Aalto University
School of Science
Degree Programme in Software engineering

Janne Suomalainen

A Remote installation automation platform

– A design proposition for automating EDMS platform software installation

Master's Thesis
Espoo, September 12th, 2016

Supervisor: Lauri Malmi, Professor, Vice Dean of Teaching
Advisor: Mikko Lampinen, M.Sc.

Aalto University
School of Science

Abstract of master's thesis

Author Janne Suomalainen

Title of thesis A Remote installation automation platform – A design proposition for automating EDMS platform software installation

Master's programme Software Engineering

Thesis supervisor Lauri Malmi, Professor

Major or Minor/Code T3001

Department Computer Science

Thesis advisor(s) Mikko Lampinen, M.Sc.

Date September 12th, 2016 **Number of pages** 103+11 **Language** English

Abstract

Software installation is a reoccurring task in complex energy data management system (EDMS) platform development projects. The installation verification is part of the deployment, Continuous Integration and Quality Assurance testing efforts. While the procedure in general follows infrequently changing patterns the task requires technical knowledge and consumes time that is taken away from other development tasks. Moreover, the business is driving the deployment model from larger infrequent deployments towards more frequent incremental deployments that will demand high efficiency of the software deployment methods.

In this thesis we studied the installation cases of a specific EDMS platform and the problems related to remote installation and installation automation. The goal was to come up with a solution that will increase the installation efficiency and decrease the human effort required to complete the installation tasks.

Based on the findings, we modelled the installation process as an abstracted work flow and designed a multi-agent software platform. The designed platform can execute the installations in remote environments in an automated manner. The design includes fully automated installation execution and result reporting, centralized management interface for all the installation processes and proposes different feedback methods for installation information distribution.

To prove that the concepts and ideas presented in the design work well in practise, we build a reference implementation. With hundreds of executed, differentiating installations that served a real life purpose the platform proved that with automation it can improve the installation process efficiency and decrease the required manual human effort after the processes that previously required manual effort were automated.

Keywords installation automation, remote installation, EDMS software installation, multi-agent software.

Aalto-yliopisto
Perustieteiden
korkeakoulu

Diplomityön tiivistelmä

Tekijä Janne Suomalainen

Työn nimi Automaatioalusta etäasennuksille – suunnitelma EDMS-sovellusalustan asennusten automatisoimiseksi

Koulutusohjelma Ohjelmistotekniikka

Valvoja Prof. Lauri Malmi

Pää tai sivuaine/koodi T3001

Työn ohjaaja(t) DI Mikko Lampinen

Päivämäärä 12.9.2016 **Sivumäärä** 103+11 **Kieli** Englanti

Tiivistelmä

Energiatiedonhallintajärjestelmien (*EDMS*) kehitystyöprojektien eri vaiheissa on jatkuvasti tarvetta uudelleenasentaa ja päivittää järjestelmiä. Asennusta tarvitaan paitsi tuotteen toimituksissa asiakkaille, mutta myös osana jatkuvaa integraatiota (*CI*) ja laadunvarmistus (*QA*) testausta. Vaikka nämä ohjelmistoasennukset pääosin seuraavatkin samaa kaavaa, ne vaativat silti teknistä osaamista sekä kuluttavat sovellusammattilaisten aikaa, joka on poissa muusta kehitystyöstä. Lisäksi sovelluskehitystyön trendi on vahvasti siirtymässä harvemmin tehtävistä suurista päivitysoperaatioista kohti jatkuvasti tapahtuvaa pienempien osa-kokonaisuuksien päivittämistä. Tämä muutos asettaa ohjelmistojen toimitus- ja asennusprosessin tehokkuuden entistäkin tärkeämpään asemaan.

Tässä diplomityössä tutkittiin erään EDMS-sovellusalustan asennustapauksia sekä ongelmia, joita asennuksen automatisointi ja etäasennus tuovat mukanaan. Tavoitteena oli löytää ratkaisu, jolla asennusoperaatio saadaan tehokkaammaksi sekä siihen ihmiseltä vaadittavan työmäärän suuruutta saadaan pienennettyä.

