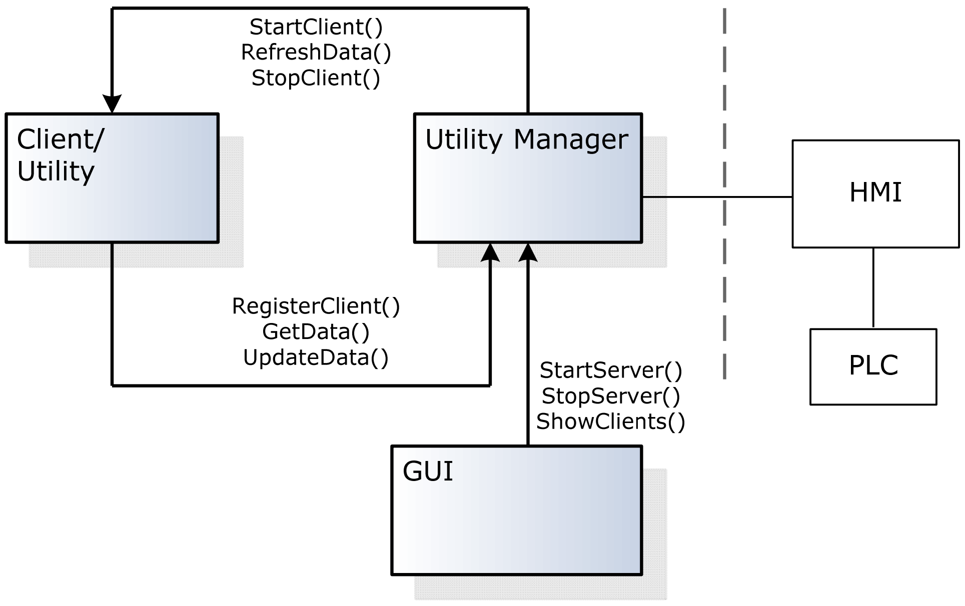


Figure 13 – Prototype model

The client uses following services from the server: RegisterClient(), GetData() and UpdateData(). The server has to "callback" next services from client: StartClient(), RefreshData() and StopClient(). The first challenge is to get the communication between the client and the server to work using the WCF technology and test its efficiency. Next step is to create a GUI part for the server that has next functionality: StartServer(), StopServer() and ShowClients(). Three scenarios with sequence diagrams further describe the prototype's functionality.

Scenario 1

The sequence diagram in Figure 14 shows the scenario when the server is started, it registers the clients (utilities), stores the information from the clients and displays it in the GUI.

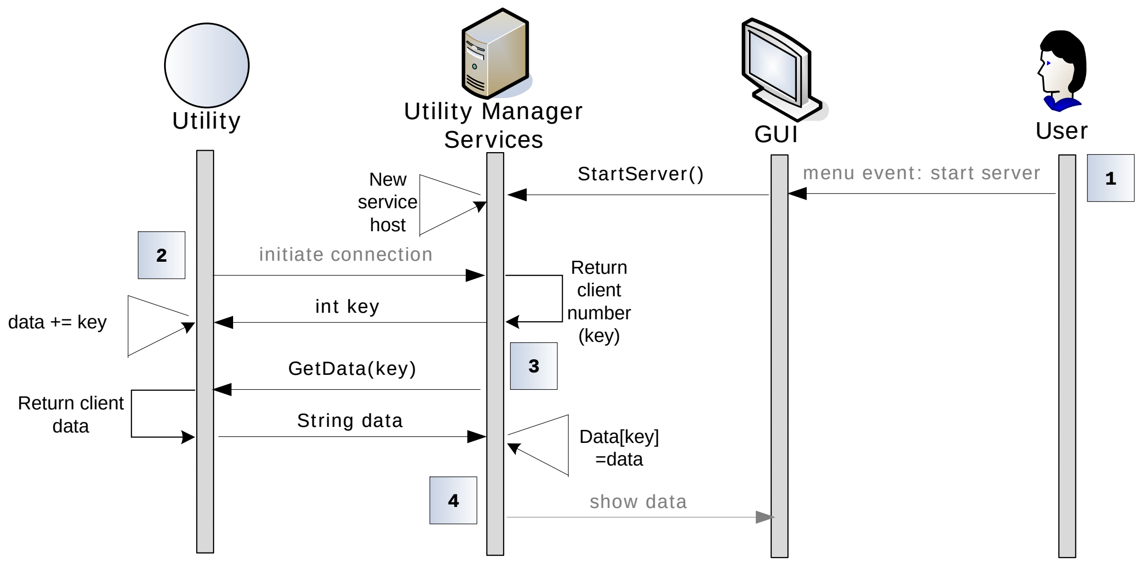


Figure 14 - Scenario 1

1. The service starts running after the action from the user (Menu – Start Server). If the service is not running, the utilities will not get an answer on theirs requests (in stage 2) and will throw an exception error message.

2. The utility client initiates connection: it asks for running the RegisterClient() method on the server side. As the result of this method, each client gets its number: the server distributes numbers from 1 to N for each client, based on the connection time. After getting its number, the client adds it to a data string, which already contains its name, IP address, some text description about a client and a list of keys and values (see scenario 2 for further information), gathered in one string.

3. The server asks the client utility for information. The utility sends its data string to the server. On the server this data is being stored in an array.

4. The server divides the string back into logical pieces (name, description, etc) for displaying it on GUI.

Scenario 2

As it is mentioned in the stage 2 in the scenario 1, the utility client sends keys and values as the part of the data string. These keys define the data that are used by the utility, and the values are the current values of this data. Therefore the keys and values vary in each utility. For example, for the Ping Client utility, the keys can be IP addresses that are going to be monitored, and a timeout variable that defines how often the check is run. For the SMS/Email Alarm utility, the keys are the email addresses and phone numbers that will be alarmed. These keys and values from the client are displayed as a table on GUI (one table for each client), resembling illustrated in Table 3.

Table 3 - Keys and values

key	value
IP	62.73.201.5
timeout	200
Email	ekates05@student.hia.no

The values for the keys can be changed by the user. Figure 15 describes actions that are performed after the user changes keys' values.

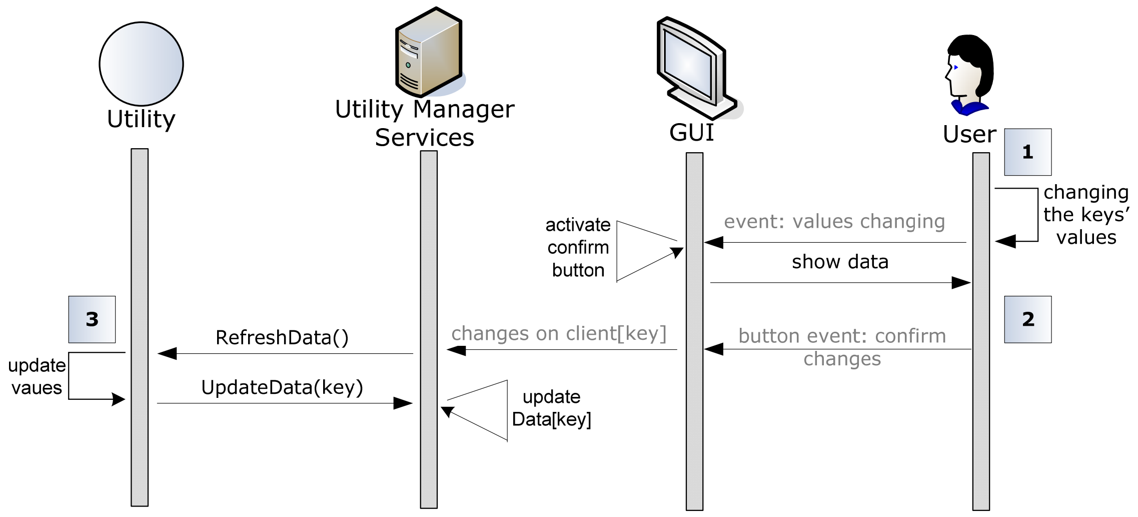


Figure 15 – Scenario 2

1. The user changes the keys' values on GUI. This action activates a confirm button ("Refresh" button on Figure 17).

2. If the changes are confirmed by the user, the server runs the RefreshData() method on the client side, and sends the updates to the client.

3. The client updates the values of the updated keys and initiate running the UpdateData() method on the server side. This method updates the data for following client stored in the array.

Scenario 3

Third scenario describes actions for starting and stopping the clients, and it is illustrated by the sequence diagram in Figure 16.

