ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

INGENIERÍA ELECTROMECANICA

ESPECIALIDAD ELECTRÓNICA

BACK-END DEVELOPMENT OF A PROFINET-BASED DATA COLLECTOR APPLICATION AND FINAL DEPLOYMENT

Autor: Germán Ferreira Peña
Director: José Daniel Muñoz Frías

Madrid

Julio 2018

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

BACK-END DEVELOPMENT OF A PROFINET-BASED DATA COLLECTOR APPLICATION AND FINAL DEPLOYMENT

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el

curso académico 2017/2018 es de mi autoría, original e inédito y

no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada

de otros documentos está debidamente referenciada.

[illegible]

Fdo.: Germán Ferreira Peña Fecha: 19/07/2018

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Firmado por MUÑOZ FRIAS
JOSE DANIEL - 52573841G

Fdo.: José Daniel Muñoz Frías Fecha: 19/07/2018

contra la Universidad por terceros que vieran infringidos sus derechos e intereses a causa de la cesión.

d) Asumir la responsabilidad en el caso de que las instituciones fueran condenadas por infracción de derechos derivada de las obras objeto de la cesión.

6º. Fines y funcionamiento del Repositorio Institucional.

La obra se pondrá a disposición de los usuarios para que hagan de ella un uso justo y respetuoso con los derechos del autor, según lo permitido por la legislación aplicable, y con fines de estudio, investigación, o cualquier otro fin lícito. Con dicha finalidad, la Universidad asume los siguientes deberes y se reserva las siguientes facultades:

- La Universidad informará a los usuarios del archivo sobre los usos permitidos, y no garantiza ni asume responsabilidad alguna por otras formas en que los usuarios hagan un uso posterior de las obras no conforme con la legislación vigente. El uso posterior, más allá de la copia privada, requerirá que se cite la fuente y se reconozca la autoría, que no se obtenga beneficio comercial, y que no se realicen obras derivadas.
- La Universidad no revisará el contenido de las obras, que en todo caso permanecerá bajo la responsabilidad exclusive del autor y no estará obligada a ejercitar acciones legales en nombre del autor en el supuesto de infracciones a derechos de propiedad intelectual derivados del depósito y archivo de las obras. El autor renuncia a cualquier reclamación frente a la Universidad por las formas no ajustadas a la legislación vigente en que los usuarios hagan uso de las obras.
- La Universidad adoptará las medidas necesarias para la preservación de la obra en un futuro.
- La Universidad se reserva la facultad de retirar la obra, previa notificación al autor, en supuestos suficientemente justificados, o en caso de reclamaciones de terceros.

Madrid, a 19 de Julio de 2018

ACEPTA

Fdo……………………………………………… [illegible]

Motivos para solicitar el acceso restringido, cerrado o embargado del trabajo en el Repositorio Institucional:

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

INGENIERÍA ELECTROMECANICA

ESPECIALIDAD ELECTRÓNICA

BACK-END DEVELOPMENT OF A PROFINET-BASED DATA COLLECTOR APPLICATION AND FINAL DEPLOYMENT

Autor: Germán Ferreira Peña
Director: José Daniel Muñoz Frías

Madrid

Julio 2018

DESARROLLO BACK-END DE UNA APLICACIÓN DE ADQUISICIÓN DE DATOS BASADA EN PROFINET E IMPLEMENTACIÓN FINAL

Autor: Ferreira Peña, Germán.

Director: Muñoz Frías, Jose Daniel.

Entidad Colaboradora: Siemens CT

RESUMEN DEL PROYECTO

Introducción

El presente proyecto forma parte de un proyecto más amplio, que consiste en el desarrollo de una solución software para poder adquirir información de los dispositivos conectados a una red PROFINET de tal forma que:

- El inventario de los dispositivos PROFINET sea transparente: esto significa que se puedan conocer los dispositivos conectados a la red, obteniendo cierta información para identificarlos y comprobar su configuración de red.
- Conocer el estado de salud de los dispositivos con el fin de facilitar sus actividades de mantenimiento.

Actualmente existen soluciones software en el mercado que proporcionan una lista que contiene los dispositivos conectados a una red PROFINET. Algunas de las soluciones más significativas son SolarWinds Network Performance Monitor [1], Siemens PRONETA [2] y SIMATIC Automation Tool [3]. Sin embargo, estos programas tan solo proporcionan información básica sobre estos dispositivos, de modo que para conocer su estado de salud es necesario utilizar otras aplicaciones especializadas y conectar cada dispositivo a un ordenador con un cable Ethernet. Por ello, es necesario el desarrollo de una aplicación que permita obtener el estado de salud de los dispositivos de la red de forma **centralizada**.

Este proyecto trata el desarrollo del componente de adquisición de datos de la aplicación, que obtiene toda la información necesaria de los dispositivos de la red para su posterior análisis; la creación de una plataforma para testear la aplicación y crear una solución para que el cliente final pueda instalar la aplicación.

Metodología

La metodología que se siguió en la realización del proyecto fue:

En primer lugar, se desarrolló el componente de adquisición de datos para la aplicación (ver Figura 1). Para programarlo, se usó el lenguaje C#, Visual Studio Professional 2015 y la API de SAT. Se programó el componente para cumplir con las prestaciones y nivel de seguridad requeridos en una aplicación de uso industrial. Además, se diseñó una parte del componente que permite obtener datos de los dispositivos por protocolo SNMP [4] con el fin de verificar la información.

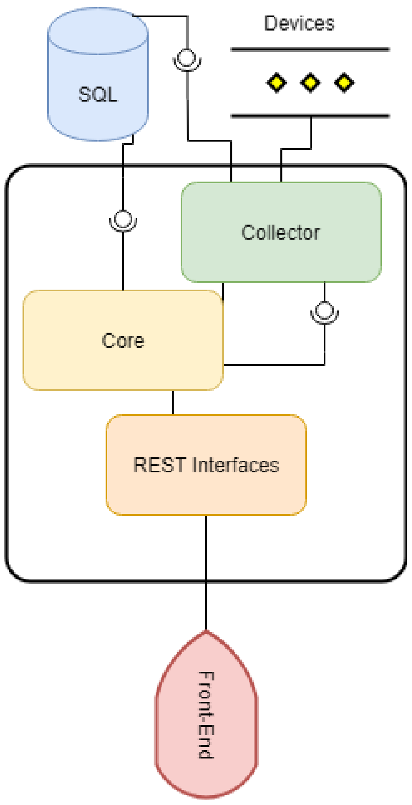

Figura 1: Arquitectura de la aplicación

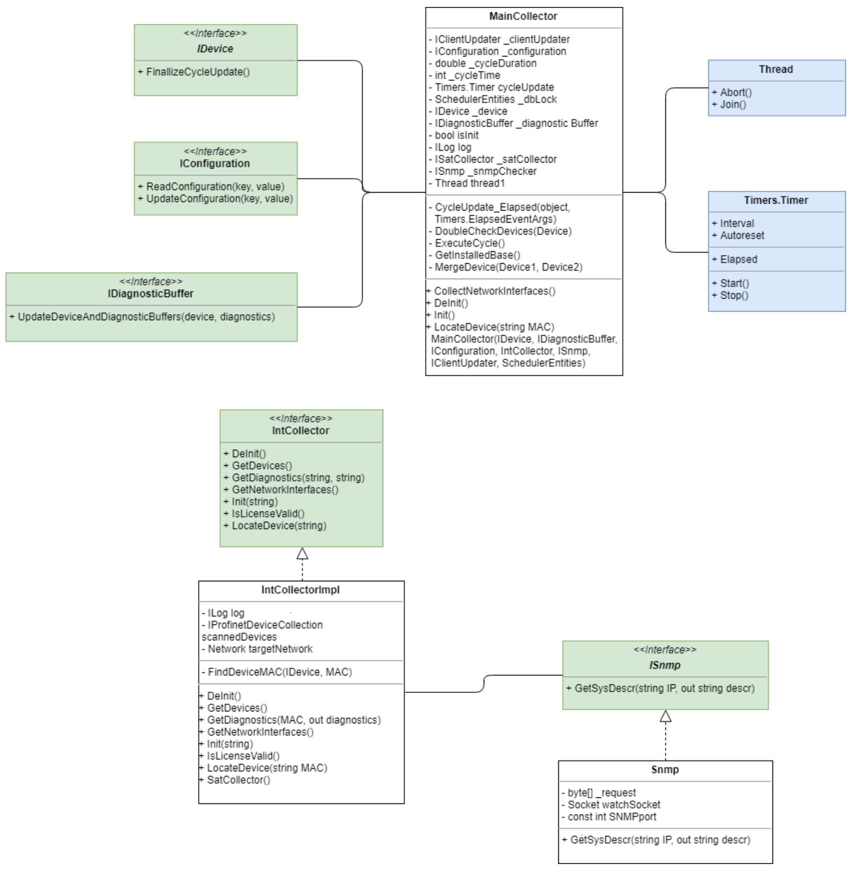
<<Interface>>
IDevice
+ FinallizeCycleUpdate()
<<Interface>>
IConfiguration
+ ReadConfiguration(key, value)
+ UpdateConfiguration(key, value)
<<Interface>>
IDiagnosticBuffer
+ UpdateDeviceAndDiagnosticBuffers(device, diagnostics)
MainCollector
- IClientUpdater _clientUpdater
- IConfiguration _configuration
- double _cycleDuration
- int _cycleTime
- Timers.Timer cycleUpdate
- SchedulerEntities _dbLock
- IDevice _device
- IDiagnosticBuffer _diagnostic Buffer
- bool isInit
- ILog log
- ISatCollector _satCollector
- ISnmp _snmpChecker
- Thread thread1
- CycleUpdate_Elapsed(object, Timers.ElapsedEventArgs)
- DoubleCheckDevices(Device)
- ExecuteCycle()
- GetInstalledBase()
- MergeDevice(Device1, Device2)
+ CollectNetworkInterfaces()
+ DeInit()
+ Init()
+ LocateDevice(string MAC)
MainCollector(IDevice, IDiagnosticBuffer, IConfiguration, IntCollector, ISnmp, IClientUpdater, SchedulerEntities)
Thread
+ Abort()
+ Join()
Timers.Timer
+ Interval
+ Autoreset
+ Elapsed
+ Start()
+ Stop()
<<Interface>>
IntCollector
+ DeInit()
+ GetDevices()
+ GetDiagnostics(string, string)
+ GetNetworkInterfaces()
+ Init(string)
+ IsLicenseValid()
+ LocateDevice(string)
IntCollectorImpl
- ILog log
- IProfinetDeviceCollection scannedDevices
- Network targetNetwork
- FindDeviceMAC(IDevice, MAC)
+ DeInit()
+ GetDevices()
+ GetDiagnostics(MAC, out diagnostics)
+ GetNetworkInterfaces()
+ Init(string)
+ IsLicenseValid()
+ LocateDevice(string MAC)
+ SatCollector()
<<Interface>>
ISnmp
+ GetSysDescr(string IP, out string descr)
Snmp
- byte[] _request
- Socket watchSocket
- const int SNMPport
+ GetSysDescr(string IP, out string descr)

Figura 2: Diagrama de objetos del componente de recogida de datos

Una vez completado el componente (Figura 2), se diseñó una plataforma para realizar pruebas de funcionamiento de la aplicación y se realizaron diversas pruebas. Para ello, se estudiaron diferentes herramientas compatibles con Visual Studio 2015 y finalmente, se realizó un test de integración del programa utilizando las herramientas del propio IDE.

Por último, se creó una solución para implementar el software en los ordenadores de posibles usuarios finales considerando diferentes sistemas operativos y configuraciones. Para ello se estudiaron diferentes herramientas para su desarrollo y se implementó utilizando Inno Setup y SIT.

Resultados

El componente de adquisición de datos que se desarrolló cumple con las prestaciones requeridas por el proyecto. Su diseño es modular, por lo que es posible reutilizar este componente en futuros desarrollos de esta aplicación u otras aplicaciones que requieran esta funcionalidad.

Con respecto a la plataforma de pruebas, su puesta en marcha permitió encontrar y depurar errores de la aplicación, así como mejorar la integración de los componentes.

Finalmente, se creó un instalador que cumple con los requisitos del cliente.

Conclusiones

Desde el punto de vista del desarrollo del componente de adquisición de datos, C# es un lenguaje ideal para desarrollar aplicaciones de este estilo, ya que posee capacidades de manejar memoria de forma automática y además permite tratar las excepciones y fallos del código de una forma más accesible.

Por otra parte, el diseño de la arquitectura de la aplicación conectando los componentes por medio de interfaces permite dividir las funciones de los componentes de forma clara y concisa.

Con respecto a la plataforma de pruebas, su diseño y utilización resultaron útiles durante el desarrollo de el presente proyecto y el resto de los componentes de la aplicación, ya que permitió detectar fallos y mejorar la integración de la aplicación. Como consecuencia, la **calidad** de la aplicación mejoró.

Finalmente, el instalador de la aplicación resulta crucial desde el punto de vista del cliente, ya que supone su primer contacto con el software. Es importante que el proceso de instalación y desinstalación sea automatizado independientemente de la configuración del ordenador del usuario.

Referencias

[1] Solarwinds, [Online]. Available: https://www.solarwinds.com/topics/network-device-scanner.

[2] Siemens AG, [Online]. Available: https://support.industry.siemens.com/cs/document/67460624/proneta-2-4-0-44-commissioning-and-diagnostics-tool-for-profinet?dti=0&lc=en-WW.

[3] Siemens, [Online]. Available: https://support.industry.siemens.com/cs/document/98161300/simatic-automation-tool-la-herramienta-de-puesta-en-marcha-y-operación-de-mantenimiento-para-los-módulos-simatic-?dti=0&lc=es-WW.

[4] Paessler, "How do SNMP, MIBs and OIDs work?," 2010. [Online]. Available: https://kb.paessler.com/en/topic/653-how-do-snmp-mibs-and-oids-work.

BACK-END DEVELOPMENT OF A PROFINET-BASED DATA COLLECTOR APPLICATION AND FINAL DEPLOYMENT

PROJECT SUMMARY

Abstract

The present thesis is part of an application which consists of the development of a software solution to gather information of devices connected to a PROFINET network in order to:

- Know the installed base of PROFINET devices. This involves gathering enough information to identify them and check their connectivity settings.
- Determine health status of the devices in order to ease maintenance activities.

Currently there are some software solutions available that provide a list containing all the devices connected to a certain PROFINET network. The most remarkable solutions are SolarWinds Network Performance Monitor [1], Siemens PRONETA [2] and SIMATIC Automation Tool [3]. However, these solutions only provide basic information about the connected devices. In order to determine health status of the devices it is required to use specialized software and connect each device to a computer using an Ethernet interface. From this point arises the need of a **centralized** solution that can provide plant transparency (knowing the devices connected to the PROFINET network) and health status of the installed base.

The present Thesis describes the development of the data collector component of the application. This component is in charge of gathering all necessary information from PROFINET devices for further analysis. A testing platform was created for the whole application including Collector, and a deployment solution was implemented for the final user.

Methodology

The methodology followed during the development of the present Thesis is depicted below:

First, the application Collector component was developed (see app architecture in Figure 1). The component was coded in C# using the SAT API and Visual Studio Professional 2015 was used as the project IDE. The component was coded to achieve the required specifications including software security for industrial purposes. Moreover, an entity that can verify device data using SNMP [4] protocol was developed.