Tutkimustyön löydösten perusteella asennus mallinnettiin sarjaksi tehtäviä ja ehdotetaan sovellusalustaratkaisua, joka kykenee suorittamaan etäasennustehtäviä automaattisesti. Ratkaisu sisältää asennusten suorittamisen, raportoinnin, operaatioiden keskitetyn hallinnan sekä ehdotuksia menetelmistä asennuksista saatavan informaation välittämiseksi.

Osana tutkimusta toteutettiin toimiva ohjelmistoalusta ehdotettujen ratkaisujen ja konseptien toimivuuden testaamiseksi. Kehitetyllä ohjelmistolla suoritettiin satoja vaihtelevia asennuksia, jotka pohjautuivat todellisiin asennustarpeisiin. Asennukset osoittavat, että ehdotettu ratkaisu toimii tarkoituksessaan. Ratkaisun seurauksena asennusprosessin tehokkuutta saatiin nostettua ja prosessien automatisoituessa asennustyöhön ihmisiltä vaadittava työpanos pieneni.

Avainsanat asennusautomaatio, etäasennus, EDMS-ohjelmiston asennus.

Acknowledgements

I would like to thank my supervisor, Professor Lauri Malmi for his support and for providing valuable feedback and comments about the thesis. I would also like to thank Mikko Lampinen for being the advisor of the thesis. Finally, I would like to thank my wife Heidi and my daughter Nea for providing support and understanding during this challenging task.

Espoo, September 5, 2016 Janne Suomalainen

Abbreviations and Acronyms

BDI software model	Belief-desire-intention software model, model developed for programming intelligent software agents.
CFA	*Central Facilitator Agent* is a software agent that passes commands and provides information to other agents in a multi-agent system in order to help them achieve their goals.
Continuous Integration (CI)	In software development CI is a practice of frequently merging all the software changes made by developers and to test with varying amount of effort that they work together.
DIA	*Domain Installation Agent* is a software agent that operates in the target environment in order to achieve goals of the multi-agent system.
Installability	Installability of a software refers to the capability of being installed.
Multi-agent system (MAS)	Computerized system composed of multiple interacting intelligent software agents within an environment.
Quality Assurance (QA)	Administrative and procedural activities targeting of ensuring the quality of a service or product.
SOA	Service-oriented-architecture - an architectural pattern in which application component provide services to other components via aa communications protocol, typically over a network.
Software agent	A computer program that acts for a user or other program in a relationship of agency.

Table of Contents

Appendices

Appendix 1. List of installation plan tasks and sub-tasks.
Appendix 2. Example installation plan in serialized format.
Appendix 3. Example installation subscription in serialized format.
Appendix 4. List of all requirements and the solutions presented by the thesis.
Appendix 5. Executed performance test scenario setup with Apache JMeter application.

1 Introduction

This Master's thesis is done for a company that is partnering on a large customer project that includes delivering energy data management solutions to one significant utility company that operates on the European market. The project delivery time schedule is challenging and it also contains major software development effort for some new functionality. Such software projects that include software development tend to have a high risk for exceeding predefined time schedules. Some known problems are related to lack of test automation and continuous integration systems that help catching the problems in earlier phases of the development cycle.

The company's existing Energy Data Management Solution (EDMS) is built on top of Oracle database that runs mainly on Windows server machines. It has close to two decades of development history. Over the long development cycle the product has grown to a very complex solution that has been deployed to various customers with highly varying components and configurations that lack standardization. As a side product the company has developed customized installer software that is used to install the product and to handle complex database migrations per customized customer system. The installer tool is a wizard-like Windows application that can be used manually to install or upgrade target systems locally.

The company has started working on standardizing the product that is sold to new customers, which should lead to more homogenous installations. There are still many customer systems that run with a customized software installation that produces some legacy-related issues. Regardless of the customization details, most of the installation cases contain more or less the same set of tasks and procedures that are currently manually executed on each installation. On this basis, there lies an opportunity for task automation. That requires identifying the actual variations in installation that could be parametrized with proper set of input and to create some pre-defined plan that can act as an input to execute most of the installation cases automatically. Some corner cases may occur that cannot be automated, but covering e.g. 90 % of the installations would already produce huge value to the company.