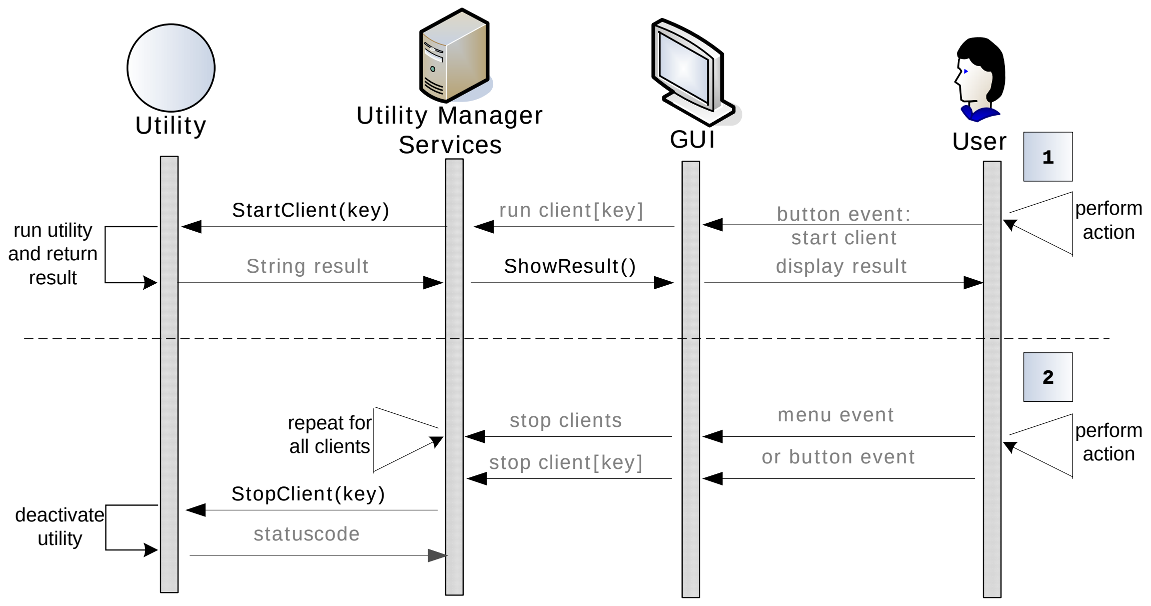


Figure 16 - Scenario 3

1. The method is started after clicking "Run" button on the server's GUI (see Figure 17). After the actions performed by the user, the server initiates starting the clients – a callback function that runs on the client side. The results are displayed on GUI.

2. After the actions performed by the user, the server initiates stopping the clients – a callback function that runs on the client side. The method can be started in two ways: after stopping the server ("Stop Server" from menu), or by clicking "Stop" button on GUI.