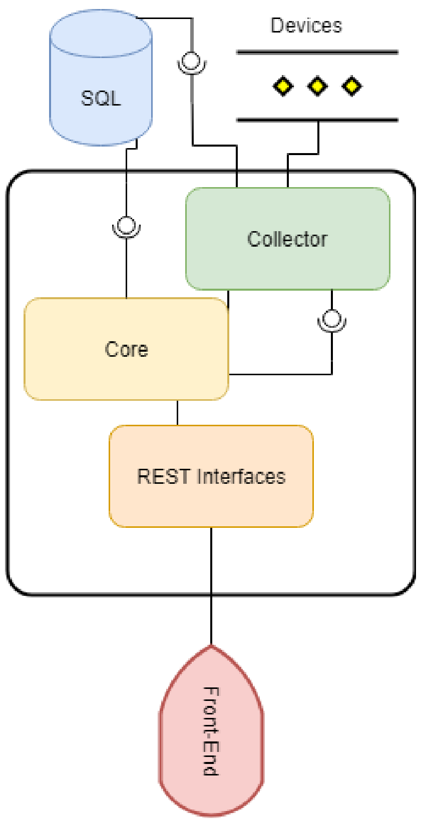

Figure 1: Application architecture

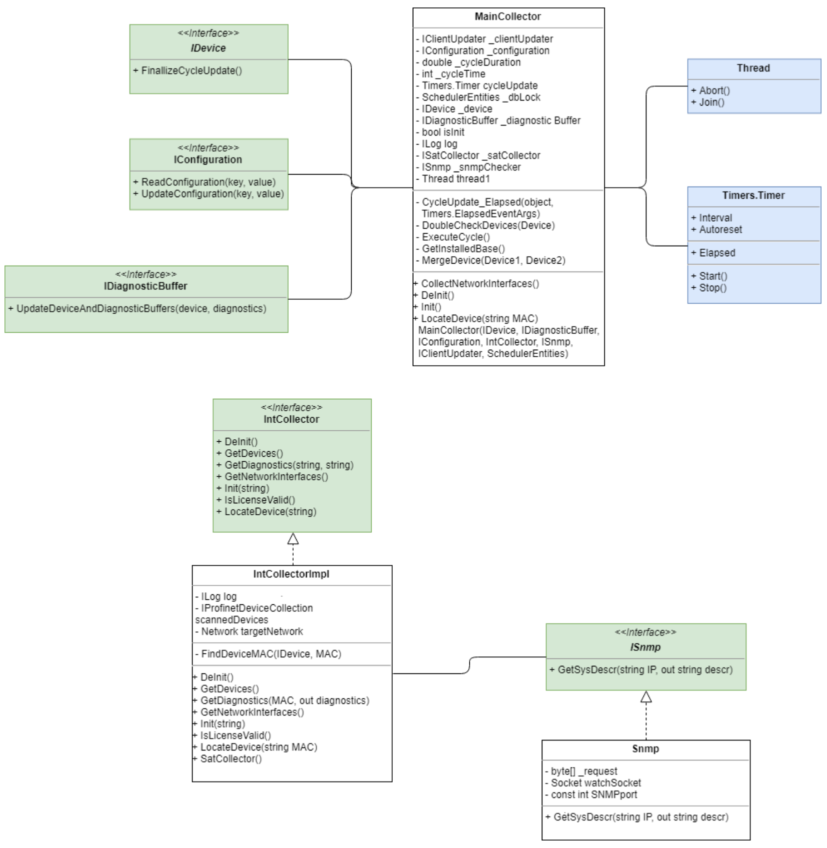
<<Interface>>
IDevice
+ FinallizeCycleUpdate()
<<Interface>>
IConfiguration
+ ReadConfiguration(key, value)
+ UpdateConfiguration(key, value)
<<Interface>>
IDiagnosticBuffer
+ UpdateDeviceAndDiagnosticBuffers(device, diagnostics)
MainCollector
- IClientUpdater _clientUpdater
- IConfiguration _configuration
- double _cycleDuration
- int _cycleTime
- Timers.Timer cycleUpdate
- SchedulerEntities _dbLock
- IDevice _device
- IDiagnosticBuffer _diagnostic Buffer
- bool isInit
- ILog log
- ISatCollector _satCollector
- ISnmp _snmpChecker
- Thread thread1
- CycleUpdate_Elapsed(object, Timers.ElapsedEventArgs)
- DoubleCheckDevices(Device)
- ExecuteCycle()
- GetInstalledBase()
- MergeDevice(Device1, Device2)
+ CollectNetworkInterfaces()
+ DeInit()
+ Init()
+ LocateDevice(string MAC)
MainCollector(IDevice, IDiagnosticBuffer, IConfiguration, IntCollector, ISnmp, IClientUpdater, SchedulerEntities)
Thread
+ Abort()
+ Join()
Timers.Timer
+ Interval
+ Autoreset
+ Elapsed
+ Start()
+ Stop()
<<Interface>>
IntCollector
+ DeInit()
+ GetDevices()
+ GetDiagnostics(string, string)
+ GetNetworkInterfaces()
+ Init(string)
+ IsLicenseValid()
+ LocateDevice(string)
IntCollectorImpl
- ILog log
- IProfinetDeviceCollection scannedDevices
- Network targetNetwork
- FindDeviceMAC(IDevice, MAC)
+ DeInit()
+ GetDevices()
+ GetDiagnostics(MAC, out diagnostics)
+ GetNetworkInterfaces()
+ Init(string)
+ IsLicenseValid()
+ LocateDevice(string MAC)
+ SatCollector()
<<Interface>>
ISnmp
+ GetSysDescr(string IP, out string descr)
Snmp
- byte[] _request
- Socket watchSocket
- const int SNMPport
+ GetSysDescr(string IP, out string descr)

Figure 2: Collector component object diagram

When the Collector development stage was completed (Figure 2), a testing framework was designed. Testing tools compatible with Visual Studio 2015 were analyzed in order to add them to the testing framework. As a result, it was concluded that Visual Studio testing capabilities were suitable to run integration and module tests.

Finally, a deployment solution was created. This solution is aimed for the final users of the application who could install it in their computers. The installer had to consider different machine configurations to deploy the application automatically. Some deployment development solutions were considered and the installer was implemented using Inno Setup and SIT.

Results

The Collector component fulfilled the features required by the project. Its modular design makes it possible to reuse this component in future developments of the present application or in other applications that require Collector functionalities.

The testing framework supported the application development, because tests discovered bugs that were corrected in time, and it improved application integration.

Finally, regarding deployment, the solution that was developed in the present thesis fulfilled client requirements.

Conclusions

Regarding Collector development, C# is the best language to develop applications like the presented in this Thesis, because it provides automatic memory management capabilities, and exception and fault handling capabilities.

Additionally, connecting application components with interfaces allows to split component responsibilities in order to have a clear and concise architecture.

Regarding the testing framework, its design and usage were successful during the development of the present project and the rest of the app components because bugs were found in time and it helped to integrate the application parts. As a result, the **quality** of the software was increased.

Lastly, the application installer is crucial from the customer point of view. It is essential that the installation and uninstallation process is automatic, regardless of final user´s computer configuration. It represents the first and the final interaction with the software.

References

[1] Solarwinds, [Online]. Available: https://www.solarwinds.com/topics/network-device-scanner.

[2] Siemens AG, [Online]. Available: https://support.industry.siemens.com/cs/document/67460624/proneta-2-4-0-44-commissioning-and-diagnostics-tool-for-profinet?dti=0&lc=en-WW.

[3] Siemens, [Online]. Available: https://support.industry.siemens.com/cs/document/98161300/simatic-automation-tool-la-herramienta-de-puesta-en-marcha-y-operación-de-mantenimiento-para-los-módulos-simatic-?dti=0&lc=es-WW.

[4] Paessler, "How do SNMP, MIBs and OIDs work?," 2010. [Online]. Available: https://kb.paessler

Table of contents

List of Figures

List of Tables

Part I THESIS

Chapter 1 INTRODUCTION

Due to the large number of devices that are present in an industrial PROFINET Network, it is interesting to access certain information about the installed base. The most remarkable topics regarding this information are:

- Plant transparency: knowing which devices are connected to the network and getting some basic information about them in order to identify and check their connectivity configuration.
- Health status: knowing the status of the different devices in order to facilitate maintenance activities.

In order to obtain this information, there are some centralized (meaning from a unique computer) software solutions that provide the list of devices connected to a PROFINET network. The most significant software apps that are currently in the market are described in 1.1. However, these solutions only gather basic information about the installed base. In order to determine the health status of a device, it requires to connect it directly to a computer with specialized manufacturer applications.

Because of the large number of devices present on a PROFINET network, connecting each device to a computer via Ethernet interface is a tedious process. Even sometimes can be hazardous because of the environment where the devices are present.

A solution is being developed to tackle the problems exposed above. From that, arises the present Master Thesis.

The current Thesis encompasses developing the Collector part of the software, which oversees gathering all necessary information from the devices; creating a testing platform for the whole application and providing a deployment solution to distribute the application to the final clients.

1.1 EXISTING TECHNOLOGIES

Due to the scope of the present thesis, the study of existing technologies will be focused on PROFINET Device scanner software solutions.

As stated in Chapter 1, these solutions retrieve limited information about the devices. Moreover, they are mostly focused on the analysis of network performance.

1.1.1 SOLARWINDS NETWORK PERFORMANCE MONITOR

This tool is intended to be a network analyzer [1] that has several features to use for network management activities. Regarding the scanner part of the software, this application uses Simple Network Management Protocol (SNMP) to discover devices on the network.

SNMP protocol has the advantage that it is compatible with equipment from different vendors. Therefore, the tool will discover all the devices that answer to SNMP requests.

The *discovery* functionality of this application scans the network for nodes. When nodes are found, it is possible to add them to a database to monitor them. Monitoring of network nodes is based on retrieving information via SNMP to ensure that certain device parameters (e.g. Temperature, Fan Speed…) are within limits. To initialize the discovery process, it is necessary to provide IP ranges, subnets or IP addresses of the devices.

This tool is a commercial software. Its license price starts at $2,895.

1.1.2 SIEMENS PRONETA 2.4

PRONETA [2] is a free tool for the analysis and configuration of PROFINET networks with special support for ET 200 distributed IO. It has a feature called IO Test which can provide support of a plant's wiring, providing the necessary documentation.

Siemens PRONETA supports the diagnosis and commissioning of PROFINET networks in automation systems by providing the following features:

- Topology overview, which automatically scans PROFINET networks and displays all connected devices. The list of all connected devices can be exported, including the details of all electronic modules. PRONETA allows for configuring components and comparing between the actual installation and a reference plant.
- IO Test for the rapid test of the wiring and module configuration of the components. By reading and writing inputs and outputs, PRONETA ensures that the connections between distributed IO components and their sensors and actuators have been set up properly. PRONETA can create test profile templates and save test protocols to document test results.
- All tasks can be accomplished even before a CPU has been linked up to the network. Since no further engineering tools besides PRONETA and no dedicated hardware are required, PRONETA facilitates a quick and comfortable check of a plant configuration at an early stage.

1.1.3 SIMATIC AUTOMATION TOOL

This is another tool provided by Siemens AG. However, this tool is not freeware.

It provides the following features [2]. Some of them are only available to Siemens devices (CPUs, HMIs, IO and others):

- Scans a PROFINET/Ethernet network and identifies all the devices connected to the network.
- Can make either LEDs of equipment or HMI screens flash to physically locate devices.
- Creates a table that maps all accessible devices of the network.
- Assigns addresses (IP, subnet and gateway) and PROFINET Name of the node.
- Sets current time of the PG/PC as time in a CPU.
- Loads programs to a CPU or HMI device.
- Erases CPU memory.
- Backups/Restores CPU data files to/from a backup file.
- Loads service data from a CPU.
- Reads diagnostics register from a CPU.
- Restores default configuration of a CPU.
- Updates CPU and modules Firmware.
- Loads, downloads and erases Recipes data (these data are stored in a SMC) of a CPU.
- Loads or erases data from log files (storage in a SMC) of a CPU.
- Writes documentation about network configuration and saves it in:
    - .csv file.
    - .sat file encrypted and protected by a password.

All CPU features can be utilized in parallel with several CPUs at the same time.

1.1.4 SOFTWARE COMPARISON

To sum up, on Table 1 the different tools that are available on the market are compared regarding capabilities and price:

	License	Network analysis	Installed base transparency	Device diagnostics
SolarWinds	Commercial	Performance	YES	NO
PRONETA	Free	Wiring	YES	NO
SAT	Commercial	NO	YES	YES

Table 1: Comparison between tools available on the market

1.2 INCENTIVE

As mentioned in the introduction, plants and industries present a large collection of devices connected to PROFINET Networks. This collection is difficult to manage, and a centralized software solution is ideal to deal with populated networks.

The application customer looks for a solution that gives transparent information about the installed devices and their health status.

This solution will improve maintenance of the plant, therefore helping to prevent failures and operation shutdowns. As consequences, maintenance costs are reduced, managing maintenance is easier and plant productivity is increased.

1.3 OBJECTIVES

As stated in Chapter 1, the aims of this project are:

- Develop the Collector component.
- Design and implement a testing framework for the application.
- Create a redistributable package (app deployment).

In the following subchapters, the features of each part are listed.

1.3.1 COLLECTOR COMPONENT

1. Gather devices connected to the network.
2. Locate devices: physically detect devices of the PROFINET network.
3. Get status information to determine devices health condition.
4. Time requisites: the Collector operation must be controlled by an asynchronous timer.

1.3.2 TEST FRAMEWORK

1. Unit testing and module testing.
2. Continuous integration testing.

1.3.3 APPLICATION DEPLOYMENT

1. Develop a redistributable package that meets customer requirements.
2. Integration of the package in the company installation framework.

1.4 METHODOLOGY

As this master thesis is based on software, a software development model is going to be followed. The chosen model is the **waterfall** approach that is shown in Figure 1:

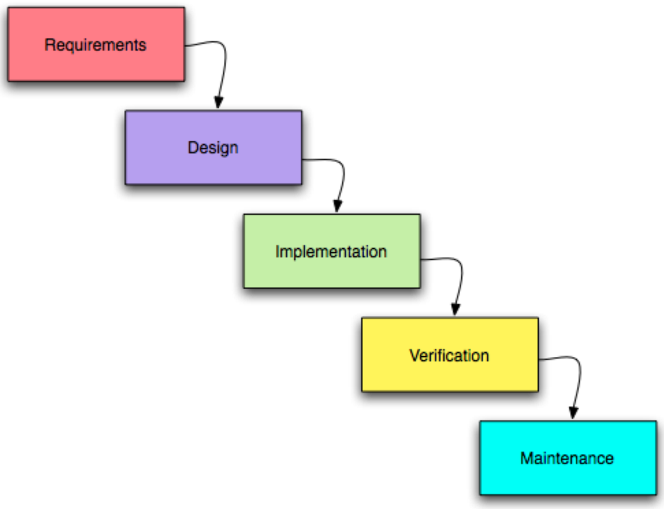

Figure 1: Waterfall approach.

Source: http://www.mikesmart.com/application_development/waterfall_method.htm

The *Implementation* stage splits into several sprints to fully develop the two-week features in time. The estimated task plan is presented in Table 2:

Sr.No.	Tasks	12-mar	19-mar	26-mar	02-abr	09-abr	16-abr	23-abr	30-abr	07-may	14-may	21-may	28-may
		Sprint 1		Sprint 2		Sprint 3		Sprint 4		Sprint 5		Sprint 6	
1	Basic Info.	🟩	🟩										
2	Locate Device.			🟩	🟩								
3	Status Info.					🟩	🟩						
4	App License.							🟩	🟩				
5	Integration Test.									🟩	🟩		
6	Deployment Dev.											🟩	🟩

Table 2: Sprint Plan of the Project

- Feature 1 aims to gather basic information about PROFINET devices such as Name, MAC and IP addresses, subnet mask…
- Feature 2 allows to physically locate the device using a functionality derived from PROFINET protocol.
- Feature 3 aims to gather detailed information from devices. This information is provided to other components of the application in order to determine health status information.

- Feature 4 integrates a module to run the application within a user license.
- Task 5 implements an integration test for the whole application.
- Task 6 consists of developing an installer solution for the application.

1.5 TOOLS TO USE

For the execution of this project, the following Hardware and Software tools are required:

- Hardware
  - PC: the computer must run Windows OS.
  - PROFINET Devices base: a test base, containing different PROFINET devices such as PLCs, HMIs, Motor controllers and IO devices.
  - PROFINET Network: the devices mentioned above must be connected to a real PROFINET network to perform testing and analyzing performance of the application.
- Software
  - Microsoft Visual Studio 2015 Professional: this software will be the IDE of the application.
  - VMWare: this software will provide virtual environment capabilities to execute another operating system in the PC. The aim of using a virtual environment is to test and simulate either the application or the deployment software in a different OS without harming registry or configuration files of the non-virtual machine.
  - SAT: this third-party software provides a C# API that provides capabilities to communicate with PROFINET Devices.
  - Deployment application: the project will use an external application to create a redistribute package to the final customers.

1.6 INTRODUCTION

In order to ease the read of the present thesis, this section gives a brief description about the following chapters:

- Chapter 2 describes the development of the Collector component of the application. This chapter explains the architecture of the component and how its required capabilities are implemented.
- Chapter 3 presents the testing framework of the application and explains integration testing and module testing of the Collector component.
- Chapter 4 explains the process of developing an installer solution using a freeware and a proprietary tool.
- Chapter 5 presents the achievements made during this project.
- Chapter 6 states some conclusions about the executed tasks.
- Chapter 7 presents some future developments based on the present thesis.