1.1 Motivation

The modular type of Energy Data Management System (EDMS) that the company is selling as a product is considered to be a complex system to install in a server environment. Because of the modularity of the product, the scope of the product installation varies over customers and setups. The EDMS system has essentially a 3-tiered architecture with backing database, application server and client machines having local installations of the client software. The current consensus in the company is that the EDMS installation operation requires lots of steps and procedures that are specific to the product that the company is selling, i.e. that the installation is somewhat different than for other systems delivered by other companies.

The goal of this thesis is to find a way to automate the installation of this specific EDMS product and to make it possible to allow deployments over the network without constant human interaction. Studying and understanding what tasks this particular installation procedure includes and how does the procedure differ from typical software installations is a prerequisite for being able to automate the installation procedure of such product that is expected to be different than other software products. Moreover, if the product is said to be modular, it is required to identify what are the differences from the installation point of view. If the analysis reveals that the installation procedure can be described as a treelike structure with branched execution paths and the size of the tree structure is inside sensible range, it should be possible to describe each unique installation scope with proper set of pre-defined inputs that can be processed in a deterministic finite automata application without on-demand type of new input information.

Energy markets are generally highly regulated by the governments. As the requirements for the energy companies' EDMS systems keeps changing, some part of the software product is always under development. Having automated product installation capability serves the software development in two ways. Firstly, there is typically a continuous integration (*CI*) system that continuously tests new software builds to verify that existing functionality keeps working. Having a system that installs and upgrades the product versions help catching various problems in a very early state. For example, if the previous version cannot be updated with the latest changes, it means that the new version is probably not compatible with the old installation. Secondly, in general, there is almost always a need for manual testing in software development projects. Some tests are hard to automate in a sustainable way with reasonable effort. There may also be some configurations that should be tested in the existing system before deployment. Therefore, having a system that is kept up-to-date with the latest software version helps the development effort. With automated installation, a system can be configured to update to the latest version in a way that in the beginning of every working day the Quality Assurance (*QA*) employees have a testing environment with latest changes to the product without any spent work hours.

Automating product installation serves also scaling up possibilities. If installing the same software to another customer does not cause a lot of extra effort on the deployment side, having more customers ordering the same product does not increase significantly the need for extra resources. In an optimal situation deploying the product to two similar systems should require about the same amount of effort than to deploy it to three similar systems. Also the amount of resources needed for maintenance should decrease with automated deployment.

The ultimate motivation for automating software deployment lies in the profitability. Software product development typically includes lots of testing, which includes also system-level testing. To be able to deliver a single new version of a product that contains new features, more than only single final installation is required. System-level testing requires existing system and the software delivery project practises suggests testing to happen as early as possible. If an installation requires more man-hours, it also costs more money as salaries, which affects to the development profitability. Moreover, if product installation is said to be complex, it also may require special skills from the installation personnel which leads to the fact that scaling up the number of installations per demand may not be very fast if it requires a lot of training.

With a system that provides automation and scalability these issues can be solved partly if not fully and it should have a positive effect on software business profitability. It should also help winning more business deals since lower production costs, provide possibility to have more competitive pricing. In addition, having more efficient QA procedures should have a positive effect on the overall software product quality.

1.2 Research goals and constraints

The research has some preliminary constraints that may set some restrictions to the technologies and solutions that are considered to be suitable for the purposes. Also the practicalities, for example, how quickly such system could be implemented or how much work it requires have weight in the process. Moreover, the business purposes of the system are also considered with high weight in the decision making.

Initial goals for the product deployment system set by the company are:

- to produce informative reports of errors and problems that helps solving them faster,
- to handle fully automated or with minimal manual effort updating of the product in testing environment where quality assurance engineers can run functional testing,
- to provide a single user interface that can be used to coordinate a set of remote installation,
- to support remote installation of files, services, database migrations and relevant tasks that run on Windows operating system,
- to possibly support fully-automated fresh installation of a new installation on top of public cloud infrastructure (e.g. Amazon AWS).