5.1 Server side

The functionality of the prototype's server side - the Utility Manager Services, is already described by the three sequence diagrams in the scenarios. This section presents configuration of the server. For client side to be able to see the services that is presented by the server, the server has to be configured. Example 5 illustrates the configuration file for the server.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <services>
      <service name="UtilityManagerV2.MessagingService"
behaviorConfiguration="serviceBehavior">
        <endpoint contract="UtilityManagerV2.IMessagingService"
binding="netTcpBinding" />
        <endpoint contract="IMetadataExchange"
binding="mexHttpBinding" />
        <host>
          <baseAddresses>
            <add baseAddress="net.tcp://localhost:9000"/>
            <add baseAddress="http://localhost:8000"/>
          </baseAddresses>
        </host>
      </service>
    </services>
```

Example 5 - Configuration file

In the prototype all the clients are placed on the same computer as the server, that is why the local base address is used. When the clients / utilities are placed on the other machines, the only change that has to be made is that the "localhost" needs to be changed to the server's IP. "MessagingService" here is the service's name, and UtilityManagerV2 is its namespace. The available services are declared in the IMessagingService interface and are implemented in MessagingService class. Further description about implementation of the Utility Manager can be found in Appendix 4.

5.2 Application GUI

The .NET 2.0 Framework gives developers the choice between two application models: the Windows application model and the Web application model. Applications that are designed for Windows uses the Windows application model. The classes provided by Windows Form library are designed to be used for GUI development and allows the developers to create command windows, buttons, menus, toolbars etc – in other words all the usual components of an application with a GUI. Windows Form also supports ActiveX controls. The other application model is an Active Server Pages .NET (ASP .NET) model. It can be used for publishing both XML Web services and Web forms. ASP .NET provides the set of Web Form controls that are used to generate the UI, usually in the form of HTML. Web Forms also support the developing of interactive Web pages.

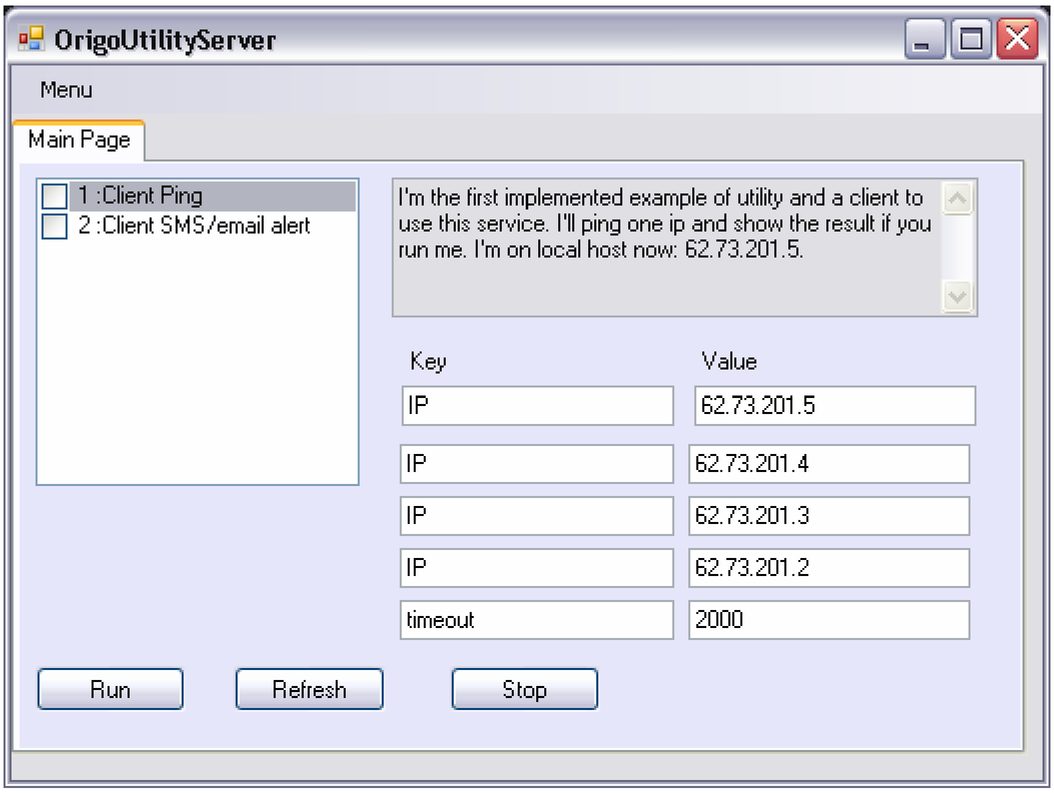
The .NET 3.0 Framework offers a new application model - WPF. This one among other things can be used for creating vector-based animations. It is in a way Microsoft's answer to the Adobe Flash (previously known as Macromedia Flash) integrated development environment (IDE). All the three application models are placed on top of the .NET

Framework and are using the same basic class libraries, virtual machine for running and runtime environment.

Among these alternatives, Windows Forms was chosen for creating the GUI for the Utility Framework Server. The decision was based on these three reasons:

- The Utility Framework Server is going to be placed on a machine with the Windows OS. This decision was made earlier and because of other reasons. The fact that applications using Windows Form can only be run in a Windows environment is not a limitation in this case.
- The accessibility of the GUI on the Web was not required in this project.
- Visual Studio simplifies the development of Windows Form based applications to a large degree, making this process less time-consuming. It also makes the learning process easier, especially for those who have created applications before but used different tools for the GUI.

Figure 17 demonstrates how the user interface of the prototype looks.



I'm the first implemented example of utility and a client to use this service. I'll ping one ip and show the result if you run me. I'm on local host now: 62.73.201.5.

Key	Value
IP	62.73.201.5
IP	62.73.201.4
IP	62.73.201.3
IP	62.73.201.2
timeout	2000

Figure 17 - GUI example

The design of the GUI was not a priority, just the functionality. The GUI allows the user to start and stop the server, activate and deactivate clients, and show the information about available utilities. The list of available clients and the short description of each of those are presented in the form. The GUI also makes it possible for the user to change the keys' values, and to send the refresh request to the client. The client will then use the new values while running. The results after running will also be presented in the GUI.

5.3 Client side

At first a common skeleton for the clients will be described, and then an example of an implemented client will be presented.

All the future clients of the Utility Framework Server shall consist of the next parts:

- *OrigoUtilityClient* class that implements the *IOrigoUtilityClient* interface. The class should contain next string variables: utilities' name, description, IP where it is placed, and a number of keys, values and results. The *OrigoUtilityClient* class should also contain the implementation of methods, which are declared in the interface; and among them a "void main" method– the method that is responsible for running the utility.
- *UtilityCallback* that implements the *IUtilityCallback* interface. Interface declares the callbacks that are available for server and the *UtilityCallback* class contains the implementation of those.
- Functional part, a class that defines utility's own functionality
- The utility also should utilize a number of automatically generated interfaces and classes from WCF (among those *ServiceClient* and *ClientChannel* interfaces).

Figure 24 in Appendix 4 introduces the class model for the Ping Client utility that was mentioned numerous times previously in the project. It will now be used as the example to describe the step-by-step rules for creating the clients.

At first, a *ServiceModel* (previously mentioned namespace from the WCF class library) reference must be added to the project. Second, while the server is up and running, the proxy for the client should be created as it described in Appendix 3. This will already give a client an insight of available services and expected callbacks. The next step is to implement the callback methods on client side. The class responsible for programming logic can vary from one utility to another, but must have some necessary variables, that were listed before. For Ping client they are: name – Client Ping; Description – some short text description about utility's functionality; IP – Ping Client runs a simple check to discover where it is placed and returns it's IP address. The same method is suggested to be implemented in all other classes. The IP address was included as the variable to indicated where the client is placed (locally with the server or not), mostly for testing purposes. Keys are the variables that the client uses for running, for Ping Client they are the IP addresses and a timeout for running a check. Values are simply the values of the keys – list of the IP-addresses and a number that defines timeout in seconds. After these values are updated by user, the client Ping will use the new values for running (the keys remain the same). The forth step is to implement the utilities functionality – in this example, actual content of the *MyPing* class, methods that perform the pings. All the parts of the utility must be placed in the same namespace.

The four steps can seem like a lot of work, but the first three are the same for all the clients, and those parts can be re-used without hesitation and with only minor changes. The part that has to be developed for each utility is the forth step – the utility's own functionality.

6 Discussion

Designing the Utility Framework server using the .NET tools was probably the best solution for this task. And the main reason is that this choice will make it easier for Origo to take over this project. But at the beginning of the project, other possibilities both concerning the main solution idea, and the choice of technologies were considered. Chapter six gives an overview of those. The chapter also contains an evaluation of the solution.

6.1 Solution idea

The Utility Framework server and the idea to relocate the scripts from the HMI onto its own server, was not the only solution proposal. Another idea was to create a Utility compiler – a translator program that would help generating the scripts automatically from the description of the script's functionality. The Utility compiler would in this way make it easier to change and to translate the scripts according to the customers' needs and requests. Figure 18 demonstrates the compiler solution.

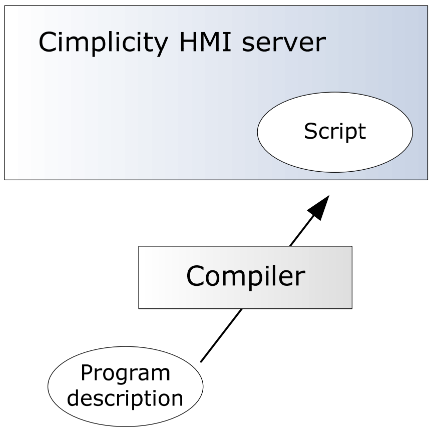


Figure 18 - Compiler solution

The strict definition of a compiler is "a computer program which converts a program written in a programming language (called the "source code") into code which a computer can execute (called "object code")" [22].

The right definition here would probably be a translator - a program which converts one source code into another [22]. In what follows, the word "compiler" is frequently used to imply "compiler, interpreter, or translator" (an interpreter is a program, which accepts a program written in a source code, converts it into some readily executable form, and performs a controlled execution); or in this sense, any one or a combination of these three definitions. Figure 19 illustrates how a compiler works [30].

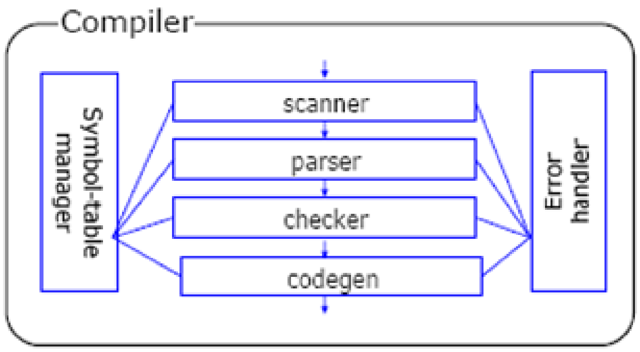


Figure 19 – Compiler

A compiler goes through different stages while translating from one language to another, or in this case, translating from a description of the functionality into the script that is suitable for the particular HMI Server. The scripts are first scanned, and then parsed, checked and after that a new code is generated. The Symbol table manager does translations of the syntax from one programming language to another (in this case it will be to Visual Basic-similar language that is used for scripting in the HMI server). The error handler removes possible errors and exceptions in the translation process. The theory around the compilers will not be described here any further. The main point of this section is to describe how the compiler would solve the challenges from the problem description.