Chapter 2 BACK END DEVELOPMENT

2.1 OVERVIEW

As stated in Chapter 1, the Collector component is in charge of gathering the data of the PROFINET network devices. Deepening in the previous functionality, the Collector is the component responsible for:

- ➢ Configuring a network interface: this functionality means that it is possible to view all the network adapters installed on the machine. The user will be able to select one of them and then the Collector will set it during runtime and validate it.
- ➢ Scanning the network: the component scans the PROFINET Network connected to the selected and **validated adapter**. Once the scan has been completed, a list stores the installed devices in the Network with the gathered information. This information is sent to other components of the software, which process these data in order to determine the health status of the devices.
- ➢ Using two different channels as source: in order to ensure that the information that is being communicated between the devices and the computer is reliable, the component also makes a *double-check* of the information. To do that, the data Collector connects to the devices after the scan using socket communication and sends SNMP protocol requests. These requests will return data fields that will be used to make the *double-check.*
- ➢ Threading and timing: the component must be executed with a certain frequency that the user can select through the UI. To improve responsiveness of the app, the component must be executed in a new thread.
- ➢ Data formatting: the information retrieved from devices needs to be processed to be sent to the rest of the components of the app (mainly database and core).

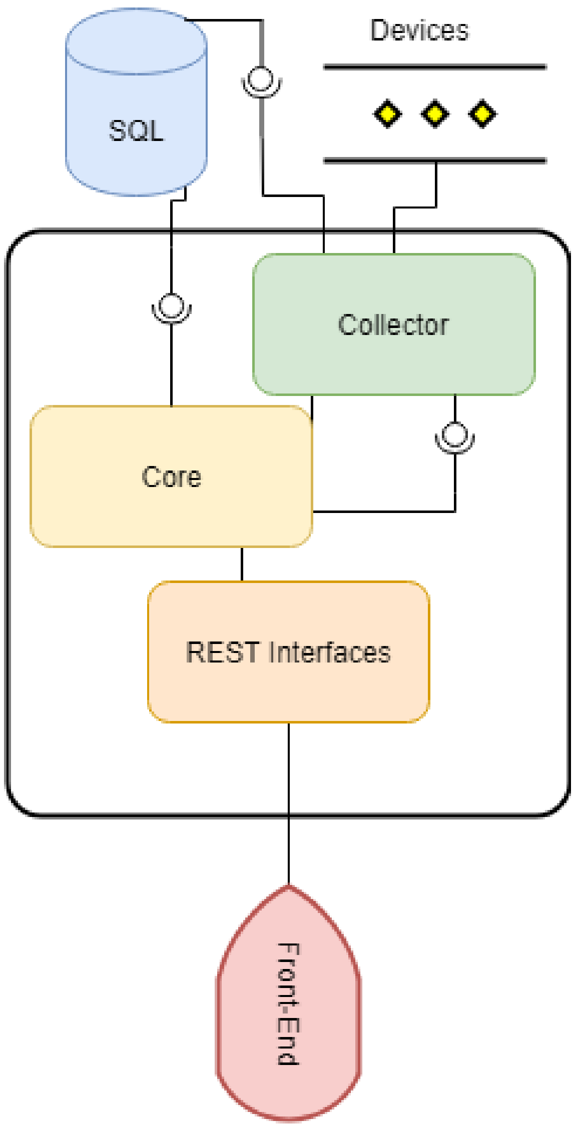

Figure 2: Basic High Level architecture

Figure 2 represents the high-level design of the components involved in the Collector and the present thesis purpose. According to the presented architecture, users interact with the app through a web interface. This web interface uses REST APIs to interact with the application and the database.

2.2 INTERFACES AND COMPONENT RELATIONS

As seen in 2.1, the Collector will interact with the actual installed base of PROFINET Devices, the core and the database of the application.

The application architecture isolates its components by using programming interfaces. This ensures that each component of the software has a single functionality, which cannot be modified by an external software part.

The use of programming interfaces is not only used in the internal structure of the application, but also the Collector component uses interfaces to communicate with the PROFINET devices. These interfaces are:

- SAT: this tool provides an Application Programming Interface, which contains high level methods, properties and self-defined types to interact with PROFINET devices without having to implement low-level protocols and hardware communication.
- ISnmp: this interface is defined and implemented in the present Master Thesis. ISnmp is in charge of the *double-checking* functionality that was described in 2.1. The interface provides methods to communicate and get certain information of the devices for that purpose. The functionality of the interface will be explained in depth in 2.5.5.

2.2.1 COMPONENTS

The relation between the Collector and other parts of the application was stated in 2.1 and 2.2. This section will focus on describing these connections. The depth of descriptions is constrained by confidentiality reasons of the Project.

- Core: core can get processed data from Collector for health status analysis. The interface provides methods to send the installed base information to the component.
- Database: there are methods to read configuration data that will be used to authorize Collector to execute its task. Configuration parameters depend on the user preferences and can be changed using the UI. Available configuration parameters that affect Collector functionality are:

- Network adapter: the user will select one network interface from a list of available adapters. This list is provided by SAT.
- Cycle-time update: the user can select the update cycle-time of the device collection.

2.2.2 IMPORTED FUNCTIONALITY

As explained in 2.2, Collector component uses an API to interact with the PROFINET Network, encompassing different classes, methods and properties to work with. In this section, the imported functionality from the API is described:

2.2.2.1 Result Class

This class is used as the returned object when calling methods of other classes or interfaces from SAT. It contains information about the degree of success of the performed operation.

2.2.2.2 Network Class

The *Network* class performs functions using a network interface card installed on the machine.

The most important methods that this class provides are presented below. All of them return a *Result* object:

- *List available network interfaces:* this method identifies the available network interface cards and returns a list of strings containing their names.
- *Set network interface:* sets the interface card (from the available interfaces got by the previous method) to work with.
- *Scan network:* it outputs a collection of items, where each item represents a PROFINET device. Devices include but are not limited to PLCs, local modules, engine controllers and HMIs.
- *Set timeout for operations:* allows to set a timeout value to the methods that are part of this class.
- *Get timeout value:* retrieves the current timeout value.
- *Check license:* with the returned Result object, it allows to check the license of the SAT API.

2.2.2.3 IDeviceCollection Interface

This interface gathers a collection containing the devices scanned by *ScanDevices* method. As a collection, it presents the functionality to iterate between items (*IEnumerable* interface), and also presents some other remarkable features:

- Filtering elements: it is possible to filter elements by type of PROFINET device e.g. CPUs, HMIs…
- Finding specific devices in the collection: the interface provides a method that can find a device by introducing its IP or MAC address.

2.2.2.4 IDevice Interface

This interface represents each PROFINET device that was included in the *IDeviceCollection* described above.

The interface provides a significant number of properties. Some of the most remarkable are:

- IP address.
- MAC address.
- Subnet mask.
- Profinet Name of the device.
- Subnet mask.

IDevice presents some methods as well. For the purpose of this application, only a method called *Identify()* was used. This method provides the capability to send a request to physically identify a certain node in the network. The devices that are compatible with this feature flash some LEDs or their screens.

2.2.2.5 IPLC Interface

This interface is only present in compatible PLC devices and gives more detailed information about them. From the collection, it is possible to filter these devices from the *IDeviceCollection* using its filtering methods or by casting each *IDevice* item to a *IPLC* interface.

2.2.3 EXPORTED FUNCTIONALITY

As described in 2.2, the application architecture is based on interfacing. For the Collector component, an interface called ICollector was created. This interface provides the following methods as exported functionality:

- *Initialize():* this method initializes the component. The initialization process checks the configuration inputs, and if parameters are valid, then Collector performs the scan operation and initializes the cyclic update of information. The scan operation and cyclic update process will be explained in detail in 2.3.
- *DeInitialize()*: this method performs a de-initialization of the component. This stops the cyclic scanning process of the Collector component. It is described in 2.3.
- *List<string> GetNetworkInterfaces():* this function will return a list containing the available network interfaces in the machine. This list is provided by the *QueryCards* method of the Network class (see 2.2.2.2).
- *Identify(string MAC address):* this method physically identifies a device by flashing some LEDs or device´s screen (as in HMIs) providing its MAC address. The communication process which sends the request to the device is done by *Identify* method of the *IDevice* interface (2.2.2.4).

2.3 PERFORMANCE REQUIREMENTS

As stated in 2.1, the scanning process of Collector must work on a separate thread with a certain frequency.

Frequency is set by the user from the UI and Collector will have to scan the network accordingly.

The reason for the Collector component to work on a separate thread is to improve the **responsiveness** of the application. The scan method from SAT, which is the main method of the scan functionality, blocks the code execution until it is completed. The time this method takes to be completely executed varies depending on the number of devices connected to the network, but it is expected to be in the range of tens of minutes.

In order to fulfil the desired specifications, part of the code design is shown in Figure 3:

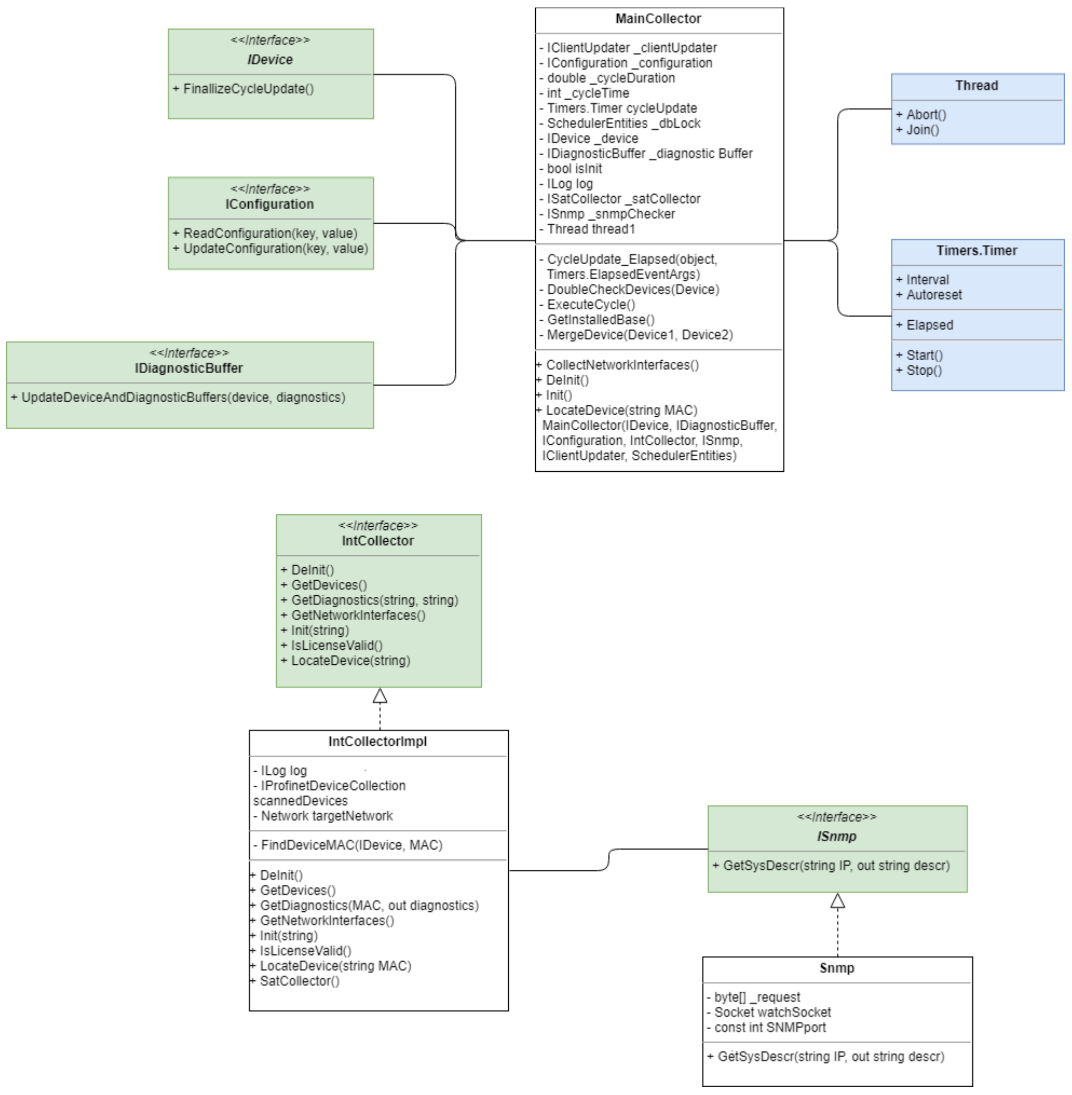
<<Interface>>
IDevice
+ FinalizeCycleUpdate()
<<Interface>>
IConfiguration
+ ReadConfiguration(key, value)
+ UpdateConfiguration(key, value)
<<Interface>>
IDiagnosticBuffer
+ UpdateDeviceAndDiagnosticBuffers(device, diagnostics)
MainCollector
- IClientUpdater _clientUpdater
- IConfiguration _configuration
- double _cycleDuration
- int _cycleTime
- Timers.Timer cycleUpdate
- SchedulerEntities _dbLock
- IDevice _device
- IDiagnosticBuffer _diagnostic Buffer
- bool isInit
- ILog log
- ISatCollector _satCollector
- ISnmp _snmpChecker
- Thread thread1
- CycleUpdate_Elapsed(object, Timers.ElapsedEventArgs)
- DoubleCheckDevices(Device)
- ExecuteCycle()
- GetInstalledBase()
- MergeDevice(Device1, Device2)
+ CollectNetworkInterfaces()
+ DeInit()
+ Init()
+ LocateDevice(string MAC)
MainCollector(IDevice, IDiagnosticBuffer, IConfiguration, IntCollector, ISnmp, IClientUpdater, SchedulerEntities)
Thread
+ Abort()
+ Join()
Timers.Timer
+ Interval
+ Autoreset
+ Elapsed
+ Start()
+ Stop()
<<Interface>>
IntCollector
+ DeInit()
+ GetDevices()
+ GetDiagnostics(string, string)
+ GetNetworkInterfaces()
+ Init(string)
+ IsLicenseValid()
+ LocateDevice(string)
IntCollectorImpl
- ILog log
- IProfinetDeviceCollection scannedDevices
- Network targetNetwork
- FindDeviceMAC(IDevice, MAC)
+ DeInit()
+ GetDevices()
+ GetDiagnostics(MAC, out diagnostics)
+ GetNetworkInterfaces()
+ Init(string)
+ IsLicenseValid()
+ LocateDevice(string MAC)
+ SatCollector()
<<Interface>>
ISnmp
+ GetSysDescr(string IP, out string descr)
Snmp
- byte[] _request
- Socket watchSocket
- const int SNMPport
+ GetSysDescr(string IP, out string descr)

Figure 3: Simplified Collector Object Diagram

To execute Collector component, the app core calls it via the *ICollector* interface. *Init()* and *DeInit()* methods provide capabilities to initialize or close the component. This sequence is executed every time the user updates one of the possible configuration values (cycle interval time and/or network interface).

Init() method first checks SAT and application licenses and then the Collector input configuration. Only if all conditions are met, *Init()* method calls an internal method called *ExecuteCycle()*.

ExecuteCycle() provides the following functionality:

- Manages the creation and the disposal of the new thread.
- Manages timer functionality.

The three methods introduced in this section must consider the following cases (assuming that the user has SAT and application valid licenses):

- Case 1: First use of the app, the configuration is updated with **valid** parameters.
- Case 2: First use of the app, the configuration is updated with **invalid** parameters.
- Case 3: The Collector component is in the meantime between scans and the configuration is updated with **valid** parameters.
- Case 4: The Collector component is in the meantime between scans and the configuration is updated with **invalid** parameters.
- Case 5: The Collector is performing a scan operation and the configuration is updated with **valid** parameters.
- Case 6: The Collector is performing a scan operation and the configuration is updated with **invalid** parameters.
- Case 7: The Collector scan function is still executing when the *Timer* elapses again.

The Collector will execute the following actions for each of the cases above:

- Case 1: In the first use of the app, the *Timer* is not initiated, and the new *Thread* is not created. When the user updates the configuration, then *DeInit()* and *Init()* methods will be called. Configuration check passes and then *ExecuteCycle()* is executed with the input configuration. The new Thread is created, the Timer is initiated with the corresponding elapse time and the scan method is executed. If the user does not make any configuration changes, the scan function is called again when the *Timer* elapses.
- Case 2: The starting point is the same as in Case 1. When the user updates the configuration and the DeInit-Init sequence is executed, the configuration check fails. Thus, *ExecuteCycle()* method is not executed and the component will notify the application core that the input

configuration is incorrect. *Timer* and new *Thread* status remain the same as in the starting point.