To help planning on how such system can be developed in an efficient and sustainable way, the research is about:

- identifying the problems related to remote installation of this particular energy data management product,
- identifying the decisions that need to pre-defined for automated installation,
- identifying the input data set that is required to do fully automated installation,
- identifying the software development related tasks that need to be done to be able to handle automatic remote installation,
- providing a solution proposition for identified problems related to the remote installation,
- providing a proposition of the full solution that can manage automatic remote installations and related tasks.

There are also some additional constraints that should have some weight in decision making when thinking solution propositions:

- considering especially the requirements of a large system that works as a national data hub for energy data,
- considering the needs of existing system installations,
- considering carefully the commitment to any specific non-mainstream technologies,
- considering continuous integration requirements for automated installation,
- Considering existing systems and technologies that the company has invested in and,
- considering the future scenarios of automating remote upgrades for customer test and production environments.

1.3 Research questions, objectives and limitations

The necessity of having many testing environments will increase the amount of work needed for maintaining and keeping the different environments up-to-date and versions synchronized, if the job requires manual tendering. Therefore, to reduce labour costs and problems related to maintainability of different environments, automating installation and providing the means to do installation tasks in multiple environments with minimal effort required for humans can produce lots of value, money and time savings and increase product quality.

The topics mentioned above suggest that automating installation is something that should have a quick return on investment over the spent time and effort. This research is about proving that the target company's EDMS product's installation can be automated in a way that it can be remotely controlled and deployed to a Windows operating system using typical network connection and available protocols and standards. The outcome should be a proposition of a system or systems, including technics, components and technologies that can be used for configuring, launching, controlling and observing the installation in multiple target hosts by using a single management system that has a network connection to all the other hosts that run the product. The research should also identify the key problems and try to find solutions for overcoming them completely or at least minimizing their effect and severity.

The research questions are defined to get proof of concept of the system described above; and to find a generalized approach for installation that should abstract away the target environment location. Each research question (*RQ*) has some sub questions that help finding answers to research questions. Research questions are defined as the following:

RQ1: How to run fully automated fresh installation of the product to a Windows machine that is accessible through connected network?

> **RQ1.1**: How to define installation that can be automatically installed without required human effort during the process?

RQ1.2: What things in the installation process can be changed to improve the automatic and remote installability without having an effect on how the application to install works?

RQ2: How to have a fully automated upgrade of the product to an existing system that is run on a Windows machine that is accessible through connected network?

RQ2.1: How the upgrade installation differs from a fresh installation?

RQ3: How to remove completely or minimize the difference in installation procedure, whether target system is located in local area network, on a network connected through the internet or on a virtual machine running in public cloud?

RQ3.1: How to minimize new problems introduced by remote installation?

RQ3.2: How to minimize or abstract the requirements from local environment?

RQ4: How to reduce required effort to put up several copies of the same environments with same installed application scope close to the effort that is required to put up one such environment?

RQ4.1: What type of errors occur during typical installation and can some of them be avoided, automatically resolved or detected earlier in the installation process?

RQ5: Besides the savings in human effort, what extra benefits does the automated installation provide as a side product compared to manual installation?

RQ5.1: How the faster feedback cycle from of a non-working installation done on top of a new build can be provided when automating installation as part of CI operations?

RQ5.2: What valuable or measurement data can be collected from the installation that can be used in analytics purposes to grow business potential or efficiency?

This research objective is to provide a conceptual idea about how to implement a system or a framework that can remotely install and upgrade product installations in fully or highly automated fashion. In addition, it should be possible to integrate that system as part of automation systems, for example, *continuous integration* (CI) system, to provide feedback of testing installations of new builds. The system should provide good enough performance for scaling the number of installations to a sufficient amount, for example, a hundred installations per night that should cover ten times the current

need. In general, performance optimization strategies for improving the speed of a single installation are out of the scope of this research. Also, the research is not concentrating particularly on providing a full proof security solution that covers network data transportation. The initial goal is to provide proof of concept that it can be done in the company's private network that is considered to have restricted outside access. Providing security over the transmission and storing the required data is also topic left for future research.

1.4 Research methods, scope and context

Installability of a software refers to the capability of being installed. There may be capability to install the software with specific configurations into a specific operating system running a specific version of the OS software where other combination results being incapable of installation. Being capable to install the software may require some or none manual steps.