A study of the existing utilities would have to be performed before creating the compiler. After comparing the utilities with the same or very similar functionality that were implemented in the different HMIs, main similarities and differences between them would be found. After analyzing the similarities, a skeleton for a script would be created. The skeleton is a part of the script that remains the same independently of the version of the HMI server. The differences between the utilities provide specifications for the part of the scripts that will differ from one HMI to another. The difference can not only be caused by the HMI version. Some constants have to be defined for each particular HMI server. The compiler then has to create the complete script by combining those parts. In other words, it has to create the script based on three specifications:

- The script's functionality
- The version of the HMI server
- Some constants that are specified for the particular HMI.

To explain possible similarities and differences in scripts, an example will be used. As for the example, the Network Monitor (also referred as Client Ping) utility will be used. From one HMI server to another, the main functionality remains the same: to ping the PLCs on the network. It most likely means that methods, received parameters and return values will remain the same as well. They will create the "skeleton" of the script that stays unchanged without regard to the HMI. On the other hand, the list of the network addresses that are going to be checked varies from one HMI system to another. This list is the constant that is specifically created for the particular HMI. And if the utility uses some built-in functions from the HMI server, those could also vary from one version to another. In some cases, there could be some changes in the syntax of the utility. These differences are the examples of changes that have to be made according to parameters determined by the HMI version. The compiler should take it all into consideration when generating the scripts. Figure 20 illustrates preliminary parameters for generating the scripts by the compiler.

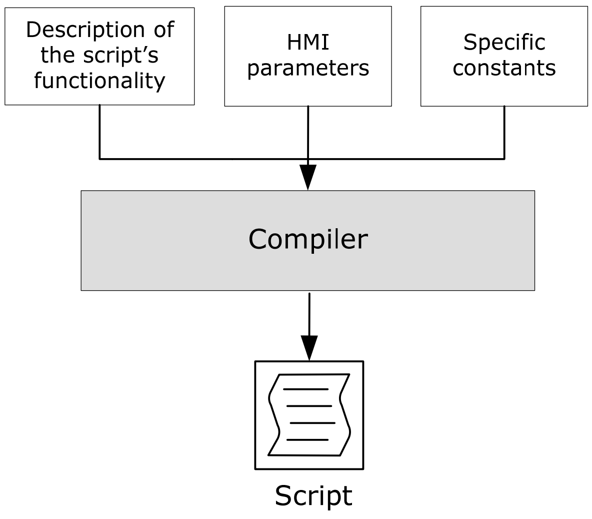


Figure 20 - Generating scripts

Additionally the description of the script's functionality should be standardized in a way that allows the compiler to understand it. Creating the standard language for such a description and describing the programs by using it is a time-consuming process, but it only has to be done once – or once for each particular program. As to the bit that is changed according to the HMI version, the new description has to be created for each new version of the HMI, if the compiler is going to be used for generating scripts for it. The specific constants differ for each script and that is why they have to be defined for each new script.

If created, the compiler will simplify the process of rewriting the scripts over and over. The developers would not have to rewrite the same scripts; they only have to specify to the compiler the part that differs in the script (assuming that they have already created the standard language and used it for describing the skeletons of the existing utilities). The compiler will automate the procedure of adjusting the scripts to the HMI and then solve the first challenge, reusability. The scripts are still being dependent of the HMI, but the problem of dependency is limited in this case. Now the only problem is that the developers have to find and describe the specifications for each new HMI version. Assuming that the number of different kinds of HMI servers that Origo is using today is limited, the problem of dependency nearly disappears afterwards.

About the error handling challenge, the problem will not disappear completely. In contrast to the Utility Framework Server, where the scripts are relocated from the HMI, in this case the scripts are still running directly on the HMI (as shown in Figure 4) and that is why they still can cause the crash of it. The chance that this happens is getting smaller. When the scripts are written manually, the errors are located in scripts and can also be different for each one of them. This makes it difficult to find and to prevent them. When the compiler is used to generate the scripts, the appearance of the errors is centralized in the compiler. The errors can still appear, but after the error has been located once, it can be corrected directly on the compiler. The errors in the scripts that have been caused by this error on the compiler are the same, and they can be fixed by re-compiling the utilities. The error handling process becomes more automated and less time-consuming. Assuming the compiler would be well tested before use, and the new generated scripts will be re-tested subsequently, the risk that a critical error occurs is minimal. The conclusion is that the error handling will benefit from the compiler solution, along with the other two challenges. Table 4 was set up to summarize which challenges the different solutions solve.

Table 4 - Challenges and solutions ideas

Challenges / Solutions	Reusability	Dependency	Error handling
Scripting on HMI	–	–	–
Utility Compiler	+	+	+/–
Utility Framework Server	+	+	+

Even thought the compiler also shows the possibility of improvement in all the three challenges, the Utility Server Framework was chosen as the solution idea. The time-consuming matter of the first stage in creating the Utility compiler – to gather the differences and similarities with the scripts, was one of the reasons the Utility Framework became the chosen solution in the end. And when Origo only agreed to provide a very limited number of examples of the existing scripts, it was not enough information for creating the Utility compiler. To conclude, the Utility Server Framework is easier in the development, better suits the challenges and also is the solution that was preferred by the external supervisor.

6.2 Choosing the technology

A number of technologies are available for developing distributed applications. Already at the early stage of the work it was decided to use the .NET Framework, but several other solutions are available, and these will be discussed later in this section. At first, the choice of WCF as the tool for building the distributed system is going to be explained.

In fact, .NET Remoting was considered as the main option at the beginning of the project. The .NET Remoting is a stable and established technology, in distinction to the relatively new WCF. It would not be a problem to find information about it, as it has been well documented by numerously sources by now. The learning curve would probably be easier as well, as it would be easier to get help from the teaching supervisors. Despite these arguments, the WCF was chosen at last. The main motivation for choosing WCF in preference to .NET Remoting was the fact that it is a relatively new technology that is exiting to learn more about. The external supervisor of the project, Origo, has also showed a desire for a closer look and research on this new technology, so they could later debate on using it and eventually get an easier start by using this report.

During the work this choice did not caused any regrets; even though it caused some challenges at the beginning. It was also discovered that choosing the .NET Remoting would probably have create some problems later in the project, the problems that were avoided by using WCF. One of the examples is the implementation of callbacks – the methods where clients have to act like a server, and a server acts like a client. Callbacks became one of the central mechanisms in the final prototype; the number of callback-methods is almost equal to the number of services provided by the server. One of the examples of callback is the decision that the clients would be the first to initiate the connection with the server. WCF provides all the rules and methods for implementing the callbacks. In fact, implementing the callbacks is not much harder than implementing usual services (or methods provided by the server for use by the clients). But how could it be done using .NET Remoting? There a developer had to attach his client to an instance of a remotely hosted *MarshalByRefObject* (a class from the .NET 2.0 class library, that enables access to objects across application domain boundaries in applications that uses Remoting) and simply subscribe to events published by that object. This model is no longer available in WCF and there are reasons for it. This architecture is known to be unstable and difficult to use.

WCF also has other benefits. In order to understand them, the previous Microsoft products that could be used for building the distributed systems are shortly presented here. There are numerous other tools and each of them was specialized for its own purposes. For example, to build basic interoperable Web services, the best choice was ASP.NET Web services, more commonly referred to as ASMX. To connect two .NET Framework-based applications, .NET Remoting sometimes was the right approach. If an application required distributed transactions and other more advanced services, its creator was likely to use Enterprise Services, the .NET Framework's successor to COM+. To exploit the latest Web services specifications, such as WS-Addressing and WS-Security, a developer could build applications that used Web Services Enhancements (WSE), Microsoft's initial implementation of these emerging specifications. And to create queued, message-based applications, a Windows-based developer would use Microsoft Message Queuing (MSMQ) [14].

The WCF was made to gather all these earlier Microsoft technologies (or support for them) at one place. Rather than forcing developers to choose one of several possibilities, WCF lets them create distributed applications that address all of the problems solved by the technologies it subsumes. While Microsoft will still support these earlier technologies, most new applications that would previously have used any of them will instead be built on WCF.