- Case 3: The *Timer* is initiated and waits for the next elapse time and the new *Thread* has been disposed. When the user updates the configuration, the DeInit-Init sequence is executed. During DeInit, *Timer* is stopped and then, after Init, the configuration check passes and *ExecuteCycle()* is called with the updated configuration. The *Timer* is reinitialized, the new *Thread* is created and the scan method is executed. If the user does not make any configuration changes, the scan function is called again when the *Timer* elapses.
- Case 4: The starting point is the same as in Case 3. When the user updates the configuration, DeInit-Init sequence is executed. During *DeInit()* the current *Timer* is stopped, and then, during Init, configuration check fails. The Collector notifies the app core that the configuration is invalid. The Collector component sequence from this point is as Case 1 or Case 2.
- Case 5: The *Timer* is initiated and waiting for the next elapse time and the new *Thread* is alive. When the user updates the configuration, DeInit-Init sequence is executed. In DeInit the *Timer* is stopped and the scan (new) *Thread* is aborted. Then, *Init()* checks the configuration. The configuration is valid and *ExecuteCycle()* is called. Then, the *Timer* is reinitialized and the scan thread is created. After creating the *Thread* the scan function is executed. If the user does not make any configuration changes, the scan function is called again when the *Timer* elapses.
- Case 6: The starting point is the same as in Case 5. After the user updates the configuration, *Deinit* behaves the same way as in the previous case. Then Init() checks configuration. Because the configuration is invalid, ExecuteCycle() is not called and the component notifies the core that the configuration is not valid. The component reaction from this point is as Case 1 and Case 2.
- Case 7: The starting point is the same as in Case 5. When the *Timer* elapses, *ExecuteCycle()* is called. The method finds that the scan *Thread* is alive and performs an abortion.

2.3.1 THREAD ABORTION

As described in 2.3, in some of the cases the scan *Thread* needs to be aborted. This abortion has to be done in a **safe** way to ensure stability during runtime.

To create and manage the threads, .NET framework provides a Thread Class ([3]).

This class provides some methods and properties to create, dispose and view or set the status of the threads that the application is running. In the same documentation, Microsoft also provides some coding guidelines to make an application stable.

The thread logic that ensures safe multithreading is the following:

- *ExecuteCycle* method checks if the scan thread is alive. If it is alive, it means that scanning the network takes longer than the update cycle time. In this case, the thread is aborted (using Thread.Join the main thread is paused until the scan thread is completely aborted) and the user is notified through the UI that the current cycle time is too short.
- *DeInit* method also checks if the scan thread is alive. If that is the case, it means that the configuration is being updated while performing a scan operation. The scan thread is aborted using safe abortion methods (Tread.Join) and Collector scan operation starts again using the updated configuration parameters.

2.4 DATA PROCESSING

The main data entry point of Collector is the method *Instance.ScanDevices(out scannedDevs)* that was described in 2.2.2.2.

This method returns a data collection of type *IDeviceCollection* that contains *IDevice* interfaces. These items provide the information about the devices that is going to be processed in the application to get the status of the installed base.

However, these data types come from the SAT API. Tool-defined data types have some disadvantages:

- Project dependencies are confusing: if multiple components of an application have one common component that is not part of the .NET framework, then the application architecture is not clear enough and it is necessary to redefine it.

- The application does not have modular design: related to the previous reason, if the components are connected by dependencies data types, it is not possible to change SAT in the future without modifying code from all other components. A modular design simplifies maintenance and upgrade of. Moreover, components part of a modular design can be reused in other applications.
- Tool-defined data types are not customizable, so it is not possible to modify them to provide more complex or simpler functionalities.

Because of these reasons, it was decided to create the own application data types: *Device and DeviceCollection.*

In the Collector, objects that come from SAT are converted to these own defined data types (Figure 4):

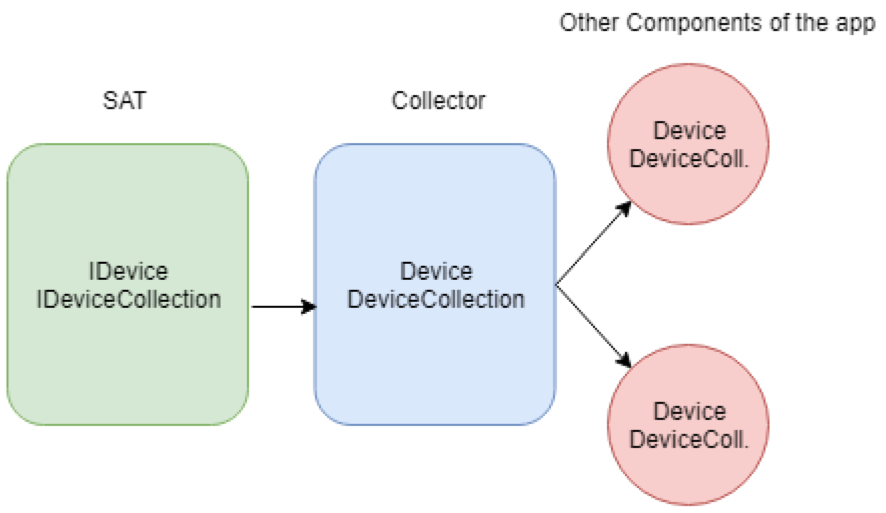

Figure 4: Device Data types layout

However, data communication from SAT could not be reliable enough for the application purposes and some information must be checked.

For this purpose, *ISnmp* interface and its implementation were created. This component uses SNMP protocol in order to get the system description of every device in the network to validate data from SAT.

2.5 COMMUNICATION PROTOCOLS

As stated in 2.4, ISnmp interface is in charge of double-checking the information from the PROFINET devices. To provide this functionality, the implemented class will receive network packets of the devices using SNMP protocol. Double-checking process is possible at this point in two ways:

- Creating a packet 'Sniffer': SAT tool uses this protocol internally while retrieving information of the devices. This way intends to receive packets from SAT during the scanning process, and then processes the packets to get the necessary data for validation. The main advantage of this way is that network traffic is optimized. Therefore, the process of gathering devices information will be faster and will not interact with the regular activity of the network.
- Creating SNMP request to the scanned devices: this means sending a SNMP request to every device that has been previously scanned by SAT and gathering the responses. Later on, processing packets and getting the information. As said previously, creating new traffic will make the process slower and may interact with the regular network activity.

Both strategies were implemented and checked if they met company security requirements and feature specification from the customer side. As personal data can be gathered from the network, it is necessary to demonstrate that the software shall only get packets of SNMP protocol.

Before going in depth with both implementations, the SNMP and underlying protocols are explained below.

2.5.1 SNMP PROTOCOL

SNMP is a popular protocol for network management. It is used to manage networks with hundreds or thousands of nodes. In PROFINET Networks, it is used for collecting information from devices and configuring them on an Internet Network.

The protocol is encapsulated in UDP and IP (see Figure 5). Because the protocol is encapsulated in UDP, it is **connectionless**, meaning that there is no control over the transport layer. Devices can receive the same packet more than once or packets can be lost in the network without noticing.

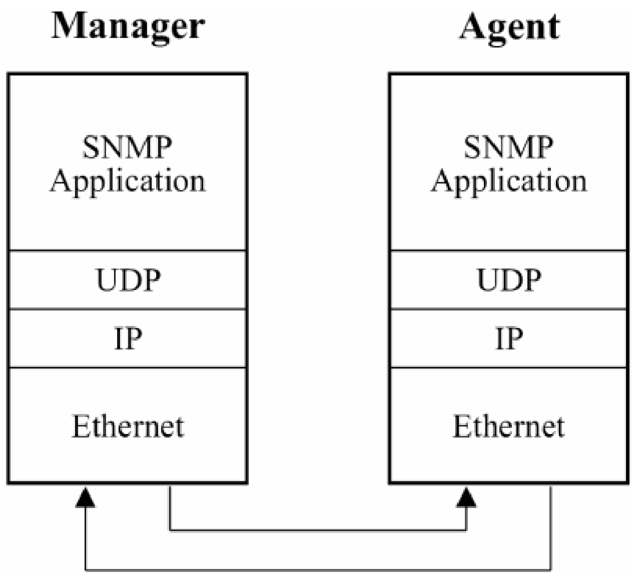

Figure 5: SNMP Protocol Encapsulation Overview.
Source: http://www.industrialethernetu.com/courses/204_1.htm

In the market there are some tools to manage networks. These tools provide features to monitor performance, audit network usage, and detect issues or inappropriate access. The aim of SNMP is to be present in as many devices as possible, having minimal operating requirements and run when other protocols do not.

A scheme of SNMP layout can be seen in Figure 6:

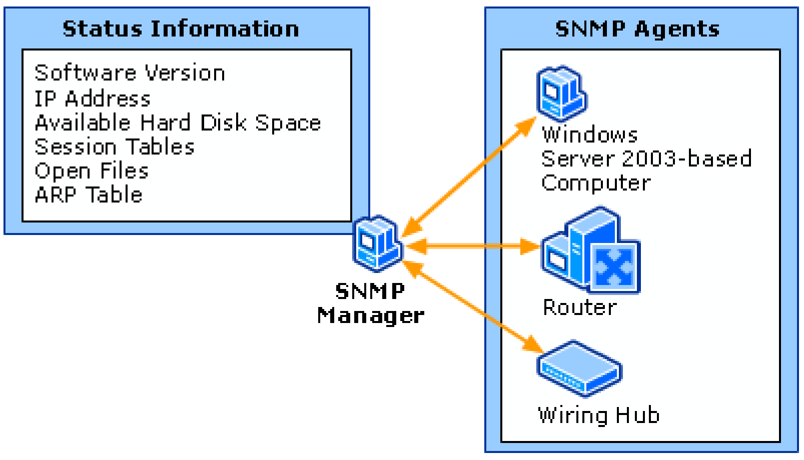

Figure 6: SNMP Network Layout

As seen in Figure 6, the layout is based in two participants: the manager and the agents.

Agents can communicate with the manager, but agents do not communicate between them. In non-industrial networks, SNMP is used with printers, routers, hubs, computers…

In the case of the current project, the protocol is going to be used with PROFINET devices. The computer of the customer is the manager, and the connected PROFINET devices behave as Agents.

The architecture of SNMP offers a solution to the network management problem in terms of:

- The scope of the management information communicated by the protocol: the information is represented by object types that are defined in Internet-standard MIB or defined according to Internet-standard SMI.
- Because of the previous term, the representation of the information communicated by the protocol is standard.
- Operations on management information supported by the protocol: management functions are based on retrieval or alteration of variables. Thus, it limits the amount of management functions to two: **read** and **write** values.
- The form and meaning of references to management information: SNMP objects are referenced by unique names so there is no possibility that MIB can resolve multiple instances of that type. Also, the protocol is intended to solve only variable names that are present in the specific version of the protocol.

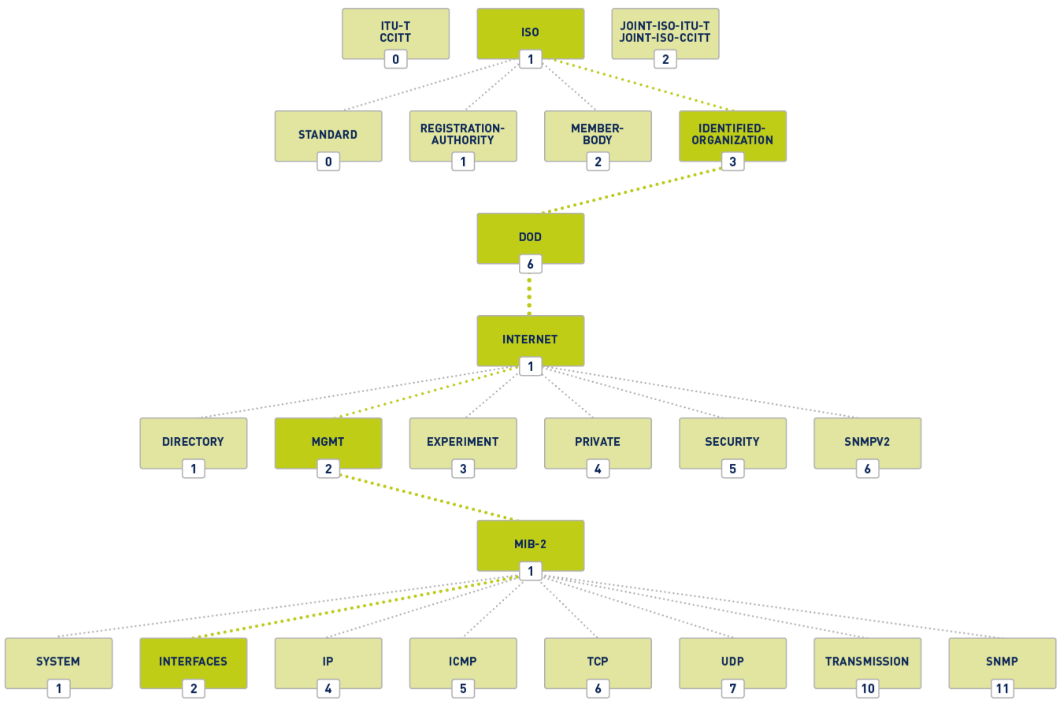

Figure 7: OID tree.

Source: [4]

In the case of the present project, Version 1 of SNMP protocol is going to be used. This implementation of the protocol has the following characteristics:

- SNMP manager receives the responses from devices on the port 161. The manager can send to the devices the requests from any available port.
- The MIBs in the first version are highly structured, therefore easier to analyze. To access MIBs, there are object identifiers (OIDs) that uniquely identify [4] managed objects in the MIB. The MIB is organized hierarchically and can be depicted as a tree with different levels from the root to the single leaves. Each OID has an address that follows the levels of the OID tree. This hierarchy can be seen in Figure 7. For the case of this project, system description is the OID that is going to be used. Its OID is 1.3.6.1.2.1.1/0.
- It presents Community Strings. Community Strings allow to authenticate access to MIB objects, acting as embedded passwords in the packet [5]. This is the only security mechanism that is present in this version. Newer versions of SNMP implement encryption techniques to protect data.

The packets present the structure shown in Figure 8:

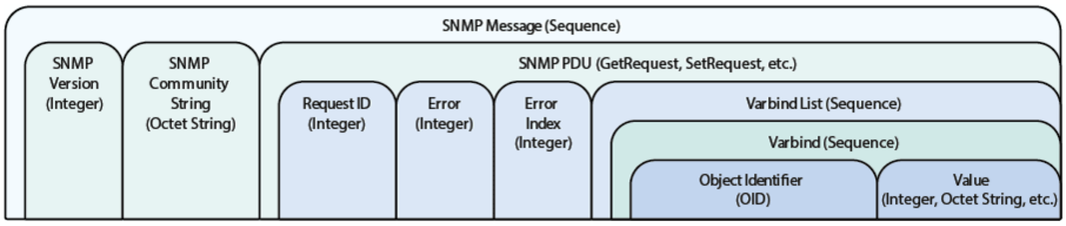

Figure 8: SNMP Packet format.

Source: https://www.rane.com/note161.html

This structure is used to analyze the packets received from devices. The contents of these packets are processed and compared with data received from SAT to make the validation.

2.5.2 UDP PROTOCOL

As introduced in 2.5.1, SNMP protocol is encapsulated into UDP and IP.

This protocol provides a procedure to send messages as datagrams with a minimum protocol mechanism.

In contrast with TCP [6], which checks and resends lost packets and makes a reassemble into the correct sequence; UDP just sends the packets. This results in lower bandwidth overhead and latency. On the other hand, packets can be lost or received out of order.

2.5.3 IP

IP protocol version 4 is included in the PROFINET standard, and it is the version that it is going to be considered in the current thesis because the next version of the Protocol (IPv6) is predicted not to be included in the standard soon. Nevertheless, both versions will coexist if the protocol is released soon.

The packet definition can be seen in Figure 9 :

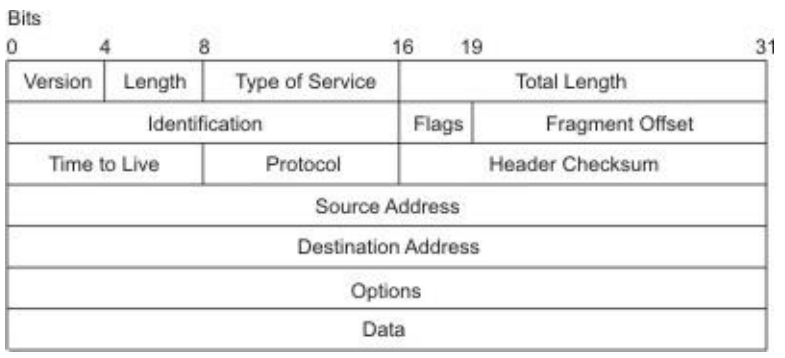

Figure 9: IP Packet Description

This description was used during Sniffer development to decode the raw packet and get the final data (SNMP data).

2.5.4 ‘SNIFFER’ IMPLEMENTATION

As a first approach, the ‘Sniffer’ option was chosen. The main reason to start with this option was because SNMP requests are created by SAT during the scanning process. Therefore, it is not necessary to generate more network traffic.

For the SNMP communication, the following C# libraries were studied:

2.5.4.1 SnmpSharpNet

This Open Source library [7] is written in C# and is compatible with SNMP protocol versions 1, 2 and 3.