The current *installability* of the target software product is not clearly known in the company. In some cases, the installation engineer conducting the installation may encounter unexpected problems that require solving by trial and error. In other cases the problem solving may require assistance from a technical specialist who runs some configuration changes that helps solving the problem. In the end, there may not be a single responsible party in the installation that knows all the steps and details required by a single successful instance installation. In the past, there has not been systematic procedures to collect the installation data related to encountered problems (**RQ5**). Therefore the understanding of current *installability* and the possible problems in installation procedures are collected verbally from the installation engineers and it may not include all the problems that are encountered during the research.

For this research, I will modify the existing installer software to be compatible with the remote installation solution to be able to execute the installation related tasks and to be able to effectively report the results and errors according to requirements and needs. The installation workflow is run based on a plan that is designed in a way that it can abstract the details of a specific environment into a form that any local agent can interpret and transform them to become environment-specific on the target site (**RQ3**). In addition, I will implement a reference implementation that implements the key components to test the remote installation concepts identified by the analysis of how the remote installation should be done to achieve the set goals (**RQ1, RQ2**). Creating the fully operational system for managing all installations and environment instances is not part of this research. Since the need for such a system is presently highly relevant, it is very likely that such system will be developed based on the concepts and solutions proposed by the thesis research results.

The current installation software has an implementation for automated installation that is made for testing installations from new builds. The system requires very specific conditions and is not very adaptable for wider use. In addition, the main message from the company's developers has been that the system is too complex and time

consuming to use and it is not clear how to react to the results that the system produces in a form on emails. It is also not very robust from the integration point of view. One desirable outcome of the research is to make the work of the software developers to be easier and more efficient. (**RQ4**, **RQ5**)

For measuring the concepts and identifying the problems occurring in actual customer environment upgrades, the tools created for this research will be used to update testing environments that are identical for actual customer environments (**RQ4**). The product installation scope can be very different for different customers and have very different sets of modules included. One specific installation scope will be selected as a reference point that can be used to estimate the selected technologies and choices that are not dependent on the product itself. The selected scope is considered as the standardized form of the product and it is part of actual project that contains ongoing development and deployment activity that should help proofing the concepts in real life like cases.

To define the objective, the installation is considered to be fully automated if by providing set of required input parameters, the system can be configured to automatically collect installation package for a specified customer, detect the current version of the existing running system and the latest available version for that version branch and to execute two consecutive installations when new versions are available without any additional effort in between or after the installation (**RQ1, RQ2**).

1.5 Structure of the thesis

Chapter 1 presents the research questions, goals and the motivation for conducting it. The rest of this thesis is organized as follows. Chapter 2 presents the related technologies, provides an overview of the problems related to the remote installation platform and reviews related research for key concepts used in this thesis. Chapter 3 adapts the research to a specific practical context and proposes concepts of a design for a remote installation platform that can fill the goals set for the research. The proposed concepts are tested with a reference implementation using a real world use cases.

Chapter 4 presents analysis and discussion about how the solution proposed in Chapter 3 worked for the research goals and requirements defined in chapters 1 and 2. Chapter 5 concludes the topic by explaining how each research question was answered, discusses briefly the alternative solutions available and provides topics for future research for continuing on the subject. Some details of the proposed solution are attached to the appendices 1-3. Appendix 4 lists all the requirements presented in Chapter 2 and for each of them explains the solution how the proposed remote installation platform will fill the particular requirement.

2 Background, concepts and literature

This chapter discusses software deployment automation, problems in remote installation and its context in terms of existing literature and applications in order to provide basic understanding in the field. Another goal is to define and explain the key terms and to state their relation to the research subject.

The chapter starts by presenting the relevant technologies and components that are interesting or important for the research and are important part of understanding the problems the research needs to address to. After covering the technologies, the chapter presents the key problems in remote installation and installation automation activities and presents design concepts in the related field. Solving each presented problem produces sets of requirements for the target proposition. In order to track the requirements and how they are addressed, each of them is given labels (e.g. *Req.1)*. To help reflecting the discussed topics there is a list of key points presented after each section. At the end, the chapter provides an overview for all the requirements presented in this chapter and the distribution of focus areas for the research.