Table 5 is brought from article [14] and illustrates the relation between the technologies and use purposes.

Table 5 - WCF and previous Microsoft's technologies

Technology / Purpose	ASMX	.NET Remoting	Enterprise Services	WSE	MSMQ	WCF
Interoperable Web Services	Yes					Yes
.NET – .NET Communication		Yes				Yes
Distributed Transactions, etc.			Yes			Yes
Support for WS-* Specifications				Yes		Yes
Queued Messaging					Yes	Yes

WCF made the life of Windows developers easier by unifying disparate technologies. The other benefit lies in the fact that the WCF's basic communication mechanism is SOAP. Using SOAP standard for communication allows WCF applications to communicate with other software running on other platforms, at least with other applications that support standard Web services.

6.3 Similar technologies from other vendors

Microsoft is of course not the only vendor supplying software for creating client - server based systems, although it got the main focus in this project. In fact, many others vendors exist and they provided almost a countless number of different technologies through the last years. Some of the technologies have expired; some projects were never finished or never became popular among developers, but there are still many of them that exist and work today. This section is dedicated to giving a general view on some of them.

Among the other vendors, Sun Microsystems is probably the most familiar. They developed popular technologies for performing remote procedure calls like Java Remote Method Invocation API or simply Java RMI. RPC (remote procedure call) is a protocol that is used for constructing distributed, client-server based applications. It is based on extending the concept of conventional or local procedure calling, and allows a computer program running on one host to call a procedure in a different address space, or host, as the calling procedure without the programmer needing to explicitly code for this. The two processes may be on the same system, or they may be on different systems with a network connecting them. Java RMI allowed in other words creating a distributed system – a whole system of which the components can be placed on different machines on a network, and have to be reached remotely when the application is executed. One of the drawbacks of Java RMI is that it only supports making calls from one java virtual machine (JVM) to another, resulting in that both server and client side applications have to be created in Java and therefore use JVM for running.

For supporting code running in a non-JVM framework, Sun Microsystems joined Object Management Group (OMG), where Apple and Hewlett-Packard are amongst the other members, and they created a Common Object Request Broker Architecture. CORBA is a standard that allows the components of the distributed system to be written in multiple computer languages. Both Java RMI and CORBA try to map object oriented design onto the network. The main problem with CORBA became its largeness, or its 'everything to everyone' nature. CORBA tried to solve all the common problems in distributed systems, but didn't give the description of solving the concrete ones. The other problem was already previously mentioned in this report. CORBA hides locations of the objects and makes no difference in treating the local or remote ones. As time showed, it is not always necessary and not always the most efficient architecture for designing a system; and it is definitely not the simplest one.

Web services are another attempt to implement RPC between platforms. Using Web services a .NET client can call a remote procedure implemented in Java on a UNIX server (and vice versa). The other advantage of web services is simplicity and standardization. The most of the current technologies for creating the web services use previously mentioned SOAP and WSDL standards for message exchanging. XML-RPC is also widely used. XML-RPC is a RPC protocol that uses XML to encode its calls and HTTP as a transport mechanism.

There are a number of frameworks are available today, and most of them are written in Java. The Java Enterprise Edition, Sun Open Net Environment, Apache Axis and gSoap are some examples of the frameworks and architectures that can be used for Web service development and deployment. All the 4 mentioned uses SOAP and WSDL standards and that is why they can create the services that are available for the clients written on the other languages than the services themselves. The first two were created in Java; Apache Axis is created partially in Java and partially in C++, and the last one gSOAP is written only in C++. Apache Axis and gSoap are the open source (which means that both code and application are available for downloading) frameworks for developing only web services.

The Java Enterprise Edition and the Sun Open Net Environment frameworks are much bigger systems and are designed for developing complete enterprise solutions. They are among of the major alternatives to the .NET Framework. JEE SDK is free for downloading, while Sun ONE is available on CD/DVD for purchasing.

There are also available several Integrated Development Environments (IDEs) that support web services (and also of course other programs) written on java, among those are IntelliJ IDEA, Sun Java Studio Creator/Enterprise, NetBeans and Eclipse. The first two require the purchase of license; while NetBeans and Eclipse are the most powerful open source extensible platforms.

As for WCF, Service Component Architecture (SCA) from vendors of Java EE technology (as IBM, BEA, Oracle, SAP, IONA, and others) is a new alternative. SCA is a set of specifications which describe a model for building applications and systems using principles of SOA. SCA allows developing the systems using the components written not only with different programming languages, but also using frameworks and environments commonly used with those languages. For developing distributed systems, SCA supports a number of communication technologies, for example web services, messaging systems and RPC. SCA looks much alike WCF, but there are also some differences. Table 6 shortly summarizes some of the similarities and the differences of these two [23].

Table 6 - SCA and WCF

Features	**SCA**	**WCF**
Dividing business logic and communication methods	yes	yes
Easily defined service interfaces	yes	yes
Wrapping different communication information into *bindings*	yes	yes
Relying on the WS-* specifications	yes	yes
Supporting one-way calls, asynchronous calls, two-way interactions and message notification	yes	yes
Supporting session management for calls	yes	yes
Supporting programming languages	A number of languages, including C++, Java, COBOL, and PHP as well as XML, BPEL, and XSLT	A number of languages too, but all those must be build on the .NET

		Framework's CLR
Using framework	Can be build on top of the various systems (for example, Enterprise Java Bean (EJB) and Spring)	.NET Framework
Using of Service Data Objects (SDO) as a standard approach for passing data	yes	no

SCA uses SDO, which is the only industry standard for data access in SOA; WCF doesn't dictate what approach should be used for data. Main advantages of SCA are that it is less technology dependent and that it is available for all existing Java platform technologies and C++. The main challenges will probably be to become relevant for a large number of users. Without Microsoft's support, SCA does not have the advantage that WCF has – WCF is easily available for developers that are working on Windows platforms, because it is included in Windows Vista operating system. More information about SCA and comparing it to WCF is available from [24]. Nevertheless, there is no doubt that a competition between .NET and JEE is good for both of them and at the end for the developers.

6.4 Evaluation of the solution

This project's final solution consists of two parts: the ideal system and the prototype solution. The ideal system was designed to implement all the requirements, and this was described in the report. This ideal system is created to be reliable, reusable and scalable solution, according to the COP and SOA principles. The whole system is constructed by the number of components. This makes it easier to update some features of the system, without changing the whole implementation. For example, for changing the GUI, changes will only concern the GUI component as long as the GUI's interface remains unchanged. Using the WCF for communication with the utilities makes it easy to add and remove the utilities, as long at these are created by following the rules.

Nevertheless, there are some issues that have not been discussed before. One of them is security. Security issues with communication in the developed solution were not the part of this project. One of the reasons for it is that the future system will be placed on a closed network (that is physically locked in a room) and will not be exposed to internet. In view of the IO principles, there is a possibility that the system will be transported onshore in the future. It will then have to become more open, and the services will have to be exposed to the utilities /clients or probably communicate with HMI through the internet connection. In these new conditions, the security issues with the system have to be reviewed, and additionally implemented.

The implemented prototype has shown some new issues.

To simplify the implementation, all the messages between the server and the utility is now sending as strings. It could be suggested considering looking at the XML format as the other possibility. Using the XML can probably further simplify creating the utilities, by using the same description format. Example 6 shows how the Ping Client can be described by using the XML format. In the current implementation all the parts of the description are combined in one string and divided with sign ";".