It supports SNMP operations (highlighted in bold those necessary for the project) such as:

- **Get.**
- Get-Next.
- Get-Bulk.
- Set.
- **Response.**
- Report.
- Trap.

The library is fully self-contained, meaning that it only depends on .NET framework and license type is GNU Lesser General Public License, making it suitable for the project.

However, the library presents some disadvantages that made it unsuitable for the project:

1. Beta version: the project is currently under a beta version [8] (the most recent version is 0.9.4) dated 2014. The use of a non-released product does not meet the **quality** requirements for this application.
2. Lack of improvement: because the most recent version is dated 2014, its library has not been improved regularly, so there is no possibility to have the first release on time.
3. Features do not fit requirements: the library allows to send and receive requests from a specific end-point, but it is not possible to make a "listener" that gathers the received information in a buffer for further processing.

2.5.4.2 SNMP Library

Available in GitHub as *sharpsnmp* project, the library is compatible with C#, and supports the following SNMP operations:

- **Get** (includes get-response).
- Set.
- Get-Next.
- Get-Bulk.
- Walk.
- Trap.
- Inform.

The library is self-contained, and it can be installed easily through NuGet package manager in Visual Studio 2015.

This library has released the version 10.0.4 on February of the present year. There are two possibilities to use it:

- Free version.
- Pro version: includes a MIB compiler. This is a GUI [9] that helps to load MIB files and write them to make configuration files for devices

compatible with SNMP protocol. This version is intended to be used by professionals and enterprises which need consulting and support services.

As in the previous case, this library was finally discarded because features did not fit with the project requirements. It is not possible to implement an algorithm that gathers data while SAT is scanning the network.

2.5.4.3 SocketClass (.NET framework)

This class [10] implements the Berkeley Socket interface. It is an interface to use network communication capabilities.

A socket is an abstraction provided to an application programmer to send or receive data to another process. Data can be communicated between processes in the same machine or different machines. As seen in Table 3, socket interface is located [11] between Application layer and transport layer:

Application
API
Transport
Network
Link
Physical

Table 3: Location of API in the communication layers

Sockets are also known as a half-association (3-tuple) that contains:

- Protocol, local IP address, local port number.
- Protocol, remote IP address, remote port number.

The most common socket types are:

- Stream sockets: provide reliable, two-way, connection-oriented communication streams. This kind of Socket uses Transport Control Protocol (TCP) as underlying transport layer.

- **Datagram** sockets: provide connectionless, unreliable service, used for packet-by-packet transfer of information. This Socket uses UDP as underlying transport layer and is the most suitable for the project.
- Raw sockets: this type of socket provides access to internal network protocols and interfaces. It allows an application to have direct access to lower-level communication protocols. Since the protocols are exposed in RFCs, it is possible to parse them and get the SNMP information useful for the project.

Microsoft .NET framework provides a class named "Socket" that holds Berkeley API. The full documentation of the class can be found at [10]. However, in this section the most remarkable methods, properties and challenges will be described:

The socket constructor used was:

```
public Socket(AddressFamily aF,SocketType sT,ProtocolType pT)
```

- *AddressFamily* enumeration specifies the type of address that has been used. There are numerous types such as AppleTalk, IPv4, IPv6 or Unix type. As mentioned in 2.5.3, IPv4 is the underlying protocol of this application.
- *Socket Type* enumeration specifies the socket type that the programmer wants to use. Possible values are DGRAM, RAW and STREAM (as defined in Berkeley API but there are others such as Seqpacket and Unknown.
- *ProtocolType* specifies the protocol that the socket is aimed for. There are many configurations including IPv4, IPv6, TCP, UDP, ICMP…

The three parameters can be configured. However, the combination of these three must be consistent among them. Otherwise, the constructor raises an **exception**. An example of an invalid configuration would be calling the constructor with DGRAM as SocketType and using TCP as the Protocol type.

Regarding properties, some of them are useful for the entire development:

- *Available*: gets the amount of data that has been received from the network and is available to read.
- *EnableBroadcast*: gets or sets a Boolean that specifies if the socket can communicate through broadcast packets.

- *ExclusiveAddressUse*: property that implies that the Socket only allows one process to bind a port.

The most significant method for the development was *IOControl* because it sets low-level operating modes for the Socket.

Finally, this way to implement SNMP communication in the “Sniffer” was selected, because:

- ✓ It is integrated in .NET framework. Therefore, there are no issues regarding licensing, functionality or support.
- ✓ This class provides more functionality: the class encompasses much more functionality than needed for the project, because it is a generic library. As seen through, other libraries have been built on top of .NET framework.
- ✓ The development process will be slower due to learning tasks, but if the project requirements change, it is possible to implement additional features regarding SNMP protocol or other protocols.

2.5.4.4 Development

From the libraries presented in 2.5.4.1, 2.5.4.2 and 2.5.4.3, the *SocketClass* was finally selected to develop the sniffer.

As mentioned in 2.5.4, the goal of ‘Sniffer’ is to capture the SNMP communication that is present while SAT is scanning devices.

A scheme of the process is shown in Figure 10. It is composed of the following phases:

1. Timer elapses, and the Collector component checks the application configuration. If the configuration is fine, it proceeds to start the scan process.
2. Prior to call SAT scan function (described in 2.2.2.2) the socket object is configured. The configuration is based on a 3-tuple as described in 2.5.4.3:
    - Protocol: UDP.
    - Local IP Address: the network adapter IP address.
    - Local port number: 161 (as commented in 2.5.1).
    - *IOControl* properties to configure the socket as a listener.

3. Start the socket listening to packets: the socket captures all packets received on port 161 and UDP protocol. Packet information is stored in a byte array buffer for further processing. This processing is made at the end of the SAT scan process.
4. Call *Instance.ScanDevices()* method. This method blocks the execution of the rest of the code. Because of that, the 'Sniffer' and SAT scan method are placed in different threads. SAT belongs to the timer thread and a new thread is created to execute the 'Sniffer'.
5. When SAT finishes scanning the network, the listener is stopped and packets stored in the buffer are processed. The timer is reinitialized and the application exits Collector component until the timer is elapsed again.

This process was implemented successfully during application development. However, during testing it was discovered that the 'Sniffer' was capturing packets that did not belong to any PROFINET device. Using Wireshark and analyzing packets it was possible to notice that the sniffer was capturing packets from other applications that were not part of the scope of the present application.

Several configurations were applied to the socket object in order to try to solve this issue. However, it could not be found a way to capture packets exclusively from SAT.

This issue made not possible to use this method for validation, so other validation methodology was implemented. This methodology is described in 2.5.5.

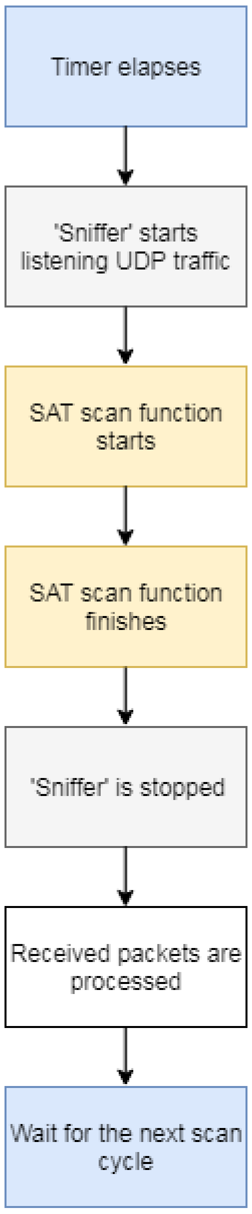
Timer elapses
'Sniffer' starts listening UDP traffic
SAT scan function starts
SAT scan function finishes
'Sniffer' is stopped
Received packets are processed
Wait for the next scan cycle

Figure 10: Sniffer process diagram

2.5.5 ACTUAL IMPLEMENTATION

This chapter will explain the actual implementation of *ISnmp* interface, creating more traffic on the network.

As explained in 2.5.4.4, the 'Sniffer' implementation was not valid for the project. To provide the same functionality as the 'Sniffer', the project took another approach.

As mentioned in 2.5, the idea is to send SNMP requests and get the responses from the devices that are previously scanned by SAT.

SAT provides the IP addresses of the devices that have been scanned. Thus, it is possible to communicate with them using their IP addresses as endpoints for the socket configuration.

The scheme of this implementation can be seen in Figure 11:

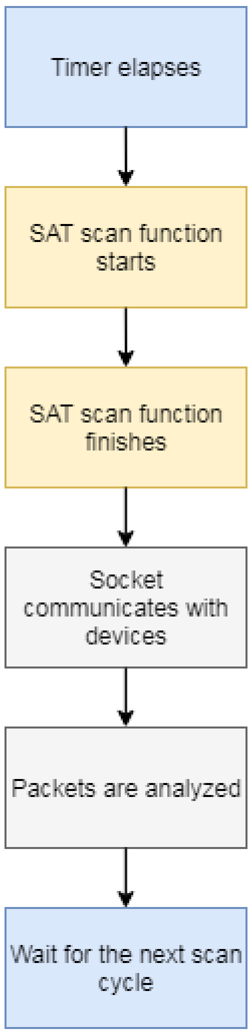
Timer elapses
SAT scan function starts
SAT scan function finishes
Socket communicates with devices
Packets are analyzed
Wait for the next scan cycle

Figure 11: Actual Implementation Diagram

The actual implementation encompasses the following steps:

1. Timer is elapsed to scan devices from the PROFINET network.
2. SAT scan function is executed, getting information and the IP addresses of the installed base.
3. IP addresses are validated to proceed later with the Socket communication.
4. Socket object is configured to send a SNMP request and get the response from a particular device. The tuple configuration is the following for sending the SNMP request:
    - Protocol: UDP.
    - Remote IP address: the IP of the desired device, given by SAT.
    - Remote Port: 161.

   It is not necessary to configure another socket object for the response; the socket has the functionality to receive responses from the endpoint of the communication. Instead of gathering all packets in a single buffer, each packet is stored in a separate byte array.
5. The information gathered from Socket communication is processed. Confidentiality requirements of the project do not allow to fully describe the whole packet process, so a brief description of this process is provided in 2.5.6.
6. Device data is validated and the database with the installed base information is updated.

2.5.6 STRING DATA PROCESSING

Response packets from SNMP are stored in buffers as commented in 2.5.5. These buffers hold the raw packet data from UDP protocol (IP protocol heading is already processed by the socket).

The process to get the information useful for the application is the following:

1. Isolate SNMP object: this means analyze the packet following the packet structure of UDP and SNMP protocol. This analysis gives information about the communication and the object that is being requested. As

mentioned in 2.5.1, **system description** is the used OID for this application.

2. Process SNMP object: in this phase the object is converted from raw data to strings. Then, strings are analyzed to validate devices.

String analysis can be made using two different methodologies:

1. Analyze strings as arrays: this method utilizes array properties to analyze strings as arrays of characters.
2. Use regular expressions: regular expressions are patterns [12] used to find matches in an input text.

For the present thesis, regular expressions methodology was used. The reasons to select regular expressions were:

- Regular expressions allow the developer to focus on the functionality rather than worrying about arrays issues.
- Patterns are easy to describe, and it is possible to implement changes in them quickly.
- There is a Regular Expression class provided by .NET framework.
- Performance is not affected significantly because there is not a lot of text to be processed.

2.6 FAULT AND EXCEPTION HANDLING

The application is part of the industrial software portfolio of the company. Because of that, it must be responsive in any situation and handle all possibilities of input data. In the case of the Collector component, there are some situations that could make the application crash:

- Initialize the Collector more than once: the Collector thread is designed to be executed every time the timer elapses. Creating more than one Collector thread would result in unexpected behavior.
- Wrong configuration input: if the user enters a configuration not suitable for the Collector or there was a failure in the localhost communication, the Collector cannot execute the scanning task.
- Socket exception: when validating the data, it is possible that the device does not have a valid network configuration, making SNMP communication impossible.

Some of these situations are solved by using input validation. However, there are some scenarios impossible to prevent. Therefore, the use of exceptions is necessary to avoid application crashing.

For this application, C# **exceptions** are used. Exceptions occur when the code encounters an unexpected scenario. These situations block the execution of the current method or function. From this point, the code tries to find a piece of code that handles the exception. If it cannot find it, the code will go backwards through the call stack until it finds the handling. If the exception is not handled in any part of the code, then the C# runtime will stop the execution of the application.

To handle exceptions in C#, there are three main statements:

- *Try.*
- *Catch.*
- *Finally*.

It is possible to use them all or use *try* and *catch. Try* statement allows encapsulating code that can raise exceptions, while *catch* statements provide handling mechanisms to these exceptions. It might happen that *catch* statements provide handling to some exceptions but not all of them. If a specific exception that is not being handled by the *catch* statement is raised, the code will go backwards through the call stack in order to find a *catch* statement that handles this exception.

Finally statements provide a functionality to take an action if any of the caught exceptions occur. Usually these statements are used to close files or safely close communications with databases.

Exceptions are represented in the C# language as objects. The C# system class *Exception* provides the functionality to log error messages and get information about the failure that is present. However, it is a good practice to create specific exceptions for the application. This gives the advantage of specifying the error and create customized handling depending on the failure.

To create specific exceptions, a class derived from *System.Exception* was created (Figure 12):

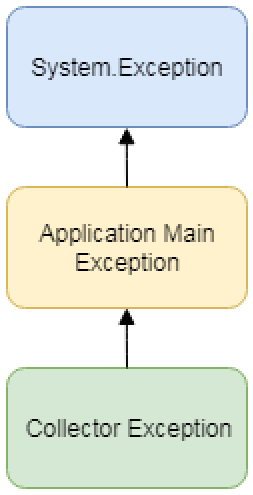

Figure 12: Exception UML diagram

To ensure that all possible scenarios are covered, two techniques were applied:

- Code review: a person or group of people who do not develop a software component reviews the whole code. This technique makes easier to find bugs and solve issues as unhandled exceptions in the code.
- Executing static code analysis: Visual Studio provides a feature that analyzes the code and gives warnings to the developer about coding guidelines that are not present in the code. The developer

can choose which guidelines wants to be included in the analysis. Regarding exceptions, code analysis can warn about catching generic exceptions.

Chapter 3 TESTING FRAMEWORK

3.1 OVERVIEW

To deliver software that is reliable and safe for the customers, the application will have to behave as expected in every scenario. For this reason, a testing framework is necessary.

A testing framework includes several testing methodologies. For this project, the following methodologies were used:

- Unit testing: to test functions, methods and other small parts of code.
- Module testing: intended to check single components of the application.
- Integration testing: to check the whole application, focusing on the integration of the different components.

Figure 13 shows the hierarchy of the different testing methods. The Integration testing method used in this project combines the *Sub System* and *System Testing* seen in the figure.

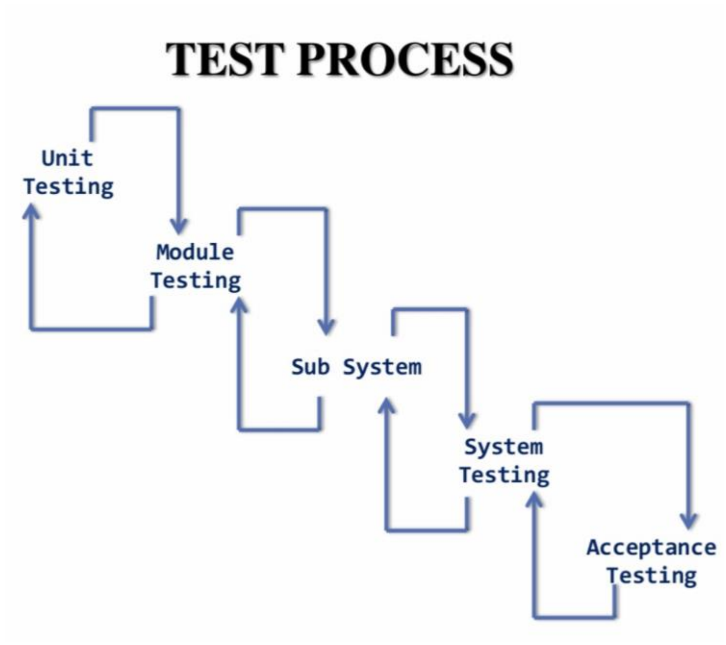

Figure 13: Testing frameworks layout.

Source: https://www.slideshare.net/deepaksharmasharma9615/3software-testing

The basic idea of testing is shown in Figure 14. Basically, testing starts by providing certain input data to the code unit that the developer wants to test. The code unit can be a single method, a class, an app module or the whole application.

Once the code is executed, output data is compared to the expected output, and if they are equal, then the test scenario is successful.