2.1 Technologies and components related to remote installation

2.1.1 Network connection layers and protocols

Operating in a network environment and being network-centric adds an extra level of complexity to a system. The network may suffer for example from unexpected temporary delays, broken links, unresponsive or dead nodes, nodes running several different versions of the controlling software and many other problematic anomalies that requires robust and reliable solutions that can cope with such events. By default, for a software that operates on the internet these events are covered by standardized protocols.

Standard network technology related specification documents use commonly agreed layers to categorize and classify roles of different communication functions that are part of network architecture. Two most commonly known basic conceptual reference models to define these layers are Open Systems Interconnection (*OSI*) model and TCP/IP internet protocol suite. The latter categorizes standards and technologies of the TCP/IP implementation used in the internet where the first one is commonly used in other data communication applications. The basic idea in these models is to provide interoperability of diverse communication systems with standard protocols, to provide a clear division of responsibilities across protocols and to maintain consistency among all related standards. (The International Organization for Standardization 1994, Baker 2009, Frenzel 2013.)

The layers of data communication models and their references are shown in the figure below (*Figure 1*). The TCP/IP model has fewer layers (4) than the OSI model (7), but they are similar, if not fully equivalent to layers or combination of layers in OSI-model (Frenzel 2013). The lowest layer is called *hardware layer* or *Data Link Layer* that covers the communication on the physical network components. Ethernet is the best known data link layer protocol. The next layer is *IP layer* (network) that handles routing packages through a network. Third layer is the *transport layer* that provides connection-oriented or connectionless services for application layer services between networks and offers reliability. The fourth and highest layer is application layer that sends and receives data from applications. The applications that operate on application layer does not require knowing or solve the complexities that are managed in the lower layers which is one of the main points of these layered models. (National Institute of Standards and Technology 2009, sec. 2-1)

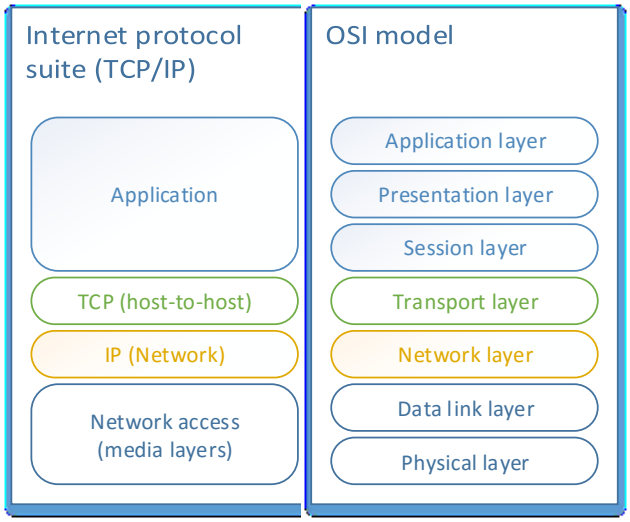


Figure 1. Layers and their relativities in most common data communication reference models (Frenzel 2013).

Transfer Control Protocol (*TCP*) is one standardized and mature protocol that handles several network-related problems. Implementing remote installation system on top of TCP protocol is a natural choice since it has been used many decades in countless production environments. Therefore, it can be considered as a very reliable protocol. Trying to invent a competitive solution for such nontrivial problem would require a lot of effort and provide very little in return. Few good reasons for using TCP protocol for remote installation systems are that it takes care of resending dropped packages, verifying the contents and assembling the packets in the correct order at the receiving end. (Internet Engineering Task Force 1981, pp. 3,10.)

TCP protocol operates at the transport layer of both Open System Interconnection (OSI) and internet protocol suite models and is designed to handle transportation requirements. To be used with actual applications, it requires some protocol that operates on application layer that is visible to the application. In general, application layer protocol should be able to trust that packet transportation over the network is reliable. It can concentrate on providing services to the application. (Internet Engineering Task Force 1981, pp. 3,10.)