```
<name>Ping Client</name>
<desc>I am the first implemented client...</desc>
<IP>62.73.201.5</IP>
<key>IP</key>
<value>62.73.201.4</value>
…
```

Example 6 - XML client description

Thread safety is the other issue. As it is implemented in a current prototype version, an array where the server stores the data from all clients is declared as global static variable. If all the clients were using this array for writing and accessing the information, it could cause the risk of deadlocks. In the current prototype implementation, the array is not being accessed directly by clients, but the server itself performs operations for filling it with data. This has not caused any problems during the testing that was performed as a part of developing process, but can probably be a hidden danger for the future development, when the number of clients greatly increases. The other issue is that this method can possibly bring delays, caused by the fact that the server has to perform numerous write operations, when it once started and being accessed by, for example, 100 clients. This should be therefore tested. In case of problems, the other solution could be to allow clients to access the static array, but to set service's *ConcurrencyMode* to *Multiple* (it is an option in WCF), and to lock the access for the array for all others while one client is performing some operations with it. It is probable that this solution can also cause the delays, and therefore it is important to test how it works with multiple clients. The other solution that could probably be considered as an option, in case of a large number of clients, is to use the database solution (instead of a static array).

Several parts of this task were challenging at the beginning, it also was one of the reasons that made this particular task interesting. The .NET Framework, C# programming language and the standards and technologies of software in industry were the new fields of study for the author of this report, and because of this the research stage of the project was time-consuming and sometimes even confusing; but the final results of this project are considered satisfying.

7 Conclusion and further work

Chapter seven consists of two parts: the first one presents the summation of the project and the second one gives suggestions for further development.

7.1 Conclusion

This report describes the problems and proposes solutions for customizing control and automation systems by using the HMI servers. The customizing of HMI servers (like one called Cimplicity from the vendor GE Fanuc) is done today at Origo Engineering AS by scripting directly on the HMI. The scripting approach causes challenges with reusability, dependency and error handling in the scripts. A solution that can simplify the customising process at Origo was proposed, developed and presented as the Utility Framework Server as a part of this thesis. By implementing the prototype and utility example of the ideal solution, the main principles of the Utility Framework Server were illustrated in practice.

The challenges that Origo experiences today appeared to be common for large modern software systems. Software developers have tried to find a way to create reliable and reusable systems for quite some time now, and an amount of techniques and programming architectures have been created during this time. Some of them have claimed to solve similar problems to those listed in the problem description. Especially component oriented programming and service oriented architecture were considered in this report. They were chosen as the solution techniques for developing the Utility Framework Server.

The two main ideas considered for the solution were the Utility Framework Server and the Utility Compiler. After research and analysis, the Utility Framework Server was chosen as the best solution for the current challenges. Earlier in the report it is shown exactly how the use of the Utility Framework Server will improve the current situation, and how it will solve the existing problems. Using the Utility Framework Server can benefit the developers directly, and the customers indirectly since the developing process will take less time and also become less expensive.

The implementation of the prototype of the ideal solution has been described. Even though this prototype has a limited functionality, it helped to discover and put the power of WCF to use. During the work with the prototype, some new challenges were discovered (deadlocks and delays in the multiple thread handling). Solution methods for these challenges were further suggested. The prototype was implemented to simplify the further development process of the Utility Framework Server and give the developers a good start, which is my contribution with this project.

7.2 Further work

The fully-functional Utility Framework Server solution is a final goal for future work. The work can be divided into following next stages:

- Implement the remaining components of the Utility Framework Server
- Implement an amount of utilities with different functionality
- Test the concurrency handling and thread safety among the utilities
- Test the implementation against the simulator of the HMI server, also by using different HMIs
- Combine the hardware and software part, creating the whole ready-to-use solution.
- Place it on the same network with the HMI server and the PLCs.
- Test the implementation in the "real" working environment.
- Document the solution (the final system description and a user manual)

Abbreviations

API	Application programming interface
ASMX	ASP.NET Web services
ASP	Active Server Pages
CLR	Common Language Runtime
COM	Component Object Model
COP	Component Oriented Programming
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DLL	Dynamically linked library
DSL	Domain Specific Language
EXE	stays for “executable”, in automatically running files
FTP	File Transfer Protocol
GC	Garbage Collection (in computer science)
GUI	Graphical User Interface
ICT	Information and Communications Technology
IDE	Integrated Development Environment
IPC	Inter process communication over named pipes
J2EE, JEE	Java (2) Platform, Enterprise Edition
Java RMI	Java Remote Method Invocation API
JVM	Java Virtual Machine
HMI	Human Machine Interface
HTTP	Hypertext Transfer Protocol
MSMQ	Microsoft Message Queuing
OLE	Object Linking and Embedding
OMG	Object Management Group
OPC	OLE for Process Control
RPC	Remote procedure call
P2P	Peer-to-peer
PLC	Programmable Logical Controller
SDK	Software Development Kit
SMTP	Simple Mail Transfer Protocol
SOA	Service Oriented Architecture
SOAP	originally stood for Simple Object Access Protocol, and lately also Service Oriented Architecture Protocol

Sun ONE	Sun Open Net Environment
TCP	Transmission Control Protocol
UI	User Interface
URL	Uniform Resource Locator
WCF	Windows communication foundation
WCS	Windows CardSpace
WPF	Windows Presentation Foundation
WSE	Web Services Enhancements
WWF	Windows Workflow Foundation
XML	Extensible Markup Language

References

[1] Stortinget: white paper nr.38 (2003 – 2004); "About the Petroleum activity". The Royal Ministry of Oil and Energy, May, 2004.

[2] An article about user interface from Wikipedia http://en.wikipedia.org/wiki/User_interface (26.05.2007)

[3] "OPC (OLE for Process Control) Specification and its Developments" (by Li Zheng', Hiroyuki Nakagawa, OPC Council; Japan, Yamatake Corporation, Volume 2, Aug. 2002). Available from http://ieeexplore.ieee.org

[4] "Standards-Based Approach Integrates Utility Applications" by D. Becker, H. Falk, J. Gillerman, S. Mauser, R.Podmor, L. Schneberger (IEEE Computer Applicarions in Power, Vo1.13, pp.13-20, Oct. 2000). Available from http://ieeexplore.ieee.org

[5] "OPC the de facto standard for real time communication" by Renee Pattle, Jurgen Ramisch (Parallel and Distributed Real-Time Systems, 1997). Available from http://ieeexplore.ieee.org

[6] "OLE for Process Control (OPC) for New Industrial Automation Systems" by Yoh Shimanuki, Japan (Systems, Man, and Cybernetics, 1999). Available from http://ieeexplore.ieee.org

[7] "The Distributed Data Integration and Performance Evaluation in Power Automation System" by Xu Hong, Wang JianHua, Zheng Shi Quan from Institute of Electrical Engineering (Parallel and Distributed Computing, Applications and Technologies, 2003). Available from http://ieeexplore.ieee.org

[8] "Human Machine Interface using OPC (OLE for process control)" by M.Raafay Anwar, Osama Anwar, Syed Faisal Shamim and Ahmer Ali Zahid (Dec. 2004). Available from http://ieeexplore.ieee.org

[9] "Complete information guide about Cimplicity HMI", GE Fanuc Automation 2006, available from http://www.gefanuc.com/Downloads/en/cimplicityhmi_databook.pdf

[10] Book: Programming .NET Components, Second Edition by Juval Lowy, O'Reilly (July 2005)

[11] Book: C# 2.0 the complete reference, Second Edition by Herbert Schildt, (2006, The McGraw-Hill Companies)

[12] An article about Service Oriented Architecture from Wikipedia, http://en.wikipedia.org/wiki/Service-oriented_architecture (26.05.2007)

[13] The figure is copied from the diagram "Stack .NET 3.0 in Vista" by Thomas Lee (under GNU Free documentation license), found in spring 2007 at: http://en.wikipedia.org/wiki/Windows_Communication_Foundation

[14] Introducing Indigo: An Early Look by David Chappell, Chappell & Associates February 2005, available from http://msdn2.microsoft.com/en-us/library/aa480188.aspx

[15] W3C "Web Services Description Language (WSDL) 1.1", 2001, available from http://www.w3.org/TR/wsdl

[16] W3C "UDDI Version 2.04 API" (Published Specification, 2002), available from http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

[17] W3C "SOAP Version 1.2 Part 1: Messaging Framework", 2007 , available from http://www.w3.org/TR/soap12-part1/

[18] "Second-Generation (WS-*) Web Services" by Thomas Erl (2004), available from http://www.soaspecs.com/page2.asp

[19] "Introduction to Building Windows Communication Foundation Services" by Clemens Vasters (Newtelligence AG, September 2005), available from http://msdn2.microsoft.com/en-us/library/aa480190.aspx

[20] Book: Programming WCF services by Juwal Lowy First Edition (February 2007), O'Reilly

[21] Web Services Glossary (W3C Working Group Note, February 2004), available from http://www.w3.org/TR/ws-gloss/

[22] "An Elementary Discussion of Compiler/Interpreter Writing", R. L. GLASS (Boeing Company, 1969 Seattle, Washington) available from http://portal.acm.org/

[23] "Foundations for Service-Oriented Applications: Comparing WCF and SCA" by David Chapel, December 2005. Available from http://www.davidchappell.com/HTML_email/Opinari_No15_12_05.html

[24] "SCA relationship with Windows Communication Framework" by Mike Edwards (IBM, Mar 20, 2007) available from http://www.osoa.org/display/Main/SCA+relationship+with+Windows+Communication+Framework

[25] "Writing a "Hello, World" service and client with WCF" sample, found in February 2007 at http://wcf.netfx3.com/content/BuildingHelloWorld.aspx

[26] "Programming Indigo: The Programming Model" by David Pallmann, 2005 available from http://msdn2.microsoft.com/en-us/library/aa480201.aspx or from book: Programming Indigo Beta Ed edition by David Pallmann (Microsoft Press, 2005)

[27] Picture from http://www.photojojo.com/content/wp-content/uploads/2006/06/photo_lego_blocks.jpg (26.05.2007)

[28] MatriconOPC's official website http://www.matrikonopc.com/index.aspx (26.05.2007)

[29] The figure is inspired by the model "Web Services" by H. Voormann (under GNU Free documentation license), found in spring 2007 at: http://en.wikipedia.org/wiki/Image:Webservices.png

[30] The figure "Compiler phases" is copied from the lecture "Abstract Syntax" given by Andreas Prinz in subject IKT408 "Software Engineering and Compiler Design" (spring semester 2006). The lecture is available from Fronter-system at HiA.

Appendix 1

Beginning with WCF and "Hello World!" example

The first natural step in learning WCF was to implement HelloWorld example for en service/client using WCF technology. This went relatively nice with minor complications due to some difficulties with finding propitiate information about it.

First problem was to install all the needed components. 3.0 Net Framework is not the part of the latest Visual Studio (2005), and in order to all libraries come up and will be usable from it, the additional components has to be installed:

- .NET Framework 3.0 SDK

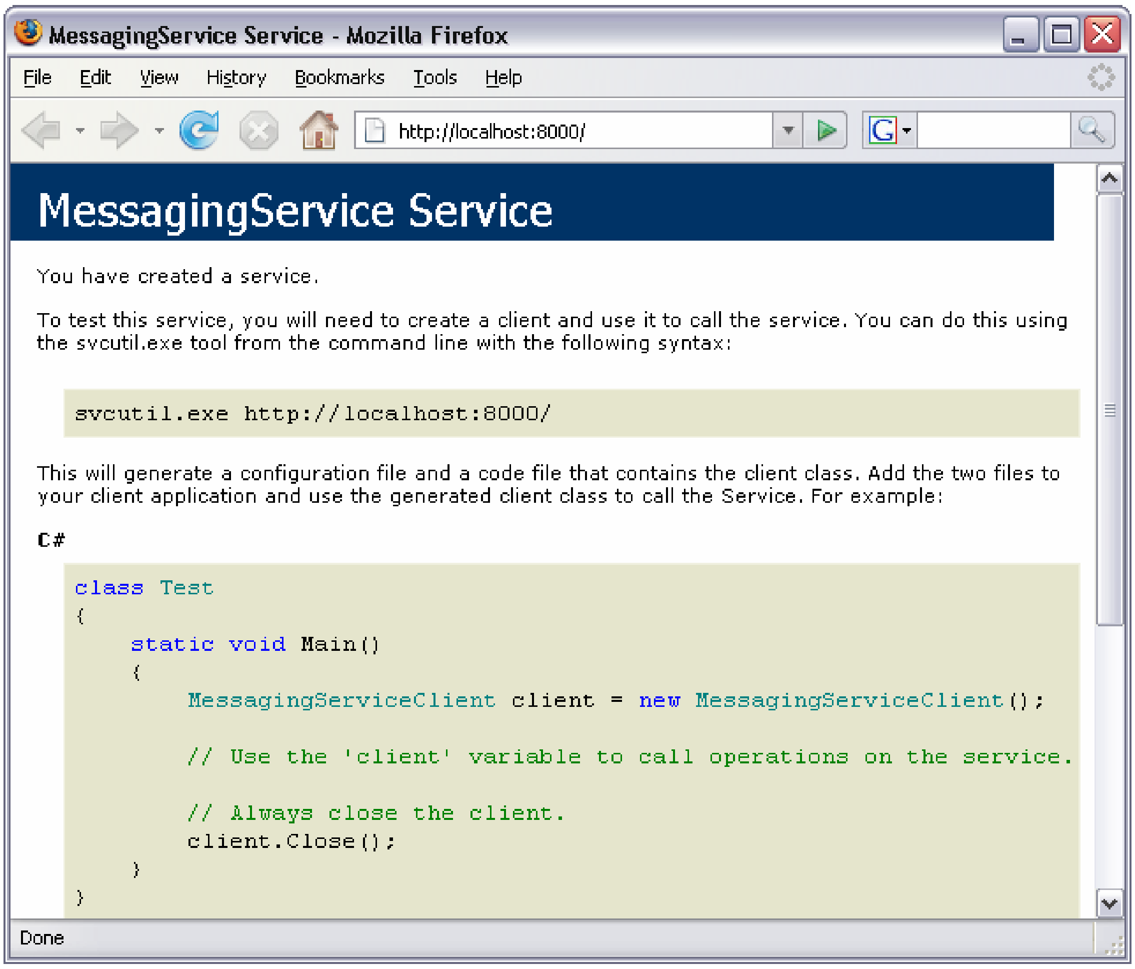
This is the part of Windows Vista, but since Windows XP is used in this project, the framework has to be installed additionally. This is the only part that needs to be installed in case if Visual Studio will not be used in a development.

- WCF extension for Visual Studio

As it was mentioned before, WCF is not a standard part of Visual Studio 2005, as the framework itself came long before the final version of .NET 3.0 Framework. This extension includes new libraries to Visual Studio, that are needed for running the WCF applications from Visual Studio; or else one gets error messages because of the lacking class libraries.

It also was some problems with HelloWorld tutorial that was found at [25] and used as an example. The problems were caused by some writing errors in the code. But after some testing and failing, and some changes in code, it worked.

Regardless to what communication protocol a developer is planning to use later, it can be practical to create a HTTP base address at first. Hosting the service in a HTTP mode simplifies getting metadata for automatic generating the configuration file. There still some problems with this in WCF when for example the TCP protocol is used, even thought it should also be possible. It is therefore usefull to implement a host in a HTTP mode at the beginning. This HTTP adress can be deleted or changed to TCP afterwards. When the service is up and running, the HTTP-hosted service will be available from any web browser on the specifyed baseaddress; and the developer will be presented with the service's homepage as it is shown in Figure 21. The description on what should be done further and an example on the service's client are presented there.


MessagingService Service - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8000/
Done

MessagingService Service

You have created a service.

To test this service, you will need to create a client and use it to call the service. You can do this using the svcutil.exe tool from the command line with the following syntax:

```
svcutil.exe http://localhost:8000/
```

This will generate a configuration file and a code file that contains the client class. Add the two files to your client application and use the generated client class to call the Service. For example:

C#

```
class Test
{
    static void Main()
    {
        MessagingServiceClient client = new MessagingServiceClient();

        // Use the 'client' variable to call operations on the service.

        // Always close the client.
        client.Close();
    }
}
```

Figure 21 - Service's homepage

More information about programming WCF can be found in [26].

Appendix 2

Bindings and theirs reliability in WCF

Bindings

WCF offers 9 standard bindings:

- Basic Binding

Implemented by BasicHttpBinding class, it is the binding for classic web service; it inherits principles from earlier ASP's web services (ASMX). Using this binding allows old clients to use new web services.

- TCP binding

Offered by NetTcpBinding, it allows machines to communicate through internet by using TCP protocol. TCP binding supports optional reliability, transactions and security. Using TCP binding implies that both communicating sides - clients and services have to use WCF.

- Peer network binding

It is implemented by NetPeerTcpBinding class and uses peer network for message exchanging.

- IPC binding

Offered by NetNamePipeBinding class, it is allows the service and the client to communicate when they are placed on the same machine. Since this communication method doesn't allow connections from the outside, it makes it the most secure one.

- Web Service (WS) binding

Offered by WSHttpBinding, it uses HTTP and HTTPS for transport. It is a more secure version of web services, provided by basic binding.

- Federated WS binding

Implemented by WSFederationHttpBinding class, this binding offers support for federated security.

- Duplex WS binding

Offered by WSDualHttpBinding class, it allows bidirectional communication between the service and clients, permitting DualCallback – a callback (a call that initiated by client) that opens a connection channel that goes both ways from client to service and back.

- MSMQ

Implemented by NetMsmqBinding class, this binding uses MSMQ for transport. MSMQ allows disconnected work between service and clients, by querying and later delivering of messages.

- MSMQ integration binding

Offered by MsmqIntegrationBinding class, it converts WCF messages to and from MSMQ messages [20].

Bindings Reliability

WCF like other service-oriented technologies separates between transport reliability and message reliability. Transport reliability offers point-to-point guaranteed delivery at the network packet level, and the order of packets. It also guarantees that each message will be delivered just once. Message reliability guarantees the message delivery at higher level, regardless to how many packages are required for this. Message reliability provides end-to-end guaranteed delivery and order of messages. Message reliability is based on an industry standard for reliable message-based communication that maintains a session at the transport level [20].

In WCF, reliability is controlled and configured in the binding. AS it was mention before, WCF allows using different communication methods, and this is also done defined by bindings. Table 7 is brought from [20] and gives an overview of the possible bindings and there reliability and order reliability.

Table 7 - WCF bindings

Name	Supports reliability	Default reliability	Supports ordered	Default ordered
BasicHttpBinding	No	N/A	No	N/A
NetTcpBinding	Yes	Off	Yes	On
NetPeerTcpBinding	No	N/A	No	N/A
NetNamedPipeBinding	No	N/A	Yes	N/A
WSHttpBinding	Yes	Off	Yes	On
WSFederationHttpBinding	Yes	Off	Yes	On
WSDualHttpBinding	Yes	Off	Yes	On
NetMsmqBinding	No	N/A	No	N/A
MsmqIntegrationBinding	No	N/A	No	N/A

Reliability is not supported by the BasicHttpBinding, NetPeerTcpBinding, and two last MSMQ - bindings. The reason for that is that BasicHttpBinding is oriented towards ASMX service world, which does not have reliability. NetPeerTcpBinding is designed for broadcast scenarious. MSMQ bindings are for disconnected calls, where no transport session is possible.

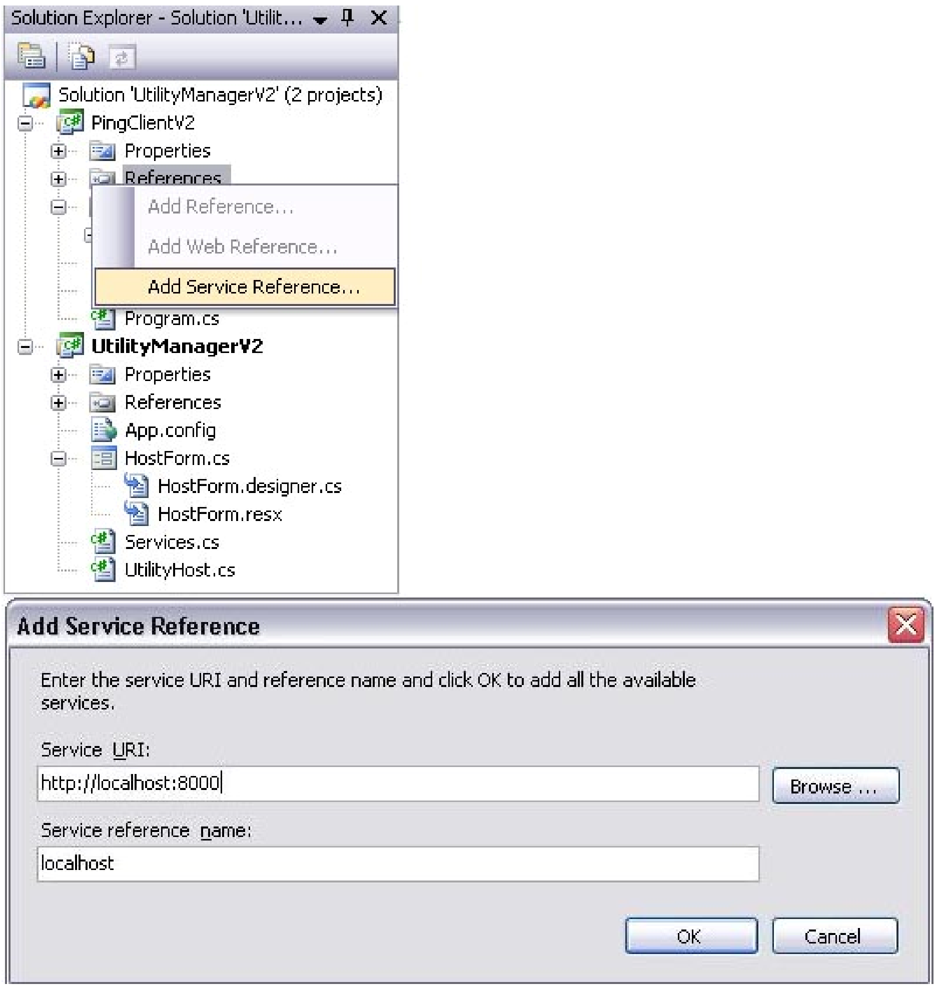
Different bindings also use different encoding methods for transport. Bindings that use HTTP or HTTPS transport (as those that are offered by BasicHttpBinding, WSHttpBinding, WSFederationHttpBinding and WSDualHttpBinding classes) use Text /MTOM encoding. MTOM - Message Transmission Optimization Mechanism, is a mechanism for transferring large binary attachments as non-text messages (for example, pictures) as raw bytes, allowing for smaller messages. The remainder bindings use binary encoding.

More information about bindings and a method that describes how to choose the binding for the implementation can be found in [20].

Appendix 3

Generating the Proxy in Visual Studio

The proxy for client can be generated automatically in Visual Studio. One has to launch the service and then select Add service reference, as showed on the next figure. URL for the service is the address for service that one defined before and listed in configuration file for Server application. Figure 22 illustrates it.



Add Service Reference

Enter the service URI and reference name and click OK to add all the available services.

Service URI: http://localhost:8000

Service reference name: localhost

Figure 22 - Proxy in Visual Studio

Visual Studio then generates a new folder Service References with two files in it:
```
localhost.map
localhost.cs
```

The first one is a configuration file similar to one that is on server side, the other one contains information about all available services. Example 7 and Example 8 illustrate these two files that have been generated for the MessagingService and MessagingServiceCallback that are avilable on the localhost.

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceReference>
      <ProxyGenerationParameters
            ServiceReferenceUri="http://localhost:8000/"
            Name="localhost"
            NotifyPropertyChange="False"
            UseObservableCollection="False">
      </ProxyGenerationParameters>
      <EndPoints>
            <EndPoint
                  Address="net.tcp://localhost:9000/"

      BindingConfiguration="NetTcpBinding_IMessagingService1"

      Contract="PingClientV2.localhost.IMessagingService"
                  >
            </EndPoint>
      </EndPoints>
</ServiceReference>
```

Example 7 - localhost.map

The base address that was specified for the server is the same as in Example 1. The address has two endpoinds, and they both are also specifyed in the configuration file on a client side.

In this example one service RegisterClient() and one callback StartClient() are avilable. First method returns int (a whole numerical value), the other method declared void and does not return anything, but takes an int variable as parameter. All this specifyed in the code illustrated in Example 8. The comments for the code are also presented.

```
namespace PingClientV2.localhost //namespace
{
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
"3.0.0.0")] //using the 3.0 for running

[System.ServiceModel.ServiceContractAttribute(ConfigurationName="PingCl
ientV2.localhost.IMessagingService",
CallbackContract=typeof(PingClientV2.localhost.IMessagingServiceCallbac
k))]
//service’s interface
public interface IMessagingService
    {
//the list of available services
[System.ServiceModel.OperationContractAttribute(Action="http://tempuri.
org/IMessagingService/RegisterClient",
ReplyAction="http://tempuri.org/IMessagingService/RegisterClientRespons
e")]
        int RegisterClient();
    }

[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
"3.0.0.0")]
    public interface IMessagingServiceCallback
    {
//... the description of avilable callbacks
        [System.ServiceModel.OperationContractAttribute(IsOneWay=true,
Action="http://tempuri.org/IMessagingService/StartClient")]
        void StartClient(int key);
    }
//generating the service channel
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceModel",
"3.0.0.0")]
    public interface IMessagingServiceChannel :
PingClientV2.localhost.IMessagingService,
System.ServiceModel.IClientChannel
    {
    }
// further services the endpoint address and binding are presented
        public
MessagingServiceClient(System.ServiceModel.InstanceContext
callbackInstance, System.ServiceModel.Channels.Binding binding,
System.ServiceModel.EndpointAddress remoteAddress) :
                base(callbackInstance, binding, remoteAddress)
        {
        }
// services contract is presented
        public int RegisterClient()
        {
            return base.Channel.RegisterClient();
        }

}
```

Example 8 - localhost.cs

Without Visual Studio, it is possible to generate proxy with the SvcUtil.exe build-in command-line utility from Microsoft .NET Framework SDK, where the endpoint's address has to be provided. Proxy.cs is the name for generated file (will be output.cs by default). Example 9 illustrates it.

```
SvcUtil
http://localhost/UtilityManagerV2/MessagingService.cvs/out:Proxy.cs
```

Example 9 - Proxy with SvcUtil.exe

Appendix 4

Appendix 4 presents two class diagrams and some examples from the code with comments, to give a better overview on the practical part of this project. Class diagrams relates mostly to the Utility Framework Server design (chapter 4). All the code examples are from the prototype solution (chapter 5).
Figure 23 presents the class diagram for the server side of the solution.