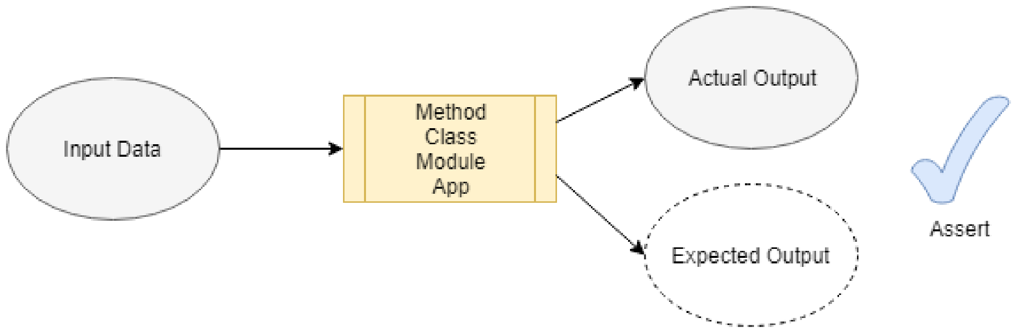

Figure 14: Testing Diagram

Input data scenarios will have to provide different conditions to ensure that the following capabilities are implemented in the software:

- *Resource management:* the application must deal with memory and object management, creating and disposing the necessary instances and threads. This capability is mostly taken care by C# runtime, but there are some cases (Threading and accessing databases) where it is required to manually manage them, therefore this must be checked during testing.
- *Functionality:* during testing, the different functionalities to be delivered by the application are checked.
- *Security:* depending on the degree of confidentiality requested on data involved, the application will have to deal with security issues that may affect the application. Scenarios that involve security treads may be considered during testing phase.
- *Fault handling:* the application has to consider cases in which components or communication with other parties fail. This includes **exception** handling, checking data integrity and ensuring that the function calls are executed in the correct sequence.

The methodology to perform these tests can be manual, e.g. with developer interaction to perform every scenario; or automated, by coding the different case scenarios.

For the project of this thesis, automated testing approach is used. In the following sections, testing framework and integration testing are fully described.

3.2 TESTING FRAMEWORK

3.2.1 TEST TOOLS

As explained in 3.1, the automated testing approach is used for the current project. To automate tests, it is necessary to have a testing framework that provides capabilities to write, execute and analyze test cases.

For the current project, Visual Studio 2015 Professional testing framework is used. The capabilities that this edition provides are displayed in Table 4 and compared to Enterprise Edition of the same IDE.

	VS Professional	VS Enterprise
Intellitest		✓
Microsoft Fakes		✓
Code Coverage		✓
Test case management		✓
Unit Testing	✓	✓

Table 4:VS Test features comparison

Extracted from: [13]

The following explains the different features shown in Table 4:

- *Intellitest:* this component [14] creates a suite of tests for the code automatically and can generate data input combinations to cover all cases.
- *Microsoft Fakes:* this feature [15] helps to isolate the code that is being tested by replacing other parts of the application with *stubs* or *shims.*
    - *Stubs*: they replace a class with a substitute that implements the same interface. Using stubs require that application components are connected by using interfaces exclusively. This design approach was used during development and is fully described in section 3.3 of this document.
    - *Shims*: modifies the compiled code of the application at runtime so that it fakes method calls by running the shim code that the test provides.
- *Code Coverage*: is used to determine the percentage [16] of the code that is being tested and can be used a progress measure of the test phase of a software solution.
- *Test case management*: this is a Visual Studio extension package [17] that gives capabilities to establish a test plan for a solution.
- *Unit testing*: this feature allows to write and run unit tests for a certain method, component or a whole application.

3.2.2 CONTINUOUS INTEGRATION TOOL

As mentioned previously, Visual Studio 2015 Professional testing framework is going to be used in this project. To ease continuous integration [18] process, it is possible to port the framework to a server that builds and tests the application in every code update using an application called TeamCity.

TeamCity [19] allows a team to track changes during development and provides building capabilities every time that new code is pushed into the server. These capabilities include:

- Code compilation.
- Run automated tests.
- Deploy the application.

A report is generated in every built, so it is possible to check changes and integrate features faster.

3.3 CODE DESIGN

As mentioned in 3.1, testing is a critical phase of the project. To make this phase successful, it is vital that the code presents a defined architecture that makes it easier to test. In this project, the code was designed using the following principles:

- Clear architecture: it is important to isolate the different functionalities and responsibilities of the classes and interfaces of the application. This guideline was also applied to the internal design of the Collector component of the present document, by splitting tasks into different classes and interfaces (see Chapter 2). This architecture makes debugging easier and faster.
- Interfacing: communicating application parts with interfaces provides isolation and prevents developers from using objects or methods that

should not be called from a certain component. In addition to that, it is possible to wrap components during phase using a *fake* tool or a mocked class. The tools that were studied for the present project are described in Section 3.3.1.

In the case of the present project, *interfacing* is crucial because there are certain components that need external communication, therefore, it is not possible to test them on the server:

- SAT implementation: this part needs to establish communication to PROFINET devices to gather information from them.
- *ISnmp* implementation class: this class also needs to communicate with the PROFINET devices using the named protocol.

To make the code of these two modules testable, the code design evolved to a more complex architecture. This architecture (Figure 15) presents an intermediate interface that provides a façade between the application and the external communication. In addition to that, data is packed into objects that are part of the .NET framework, letting custom objects which belong to other dependencies (such as SAT API) be part only of the interface implementation.

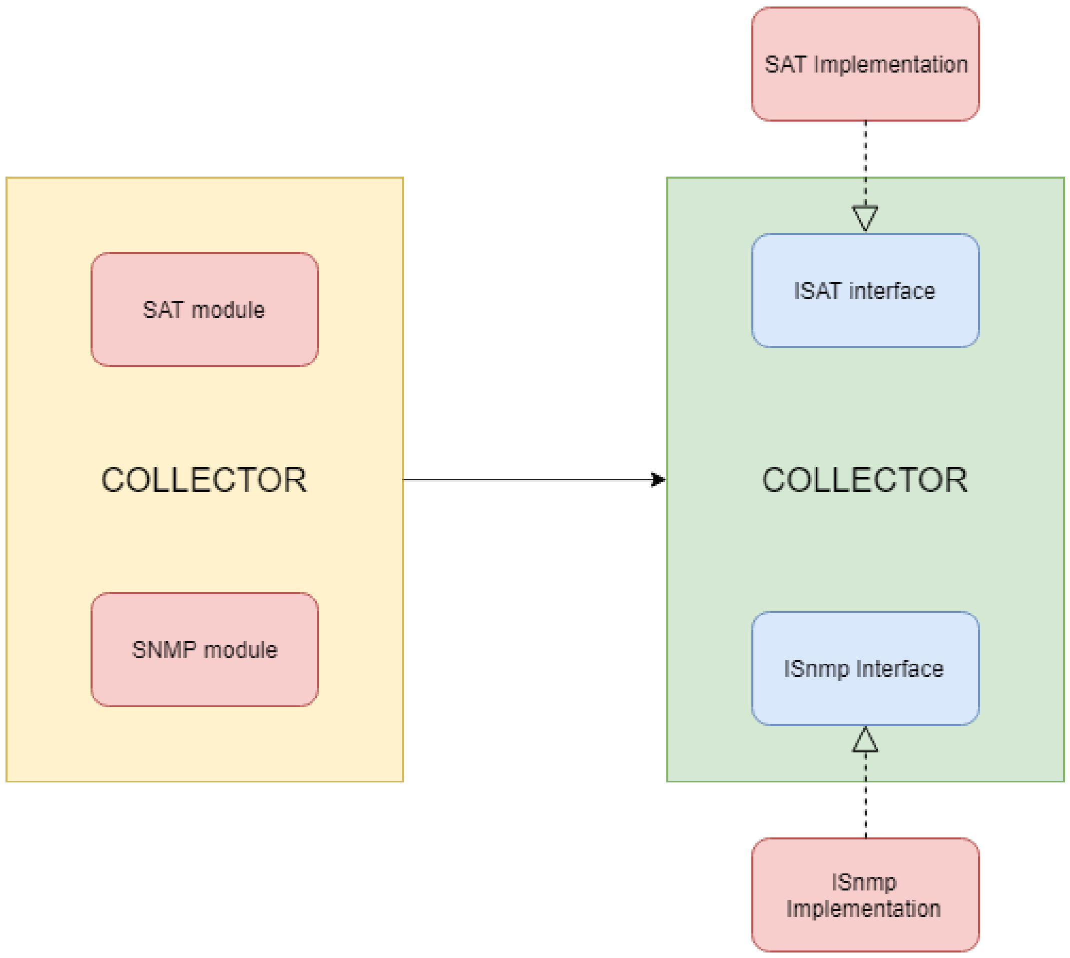

Figure 15: Code Architecture Evolution

3.3.1 FAKING AND MOCKING CLASSES

In 3.3 it was explained that wrapping components to focus testing to an isolated part is possible by using a faking framework. The IDE used in this project has no capabilities to fake objects according to Table 5. Therefore, other free options from Visual Studio marketplace were studied. The best option that was studied is called FakeItEasy.

FakeItEasy has some capabilities suitable for testing purposes. It can fake several types of objects and function calls.

Regarding types [20], the package can fake:

- Interfaces.

- Classes that fit these conditions:
  - Not sealed.
  - Not static.
  - Have at least one public or protected constructor whose arguments the package can construct or get.
  - Delegates.

Regarding function calls [21], the package can configure:

- Function calls
- Getters.
- Setters.
- Fake return values.
- `ref` and `out` parameters assignment.

Nevertheless, as mentioned in 3.3, the code was designed to split the different components and dependencies with interfaces. According to FakeItEasy features, the package can fake interfaces and was used as a first approach to deal with actual communication issues. However, trying to fake SAT custom objects presented problems. For this reason, FakeItEasy approach was discarded.

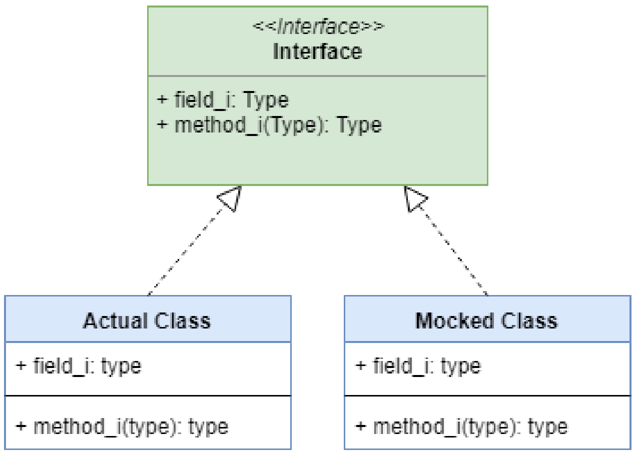

Figure 16: Mocking interface implementation

The approach that was finally chosen was **mocking** the interface implementation. This concept is described in Figure 16. Basically, it is necessary to create another implementation for the interfaces described in Section 3.3.

These interface implementations consist basically of a basic encapsulation of their methods to interchange data in standardized objects (objects that are part of the .NET framework). This wrapping is shown in Figure 17:

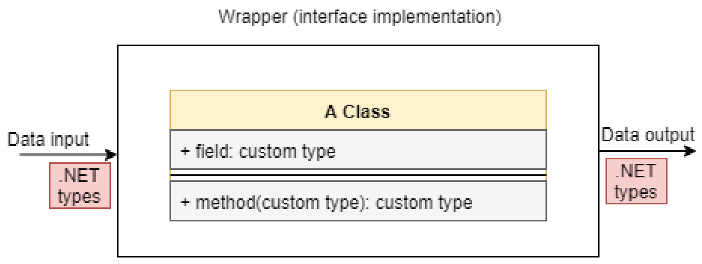
Wrapper (interface implementation)
A Class
+ field: custom type
+ method(custom type): custom type
Data input
.NET types
Data output
.NET types

Figure 17: Type encapsulation

3.4 INTEGRATION TESTING

The tests that are performed on the server are part of **integration testing**. This test methodology helps the developers to find bugs and make the necessary changes in order to integrate app components.

The following subsection describes the structure of the tests that were performed and the strategy that was followed to make a reliable integration testing framework.

3.4.1 TESTS STRUCTURE

To test the application as an End to End test, the tests are designed using the following structure:

1. Make a request via REST APIs: simulating the user UI, the test starts using REST APIs as the user does when navigating through the web interface.

2. Read/Update information: the REST request can either retrieve information or update certain data.
3. Make assertions: if the REST request updates information, it is necessary to retrieve again the updated data fields and ensure if the update has been successful. If the test request was retrieving information, this has to be compared with the **mocked** data that has been hardcoded previously.

To ensure that the tests are designed correctly, they must:

- Provide a defined starting point of the application: tests must start in a known and defined environment to get the same results with the same inputs.
- Test as many conditions as possible to increase code coverage.
- Make logical assertions e.g. compare data that is being modified during the test.

3.5 COLLECTOR MODULE TESTING

Sections 3.3 and 3.4 described the code design and the procedures to test the application on the server using TeamCity. However, this approach excludes the actual device communication with SAT and UDP sockets.

To test these dependencies, it is necessary to implement another type of test called module testing. This test approach just performs test of a specific component of the application. In this case, the test is applied exclusively to the Collector component as shown in Figure 18.

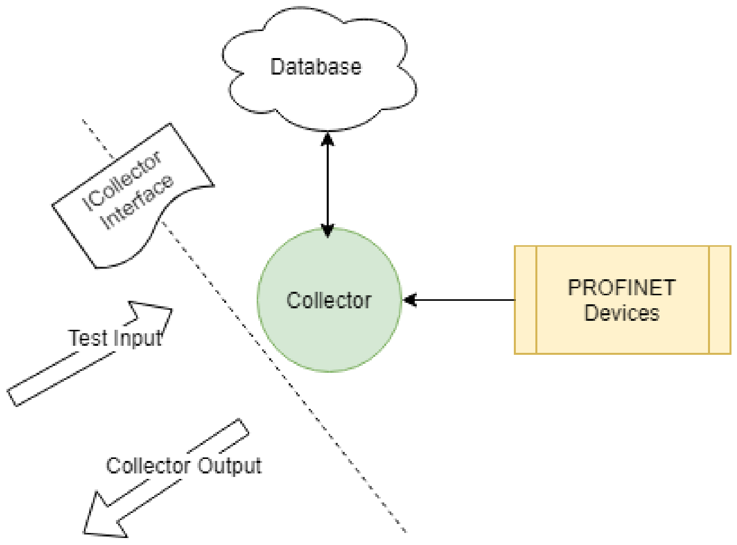

Figure 18: Module Testing Overview

The testing is made through the *ICollector* interface. This methodology allows to test:

- Functionality: to check the different features that this component must offer according to the application architecture.
- Fault handling: to ensure that the component works in every possible scenario.
- Responsiveness: check times of the scan process.
- Communication: to ensure that the different communication protocols are well configured and work in a test network.

-

Chapter 4 APPLICATION DEPLOYMENT

4.1 OVERVIEW

After application development, the next step is to deliver the application to the final customers. To do that, it is necessary to create an installer solution.

The features that the installer must provide are:

- Check the OS version: the installer must install the application in the operating systems Windows 7 or Windows 10. All other OS versions are not allowed.
- Check for requirements: the installer can run only if certain software tools are installed in the customer´s computer. In the case of this project, the prerequisites are:
    - .NET version 4.6.2.
    - SAT 3.1.
    - ALM: the lower required version is 6.0.
- Copy Files: the files to run the application need to be copied in certain locations of the computer:
    - [ProgramFiles]: libraries and files that are not going to be changed during runtime are copied here. The reason why only "static" files must be copied in this folder is because in order to modify, add or remove files in this location, administration permissions are needed. The current application must run without admin permissions.
    - [ProgramData]: the local database of the app must be copied here because it is going to be modified during runtime.
- Write registry keys: during installation, it is needed to write registry keys for:

- Application configuration: some of the configuration parameters of the app are stored there, in order to save the configuration when the customer shuts down the computer or the app.
- Installation: to store the installed version of the application. This can be used for future installers that might check the current version of the installed application.

- Execute post installation programs: the current application will be delivered as a Windows Service. The application as Service will be installed after the files are copied in the proper locations. To do that, it is necessary to execute an *.exe* file.
- Reboot the machine: because of the previous point, the machine has to be rebooted in order to start running the service.

To develop an installer solution that can comply with these technical features, different installer development frameworks are going to be studied in 4.2. After that, two implementations of the installer are going to be described in detail.

4.2 TOOL SELECTION

To deploy the application satisfying the requirements defined in 4.1, different tools are going to be analyzed regarding features, ease to use and licensing terms. For this thesis, the tools that are going to be analyzed are:

1. Microsoft Visual Studio 2015 Installer Projects extension: this is a tool available in the Visual Studio *marketplace.* It is an extension of the IDE.
2. Inno Setup: this is a third-party tool that supports from Windows 2000 on. It provides full support for 32-bit application and an extensive support for 64-bit.
3. WiX Toolset: this software has two components:

a. Toolset build tools.
b. Visual Studio WiX extension.

4. InstallShield Limited Edition: this is a software integrated in Visual Studio to create setup packages.

Tool versions are presented in Table 5:

Tool Name	Version	System Description
MS VS 2015 Installer Projects Extension	2.1	Microsoft extension to Visual Studio 2015 to create installer packages
Inno Setup	5.5.9 (a)	Tool that generates installers from script files
WiX Toolset	3.11	Tool that generates installer solutions from *xml* files or from VS UI
InstallShield Limited Edition		Limited InstallShield capabilities

Table 5: Studied Tools version

4.2.1 ANALYSIS OF ALTERNATIVES

In Table 6, a simplified table with the features that are required for this project is presented. Each tool has been studied for this application and some have been tried to check for feasibility.

	Visual Studio Ext.	Inno Setup	WiX	InstallShield Limited Ed.
License	Not commercial use	Commercial use-free	MIT	Not comercial use
Firewall exceptions	Separate batch file	Yes	Yes	No
Learning curve	Easy	Medium	Hard	Easy
Learning resources	Few	Stack Ov. & Docs	Stack Ov. & Docs	Stack Ov. & YouTube
Customizable installer	Few	Custom pages	All	Few
Custom GUI capabilities	Few	Few	All	Few
VS integration	Yes	No	Yes	Yes
R/W Registry capabilities	No	Yes	Yes	Yes
Preinstall capabilities	Yes	Yes	Yes	N/A
Postinstall capabilities	Yes	Yes	Yes	N/A

Table 6: Tool feature comparison

As presented in Table 6, **Visual Studio 2015 Projects Extension** is not valid for the scope of this project because it cannot be used for commercial purposes. However, the tool provides some remarkable features:

1. It is simple to use: it is good as a first step to get familiar with installer environments.

2. Following the previous statement, the tool is fully integrated in Visual Studio IDE. The tool provides a *wizard* that helps the user to configure the main settings of the installer application such as:
    a. Type of application (Windows or Web).
    b. Type of redistributable package.
    c. Dependencies and files to be deployed.
    d. Changes in the registry.
    e. Changing the user interface of the *.msi* installer.
    f. Checking version of installed .NET framework and providing the download link if necessary.
    g. Feature to give the user the option to install the software to all users of the machine.

Nevertheless, this tool presents a main drawback: the full set of capabilities is not clear, because the extension has a lack of documentation. The user can only learn by trial and error.

WiX Toolset encompasses a set of utilities that build Windows installation packages from XML source code. It is an open source project, originally developed by Microsoft and currently maintained by Rob Menshing. In fact, the tool was used to develop the installer for the Microsoft Office suite.

To work with the tools, it is possible to use a bash environment or the UI that is available as a Visual Studio extension.

The main features that this toolset presents are listed below:

- Declarative approach: using tagged language XML.
- Unrestricted access to Windows Installer functionality.
- Source Code instead of GUI-Based assembly of information.
- Complete integration with application development: it is possible to develop the application and the installer at the same time.
- Support for team development, both in-house and third-party software.

InstallShield Limited Edition is a Visual Studio extension to build *.msi* installer packages. It is available for all Visual Studio versions except Community Editions. It has plenty of capabilities [22] that include:

- Customize application description information.
- Check .NET and Internet Explorer version.
- Check for required Software.
- Manage files to deliver from folders or from Visual Studio released code.
- Registry capabilities in 32 and 64-bit registry. It is possible to:
    - Add keys.
    - Change values of existing keys.
- Customize application icon and create an entry in the Start menu.
- Customize dialogs as License Agreement, Users installation, installation path and launching the application after installation is completed.
- Customize environmental variables.
- Perform installation of IIS server.
- Capabilities with Windows Services.
- Customize UI: it is possible to modify the default images that appear during installation.
- Advanced configuration: the installation capabilities can be modified from some INI files.
- MSI deployment.

However, this tool has some drawbacks that made it unsuitable for the current app:

- Architecture of installation cannot be modified: because this software is a Limited Edition, the installer architecture is not changeable.
- Licensing. In the End-User License Agreement (EULA) [23], it is described for the Visual Studio Extension that this tool may not be used for commercial purposes.

From the tools that have been analyzed in this chapter, Inno Setup was selected to deploy the software. In 4.3, the internals of this tool in order to get the application successfully deployed are described.

4.3 INNO SETUP APPROACH

InnoSetup was the first approach to develop the installer solution as discussed in 4.2.1.

This tool uses a script to create the installer solution. The script is created and compiled by Inno Setup.

This script is divided into the following sections:

[Setup]

This section contains settings that affect the installer and the uninstaller and are required for every installation. There are mandatory properties like application name and version, and this section can also configure the user interface and more detailed product information.

[Code]

This section specifies a Pascal script. This script can customize the internals of the setup process. For that, Inno Setup provides support functions and classes that can be easily customized.

[Types]

It is an optional section. It allows to define the setups types that are shown when selecting components during installation e.g. minimal, custom, full. It is possible to define additional, developer created types.

[Components]

This optional section configures the components that are shown in the component selection window during setup.

[Tasks]

This section defines the user-customizable tasks that the setup will perform during installation. These tasks can be selected by the user in a wizard page through checkboxes and radio buttons.

[Files]

This section defines any files to install on the user´s system.

[Icons]

This section defines the shortcuts to the application. Shortcuts can be placed in the Start Menu or in other locations such as Desktop.

[Languages]

This section allows to customize multilingual installations. This section defines the languages to use during setup from a *.isl* file.

[Messages]

This section allows to customize the messages that are going to appear during the installation process. Inno Setup provides default messages, but in this section, it is possible to define new messages or override these default texts.

[Registry]

In this section it is possible to define registry keys and values to write, modify or delete. The tool can work with 32 and 64-bit registry.

[Run] &[UninstallRun]

These sections specify programs to execute:

- After installation.
- Before uninstallation.

Several parameters can be configured to execute the programs such as silent installation flag or run a certain program with admin privileges.

4.4 SIEMENS INSTALLER TOOL APPROACH

During the development phase of the installer with Inno Setup, the customer requirements changed, and the installer could not be finally developed with Inno Setup, so an internal company tool was eventually used. Due to confidentiality reasons, the utilization of this tool cannot be explained in detail.

In the following subchapters, the architecture and the building process of the installer are described.

4.4.1 TOOL ARCHITECTURE

The structure of the tool is shown in Figure 19:

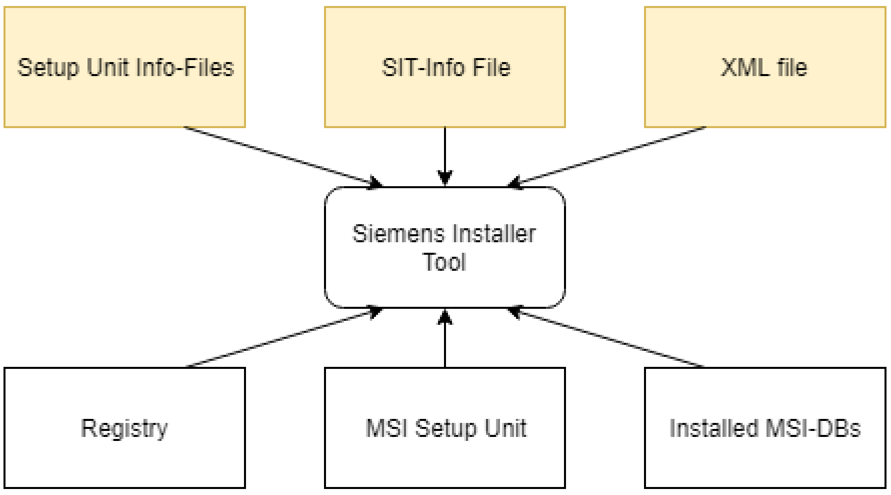

Figure 19: Tool Structure

- MSI Setup Unit: these files are *.msi* installers of the desired applications. These files provide information about the setup.
- MSI-Databases provide information about the installed products in the computer.
- Registry is used to read installed product data, edition and system configuration.
- Setup Unit Info-Files (SPF) contain all relevant data of a setup unit.

- XML files contain Product configuration and they are used by a SIT component.
- SIT info file (SIT.ini) contains general information for the SIT and is responsible for its entire control.

There are two definitions that are used during this section:

- Builder: is a person who assembles setup units within a data media and configures a SIT for the requirements of the product.
- User: is the end user of the assembled bundle, basically the customer of the product.
- Setup Unit: is a generic term for all the setup programs which installs the files of the delivered products or components of a computer.
- Bundle: is a delivery unit which can be assembled from several setup units. Generally, all products of a bundle are related with each other.

To better distinguish between the builder and the user, in Figure 20 the interfaces of the tool for both are presented:

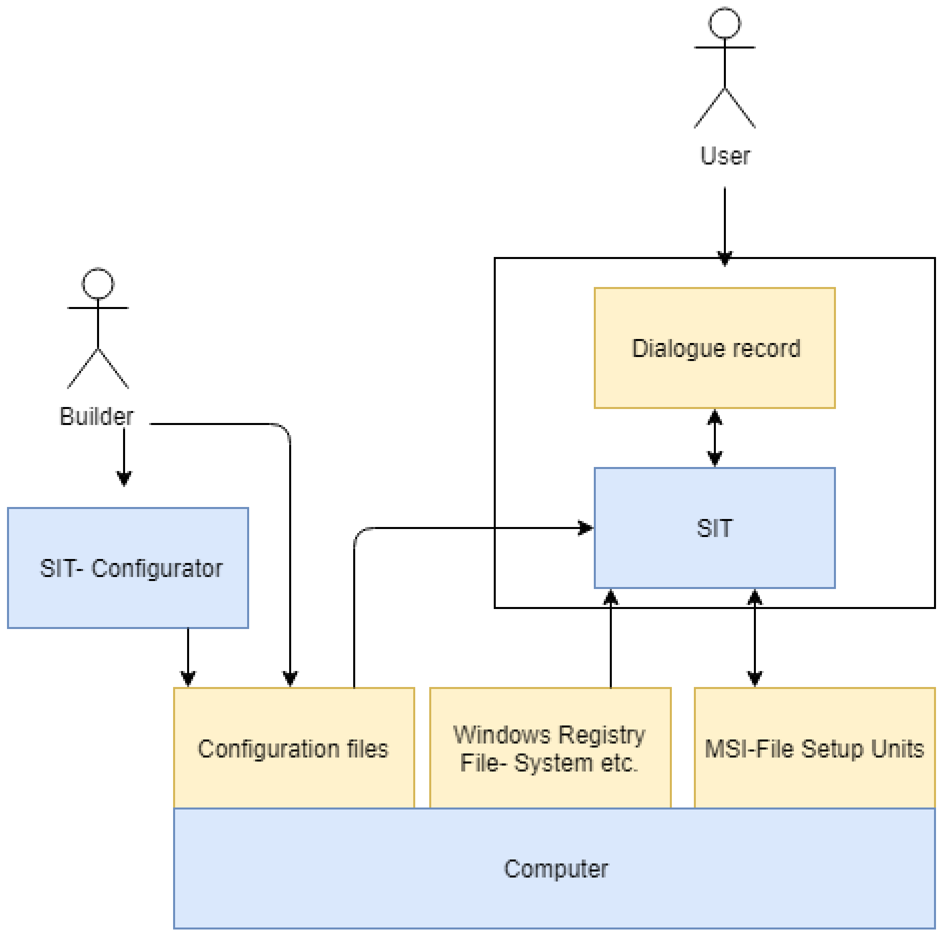

Figure 20: Siemens Installer Tool Interfaces.

Source: Internal SIT documentation

4.4.2 BUILDING PROCESS

Building the installer requires two main steps:

- Create *.msi* installer: before “wrapping” the *.msi* into SIT it is necessary to create the package that performs most of the installation operations.
- Configure SIT to link *.msi* installer: the main idea is to make a GUI to access the *.msi*. However, SIT also can perform some installation steps.

4.4.2.1 Creating .msi installer

To create this package, another tool provided by the firm is used. This tool acts as an interface to create packages using the SDK of Microsoft and provides another way to create packages with the capabilities that are most used in the products of the company.

The overall architecture of this tool (from now on, MSI creator) is shown in Figure 21:

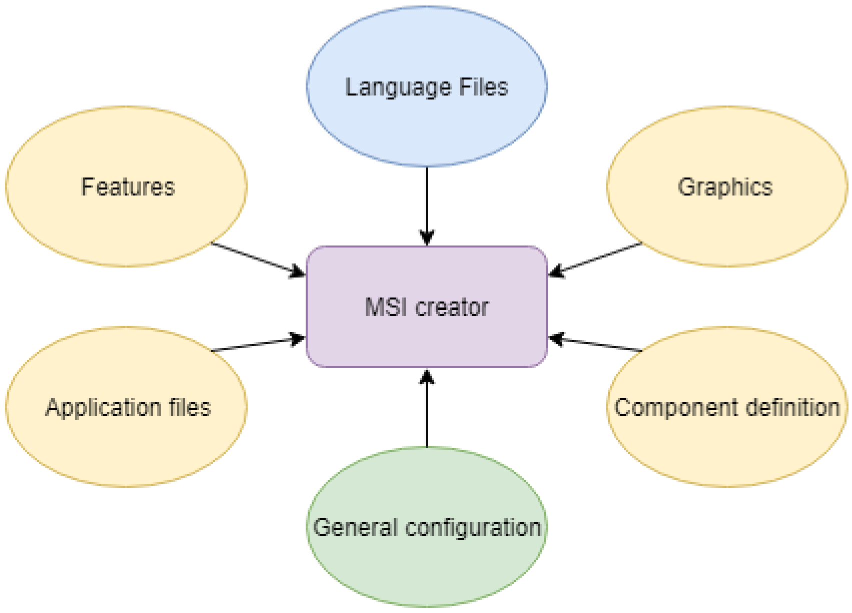

Figure 21: MSI creator Tool architecture.

As shown in Figure 21, the tool outsources the configuration of the installer to have a modular design. Therefore, if the released application changes, it is possible to modify the configuration easily.

The presented architecture is described:

- Language files: it is possible to configure the languages available for the installer and the application. To make this configuration possible, it is necessary to provide some language files.
  The structure of these files is simple:
  - ID: the language is identified by a language code. For example, German has the code ‘1031’ whereas English has ‘1033’.

  - Language content: each string or text that will appear in the application (or installer app) is classified by a tag code. The application code points to this tag to display the final text in the selected language.
- Graphics: the tool allows to customize the GUI of the *.msi* installer. The customization can be made:
  - Providing images for the installer: these images, typically in *.png* format will be displayed during the installation process. If they are not defined, default *.msi* images will be displayed.
  - Application icon: it is possible to define the installer icon and the application icon to link it to a shortcut. The icon image format is *.ico*.
- General configuration: the general configuration is provided to the tool by a *.ini* file. When the creator software is launched, it loads and read this file as the first step.
- Application files: these are the files that are going to be deployed in the final customer machine.
- Features: the software allows to define the different features that are going to be deployed in the software. Splitting the program into different features allows to split the product into parts that the installer user can select during the installation process. In the case of the application of this project, there is only one feature, called *Main_feature*.
- Component definition: the installer has its own definition of components. Components are virtual single units that perform one or more installation operations. It is possible to solve the installation process using a single component, however, splitting the installation process in components allows a modular customization of the different features the installer must provide. Thus, debugging is easier, and it is possible to plan a feature implementation schedule.

4.4.2.1.1 General Configuration

As stated in 4.4.2.1, the general configuration of the *.msi* installer is made via a *.ini* file.

This file can be edited with a text editor such as Notepad and has a well-defined structure. The structure consists of different sections with attributes and values for them.