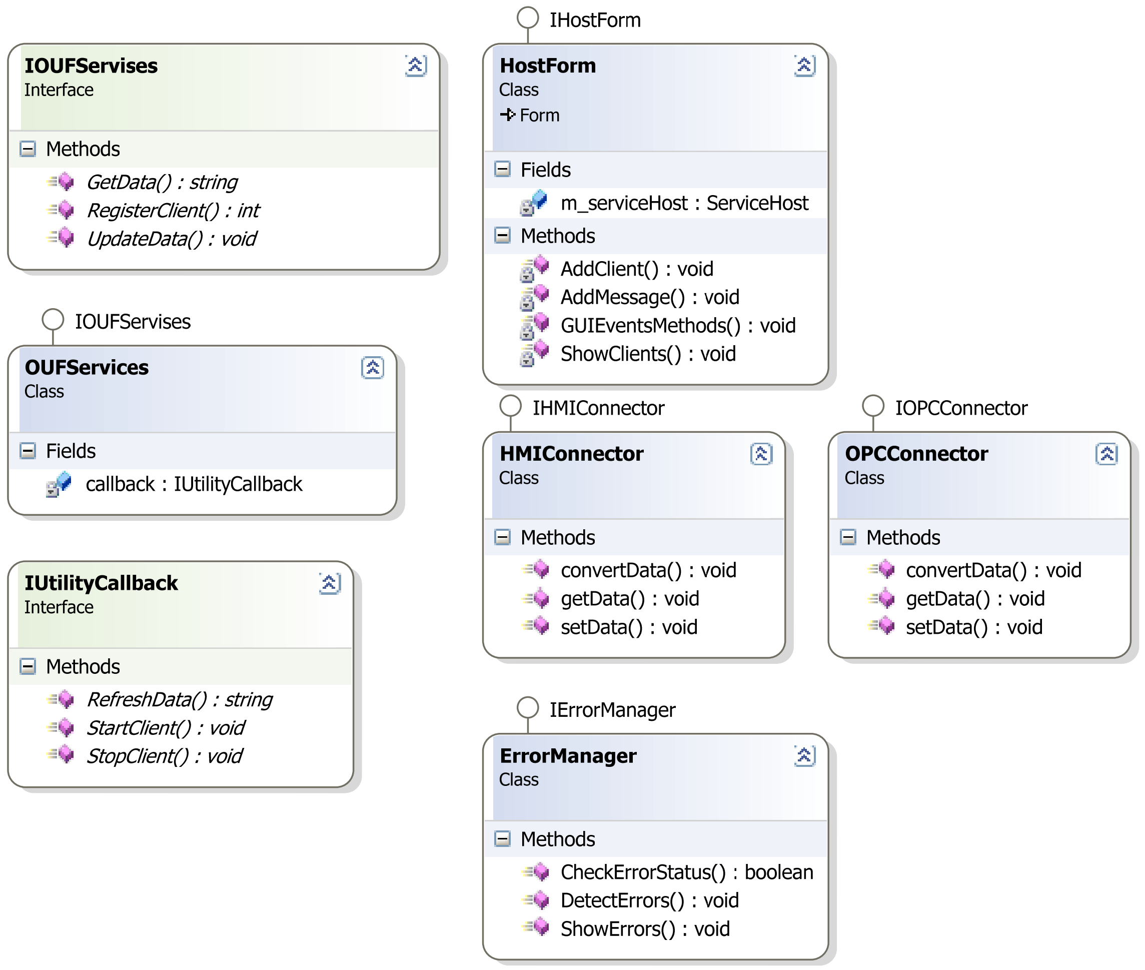


Figure 23 - Class diagram, server side

Most of the classes and interfaces have the same names as the components and interfaces in Figure 10. Interfaces IOUFServices and IUtilityCallback introduce the services and callbacks that are implemented in the prototype. HostForm (and interface IHostForm) represents the GUI component. The HMIConnector, OPCConnector and ErrorManager components were not implemented in the current prototype.

Example 10 and Example 11 show the implementation of the interfaces for the services (*IMessagingService*) and callbacks (*IMessagingServiceCallback*) in the prototype.

```
//interface for the prototype’s services
public interface IMessagingService
    {
//the list of services from the prototype
//method gets the information from the client and stores it in the
array
        [OperationContract]
        string GetData(string message);
// method distributes the numbers for the clients, returns int value
        [OperationContract]
        int RegisterClient();
// method updates the data from client,takes two parameters: int – is
the clients number, and the string – is the data
        [OperationContract(IsOneWay = true)]
        void UpdateData(int key, string data);
    }
```

Example 10 - IMassagingService interface

```
//interface for the prototype’s callnacks
public interface IMessagingServiceCallback
    {
//the list of callbacks from the prototype
//method that allows the server to run the client, takes int value
that is the clients number as a parameter
        [OperationContract(IsOneWay = true)]
        void StartClient(int key);
//method that allows the server to stop the running client, , takes
int value that is the clients number as a parameter
        [OperationContract(IsOneWay = true)]
        void StopClient(int key);
//method that allows the server to updates variables on the client
side (keys’ values), takes two parameters: int – is the clients
number, and the string – is the data
        [OperationContract(IsOneWay = true)]
        void RefreshData(int key, string data);
    }
```

Example 11 – IMessagingServiceCallback interface

Example 12 illustrates the implementation of the service class (MessagingService) in the prototype, and among other things the implementation of the RegisterClient() service. Example 13 illustrates how the client uses this service.

```
//prototype’s service class
public class MessagingService : IMessagingService
    {   //region allows to select a region of code and make it
collapsible
        #region IMessagingService Members
//declaration of a variable for client counter
        private static int ClientCounter = 1;
//identifies callback = 0
        IMessagingServiceCallback callback = null;
//declares an arraylist of storing the clients
        public IMessagingServiceCallback[] Clients = null;
//declares a static arraylist of storing the information
        public static ArrayList desc = new ArrayList();
public MessagingService() { //constructor
//creates an array for clients with size 100
            Clients = new IMessagingServiceCallback[100];
        }
//an example of the service’s implementation: service RegisterClient
        public int RegisterClient()
        {
//local variable
        int newClientKey = ClientCounter++;
//adds the new client to the client’s array
        Clients[newClientKey] =
OperationContext.Current.GetCallbackChannel<IMessagingServiceCallback
>();
//the method returns int value
        return newClientKey;
        } //... more methods here
        #endregion //end of region
    }
}
```

Example 12 - The service's class implementation

```
namespace Client //client’s namespace
{
      public class OrigoUtilityClient : IOrigoUtilityClient //class
that implements the client
    {
// see section 4.3 for this part
InstanceContext context = new InstanceContext(new
MessagingServiceCallback());
            MessagingServiceClient client = new
MessagingServiceClient(context);
      public static void Main(string[] args)
        {
           int key = client.RegisterClient(); //client using the
RegisterClient() method from the server side
        } //... more implementations
   }
}
```

Example 13 - Client calling the service

Figure 24 presents the class diagram for the client side of the solution. It is created by using the Ping Client as an example client.

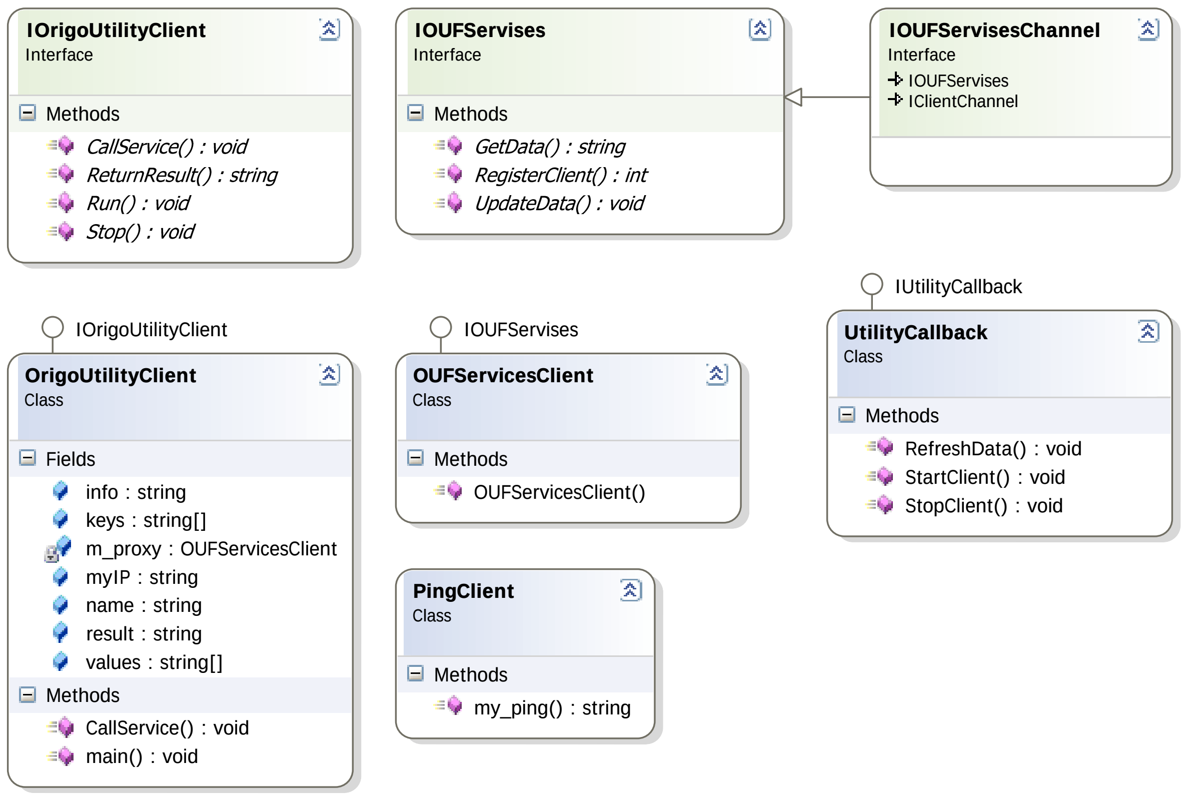


Figure 24 - Class diagram, client side

Example 14 shows the implementation of the callback in the prototype, and how it resembles to the service's implementation. How the server side calls the callback also resembles to the client calling the service, and an example for it (from the *MessagingService* class) is presented in Example 15

```
[CallbackBehavior(UseSynchronizationContext=true)]
    class MessagingServiceCallback : IMessagingServiceCallback,
IOrigoUtilityClient
    {
        #region IMessagingServiceCallback Members

        public void StartClient(int key) {
          // method's implementation here
        }

        #endregion
    }
}
```
Example 14 – MessagingServiceCallback

```
//...
public void RunClient(int key)
{
callback =
OperationContext.Current.GetCallbackChannel<IMessagingServiceCallback
>();
 callback.StartClient(key);
}
//...
```

Example 15 - Server side calling the callback