It is not possible to describe the configurable parameters in detail due to confidentiality reasons, but briefly, the sections and parameters are:

- Product information: this encompasses the following attributes
    - Setup type: this field means the type of installation that this product represents. These types can be:
        - Language packs: packages that provide files to display a program in a specified language.
        - Merge modules: these are setup packages that cannot be installed directly [24] and must be included within an MSI. These modules enable the developer to add third-party features to the installation. Examples for Merge modules are subcontracted Microsoft *.dlls*.
        - Client edition: this is a type of installation that needs a main setup.
        - **Main** edition: this represents the main setup or main edition of the product and is suitable for the application that is described in the present Master Thesis.
        - Global edition: this means an edition independent setup.
    - Naming: there are some attributes that define the name to display during the installation process, in the window title, banners…
    - Versioning: there is another group of attributes that describe the product number, release number…
    - GUIDs: GUID stands for Globally Unique Identifier [25], it is a 128-bit key that identifies the product. This key is used to identify

    products, versions and to check authenticity. To generate them, it is possible to use a Microsoft tool or use an online GUID generator.
  - Packed MSI flag: this option allows the build to generate a unique *.msi* file that contains the compressed application data or having a *.cab* file that contains the compressed data, *.mst* files that gather the configuration settings and the *.msi* file.
  - Setup in 64-bit flag: to allow and restrict the installation on x64-Windows machines.
- SIT-defined properties: in this section the user can define properties that are going to be used by SIT.
- ARP attributes: ARP refers to Add or Remove Programs feature of Microsoft. These attributes allow to customize the entry that the program will create in this menu:
  - Icon: route to the icon that is shown in the menu.
  - Contact: the contact information that is shown in the entry.
  - Help link: an URL to find support information of the app.
- Path: defines the default installation path of the Setup Package.
- Languages: define the languages that are going to be used during the installation process and the product languages that are going to be installed.
- Special conditions: it can be configured whether the installer asks for rebooting the machine or execute this action without user permission.
- Create Desktop icon.
- Preconditions: in this section it can be checked certain conditions of the customer machine such as available RAM, Internet Explorer version, .Net version…
- Environment: it is possible to create or modify environmental variables on the machine.
- Start applications before installation: in this section, it is possible to define applications to start before the actual installation process. In the case of the project app, it is necessary to execute a program that uninstalls the

application as a Windows service prior to deleting the files and registry entries during **uninstallation.**

- Start applications after installation: as the previous one, it is possible to define applications to start right after the installation process is complete. For the case of the thesis's app, it is necessary to execute a program that installs the application as a Windows service.
- Registry: it is possible to create registry entries for different purposes for an application. However, it is also possible to define registry keys and values from **components.** For this project, registry keys are defined in a unique component.

4.4.2.1.2 Component definition

As explained in 4.4.2.1, components are used to split the installation processes into virtual units. That procedure allows to add features to the installer and track changes easily.

As in 4.4.2.1.1, component definition is made by a *.ini* file. In this file, components are listed and then each component is customized separately to assign it certain functions.

For the case of the application, several components have been created to give the installer the features required by the application and to comply with customer requirements.

The components are split into:

- Copy files: components that exclusively deliver files needed for the application and are copied in different locations. They were created as many components as different locations for these files were needed.
  - Copy static files: these are copied in subdirectories of [ProgramFiles64] folder.
  - Copy dynamic files: files that are going to be modified during runtime are copied under a subdirectory of [ProgramData].

- Copy registry keys: another component oversees delivering the necessary registry keys to the customer computer. Registry keys are placed in a *.reg* file that is created manually.

Each component can be customized separately. The main customizations available are:

- Language: it is possible to specify the language which component belongs to. This is used to distinguish between components that are only available for certain languages.
- Permanent flag: this property makes the delivered component not to be removed during uninstallation.
- Empty flag: to specify that the component has no files associated. As stated previously, components are virtual installer features and may perform actions that do not involve files.
- Destination: to specify the path to deliver the files (if any).
- Files: files are specified using whitelisting. This means that files have to be defined one by one. Although this makes the process a bit tedious, it forces the builder to have full control over the delivered files.

4.4.2.1.3 Development strategy

As explained in 4.4.2.1, the MSI creator needs several dependencies to create an installer. This makes the first steps tough until the builder gets familiar with the tool. It is necessary to understand the relations and the constraints of the different areas of the tool to add features and reduce debugging times.

To avoid getting blocked during the development phase of the installer, the following strategy was used (see Figure 22):

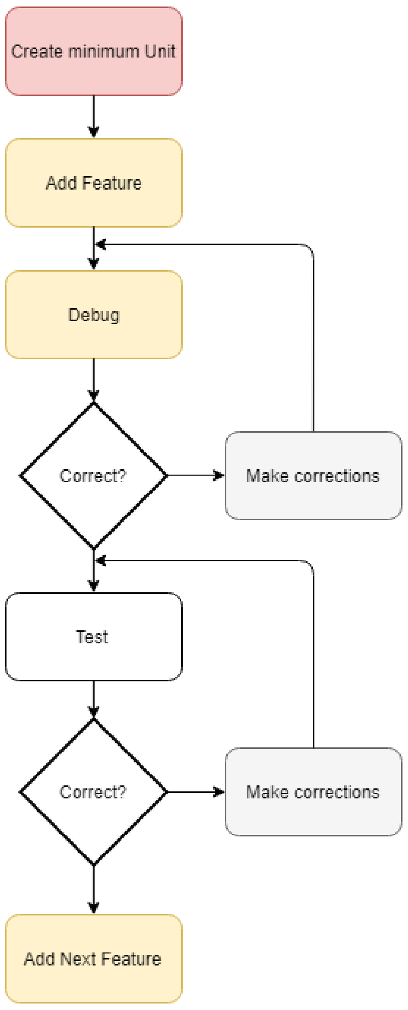

Figure 22: MSI installer development process

1. Create minimum installer unit: this step is the most difficult one. It consists of creating a minimum functional feature, component and general configuration files to build an installer. This allows to set up the environment to start adding features having the simplest possible structure.
2. Adding features: from the minimum setup, each feature that the installer requires is included to the previous version.
3. Debug: the setup unit that is deployed by the MSI creator is checked. Prior to actual testing, log files of the MSI creator are fully reviewed, removing warnings and errors. Once the build is successful, testing phase is started. If not, corrections are made, and the debug process starts again.

4. Testing: when the build is successful, the built installer is executed in test mode in different environments on a virtual machine. To check whether the installation has been successful, the following steps have to be taken:
    a. User checking: check if all files have been deployed in the correct locations and the necessary registry entries created.
    b. Log files: SIT provides the test execution of the built MSI and creates some log files that give information about installation phases, property resolution and default behavior of the installer.
    c. ORCA: ORCA is a tool provided by Microsoft SDK that provides full capabilities for editing MSI installer properties database. This program was used to check directory resolution properties and to give custom graphics to the *.msi* installer.

4.4.2.2 “Wrapping” MSI into SIT

As described in 4.4.2, the MSI package performs most of the actual installation tasks. However, there are functions that are not fully satisfied by MSI:

- GUI: the user interface of MSI packages is poor and difficult to modify. SIT provides a user interface that fits the software portfolio of the company.
- Integration: SIT is intended to ease integration of the products of the company by solving the interdependencies between them. For example, if there is a Product A that uses components from Product B that are already installed, SIT will check the version of these components and upgrade them if necessary.
- Prerequisites: although MSI provides capabilities to check for prerequisites, SIT presents a friendlier interface.

In order to wrap the MSI package into SIT, it is necessary to do the following steps:

- Define Setup settings: SIT main configuration is done with a *.ini* file. This file can configure the basic setup parameters of the installation process such as:
  - GUI.
  - Check for prerequisites.
  - Installation languages.
  - After installation/uninstallation reboots.
  - Installation checkpoints.
- Define product relations: SIT can manage product, features and component relations from an xml file. This file states the features and components that the product requires, so during installation, installed features and components can be upgraded or installed.

Chapter 5 RESULTS

This chapter mentions the results that were achieved during the development of the present thesis.

Regarding Collector, a modular component for the application was developed. The component is coded using secure coding guidelines, making it reliable for industrial purposes. The architecture of the application and the internal component structure makes Collector a modular component that can be reused with or without modifications in future applications.

Regarding testing, the creation of the testing environment was successful because it helped to find issues during development.

Finally, regarding deployment, an installer solution that complies with customer requirements was developed. Changing the tool during development phase made installer creation longer than expected, but successful.

Chapter 6 CONCLUSIONS

6.1 BACK-END CONCLUSIONS

From the Collector development it can be said:

- C# is a programming language that allows to develop applications faster. It provides capabilities to manage memory and exception handling that makes it suitable to develop industrial software for Windows.
- The use of interfaces to create the architecture of the component and the whole application allows to divide component responsibilities, therefore defining a functional, simple architecture.

6.2 TESTING PLATFORM CONCLUSIONS

The creation of the testing platform was successful because it helped debugging the application, improved the continuous integration of the application components. This ensures **quality** of the released product

6.3 DEPLOYMENT CONCLUSIONS

Clients' first contact with the application is the installer. Therefore, it is crucial to develop a solution that could performed an automated installation regardless machine´s configuration.

Chapter 7 FUTURE DEVELOPMENT

As described in the present thesis, this project provides plant transparency and health status information about devices of a PROFINET network.

The idea of having a centralized asset management can be applicable to many other sectors as the following:

- Transportation companies: rental car companies, airlines or logistics companies could also take advantage of a solution that provides status and information of their assets. For example, rental car companies could know in real time the status of their vehicles and program their maintenance.
- Hospitals and other health centers: could use a similar application to show status of their patients and manage resources more efficiently.

Regarding the present application, its future development is confidential and cannot be described in this document. The only future developments that can be mentioned are:

- Develop a scanner software that could substitute SAT. This scanner software could improve the scan method to be faster.
- Find out an application that allows to make testing on the server without using mocked or faked objects to test the whole application automatically.

REFERENCES

[1] Solarwinds, [Online]. Available: https://www.solarwinds.com/topics/network-device-scanner.

[2] Siemens, [Online]. Available: https://support.industry.siemens.com/cs/document/98161300/simatic-automation-tool-la-herramienta-de-puesta-en-marcha-y-operación-de-mantenimiento-para-los-módulos-simatic-?dti=0&lc=es-WW.

[3] Microsoft, "Thread Class," [Online]. Available: https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.110).aspx.

[4] Paessler, "How do SNMP, MIBs and OIDs work?," 2010. [Online]. Available: https://kb.paessler.com/en/topic/653-how-do-snmp-mibs-and-oids-work.

[5] Cisco, "Simple Network Management Protocol," [Online]. Available: https://www.cisco.com/c/en/us/td/docs/voice_ip_comm/cucm/managed_services/8_6_1/cucm/managed_services/snmp.pdf.

[6] SearchNetworking, [Online]. Available: https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol.

[7] SnmpSharpNet, [Online]. Available: http://www.snmpsharpnet.com.

[8] SourceForge, [Online]. Available: https://sourceforge.net/projects/snmpsharpnet/files/snmpsharpnet/.

[9] SharpSNMP, [Online]. Available:

https://pro.sharpsnmp.com/en/latest/getting-started/compiler-features.html.

[10] MSDN, "Socket Class," [Online]. Available: https://msdn.microsoft.com/en-us/library/system.net.sockets.socket(v=vs.110).aspx.

[11] N. Shetty, "Socket Programming," University of California, Berkeley, 2007.

[12] Microsoft, "Regular Expression Language - Quick Reference," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference.

[13] Visual Studio, [Online]. Available: https://www.visualstudio.com/es/vs/compare/.

[14] Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/introduction.

[15] MSDN, "Isolating Code Under Test with Microsoft Fakes," [Online]. Available: https://msdn.microsoft.com/en-us/library/hh549175.aspx.

[16] MSDN, "Using Code Coverage to Determine How Much Code is being Tested," [Online]. Available: https://msdn.microsoft.com/en-us/library/dd537628.aspx.

[17] Microsoft Visual Studio, "Test Planning and Management with Visual Studio Team Services," [Online]. Available: https://almvm.azurewebsites.net/labs/vsts/testmanagement/.

[18] Codeship Inc., "Continuous Integration Essentials," [Online]. Available: https://codeship.com/continuous-integration-essentials.

[19] JetBrains, "TeamCity," [Online]. Available: https://www.jetbrains.com/teamcity/?fromMenu.

[20] FakeItEasy, "What can be faked," [Online]. Available: http://fakeiteasy.readthedocs.io/en/stable/what-can-be-faked/.

[21] FakeItEasy, "Specifying a Call to Configure," [Online]. Available: http://fakeiteasy.readthedocs.io/en/stable/specifying-a-call-to-configure/.

[22] Flexera Software, "Building your first project using InstallShield Limited Edition for Visual Studio," [Online]. Available: https://www.youtube.com/watch?v=xI36omxTsSw.

[23] Flexera Software, "End-User License Agreement- InstallShield," 2011.

[24] Symantec, [Online]. Available: https://www.symantec.com/connect/articles/about-merge-modules.

[25] "guid.one," [Online]. Available: http://guid.one/guid.

[26] Siemens AG, [Online]. Available: https://support.industry.siemens.com/cs/document/67460624/proneta-2-4-0-44-commissioning-and-diagnostics-tool-for-profinet?dti=0&lc=en-WW.

[27] IBM, "Socket Types," [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/ssw_aix_61/com.ibm.aix.progcomc/socket-types.htm.

[28] Microsoft, "Overview of Microsoft IntelliTest," [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/test/intellitest-manual/introduction.

Part II BUDGET

Chapter 1 MEASUREMENTS

All resources that may apply an economic contribution and the used amounts of them for this project are listed in this chapter. In order to ease the reading, the following resources are divided by categories.

1.1 EQUIPMENT MEASUREMENTS

Equipment	Quantity	Project Usage(h)	Year Usage Time (h)
Computer	1	880	2112
Test Devices Platform	1	250	500

Table 7: Equiment Measurements

1.2 SOFTWARE MEASUREMENTS

Software	Quantity	Project Usage(h)	Year Usage Time (h)
Visual Studio Professional 2015	1	880	2112
SAT	1	880	2112
Microsoft Office	1	120	300
Wireshark	1	250	500
SIT	1	250	-
ORCA	1	250	-

Table 8: Software Measurements

1.3 DIRECT LABOR MEASUREMENTS

Direct Labor	**Hours**
Information Collect	100
Collector Development	250
Testing framework design	50
Testing	80
Code analysis	50
Documentation	100
Installer development	250

Table 9: Direct Labor Measurements

Chapter 2 UNIT PRICES

Unit prices of materials and equipment used for the present Thesis, as well as hourly prices for the carried-out tasks are listed in this chapter.

2.1 EQUIPMENT UNIT PRICES

Equipment	Unit Price (€)
Computer	1.000
Test Devices Platform	25.000

Table 10: Equiment Unit Prices

2.2 SOFTWARE UNIT PRICES

Software	Unit price(€/year)
Visual Studio Professional 2015	641
SAT	300
Microsoft Office	30
Wireshark	Free
SIT	Free (Internal)
ORCA	Free

Table 11: Software Unit Prices

2.3 DIRECT LABOR HOURLY PRICES

Direct Labor	**Hourly Prices (€)**
Information Collect	20
Collector Development	60
Testing framework design	60
Testing	50
Code analysis	50
Documentation	40
Installer development	60

Table 12: Direct Labor Hourly Prices

Chapter 3 PARTIAL BUDGETS

This chapter lists the partial budgets calculated as the product between the measurements and their unit prices. Moreover, the amortization of equipment and software is considered in the following calculations.

3.1 EQUIPMENT BUDGET

Equipment	Price (€)	Project Usage(h)	Year Usage (h)	Annual Amortization	Cost (€)
Computer	1.000,00	880,00	2.112,00	0,25	104,17
Test Devices Platform	25.000,00	250,00	500,00	0,15	1.875,00
TOTAL					1.979,17

Table 13: Equipment Partial Budget

3.2 SOFTWARE BUDGET

Software	License price(€/year)	Project Usage(h)	Year Usage Time (h)	Annual Amortization	Cost(€)
Visual Studio Professional 2015	641,00	880,00	2.112,00	1,00	267,08
SAT	300,00	880,00	2.112,00	1,00	125,00
Microsoft Office	30,00	120,00	300,00	1,00	12,00
Wireshark	Free	250,00	500,00	NA	0,00
SIT	Free (Internal)	250,00	-	NA	0,00
ORCA	Free	250,00	-	NA	0,00
		TOTAL			404,08

Table 14: Software Partial Budget

3.3 DIRECT LABOR PARTIAL BUDGET

Direct Labor	Hours	Hourly Prices (€)	Cost(€)
Information Collect	100	20	2.000
Collector Development	250	60	15.000
Testing framework design	50	60	3.000
Testing	80	50	4.000
Code analysis	50	50	2.500
Documentation	100	40	4.000
Installer development	250	60	15.000
TOTAL			45.500

Table 15: Direct Labor Partial Budget

Chapter 4 GENERAL BUDGET

The general budget is obtained as the sum of the partial budgets that were calculated in Chapter 3. Other expenses as electricity, maintenance and office equipment are included as a percentage of the total sum.

General Budget	
Category	Partial Sum
Equipment	1.979,17
Software	404,08
Direct Labor	45.500
Sum	47.883,25
Other Expenses (5%)	2.394,16
TOTAL	50.277,41

Table 16: General Budget