2.1.2 Virtual private network connections

Virtual Private Network (VPN) is a generic term for logical network that connects two physically separate groups (networks) through a secure tunnel that is built to transfer any data through open, insecure network (e.g. through the internet) in encrypted form in a way that the communicating works like it was in a private network (Internet Engineering Task Force 2005, p. 6). Virtual Private Network (*VPN*) uses protocols to encrypt network traffic and provide user authentication and integrity checking. VPNs are often used to provide secure connection across untrusted networks. For example, VPN technology is commonly used to extend protected network of a multi-site organizations across the internet and to provide secure remote user access to the organization's internal network resources. Common technologies for establishing VPNs are *IPsec* and *Secure Sockets Layer* (SSL) / *Transport Layer Security* (TLS). The connection can be made as a temporary that disconnects when the session is over or as a permanent connection between two or more locations. (National Institute of Standards and Technology 2009, sec. 2-7, Harmening, Wright 2013, p. 855.)

VPN access requires hardware resources for processing the encryption and decryption of packets. The VPN protocols operate primarily on network layer of the *OSI model*. The majority of the processing work is done by the interconnecting hardware. The other common VPN technology is *IPsec*. The main idea in *IPsec* is to provide secure communication for the IP (network) layer. The protocol includes security protocol where the communication protection methods are defined and key negotiations where negotiation parameters and identity authentication are defined. (Mao, Zhu et al. 2012, p. 1.)

One of *IPsec*'s perks is that it is invisible to the applications using a VPN connection and therefore they don't need to know anything about whether it is used or not. IPsec has two modes. The first, transport mode secures the information to each device trying to establish a connection. In transport mode the header is not encrypted while the payload data is. The altering of data is secured by using calculated hash values. The second, tunnel mode encrypts the whole packet and encapsulates it into a new packet. It is used for network-to-network communications, e.g. in virtual private network connections. Using *IPsec* VPN requires having proper IPsec client software installed on all the target computers. (Harmening, Wright 2013, p. 860, Mao, Zhu et al. 2012, p. 1-3.)

Secure Socket Layer (SSL) is a set of internet data security protocols which has been widely used for identity authentication and data transmission between web browser and the server. SSL protocol is in TCP/IP protocol stack and other protocols that operates in the application layer. It provides security support for data communications. (Mao, Zhu et al. 2012, p. 1-2.)

SSL VPN is not an actual VPN. It is more of an interface for a web browser that provides services that acts like VPN's. *Transport Layer Security* (TLS) is the successor to SSL and is used to prevent eavesdropping on information transferred between

networked parties. The security level depends on the strength of the used encryption algorithms. In TLS/SSL the parties negotiate on what encryption method is going to be used between them, which should be the strongest one that both of them supports. After the encryption method is agreed, the parties exchange their public keys which are used to encrypt the packets sent to the other party. The process may include having certificates that are used to authenticate the public keys of the other parties through trusted certificate issuer parties (3rd parties). Using certificates should also help authenticate the other party since they can only decrypt the packets encrypted with their public keys if they possess the matching private key. Having support for SSL VPN requires implementation from the application which can be done with web browser integration. In general, operation of SSL VPN is less complicated compared to IPsec VPN. SSL VPN is not affected by firewall between the client and the server, but in general it is not suitable for point-to-point VPN and is therefore used per application basis instead of per network basis. (Harmening, Wright 2013, p. 862-863, Mao, Zhu et al. 2012, p.2, 4.)

2.1.3 Firewalls and access policies

United States National Institute of Standards and Technology (*NIST*) defines that "*Firewalls are devices or programs that control the flow of network traffic between networks or hosts that employ differing security postures.*" (National Institute of Standards and Technology 2009, sec. 2-1). Moreover, firewalls are used to separate networks with differing security requirements such as the internet and an internal network having access to sensitive data. The primary function of a firewall is to prevent unwanted traffic from entering a network. Therefore, they are typically placed at the edge of logical network boundaries which is usually a single node or nodes that connects the internal network to the internet. (National Institute of Standards and Technology 2009, sec. 3-1)

Basic firewalls operate on network layer inspecting which packages are allowed to pass and which are not. More advanced firewalls examine all the layers or *OSI model*, providing more granular and thorough examination (National Institute of Standards and Technology 2009, sec. 2-2). Because firewalls have some control over the network traffic and they may operate on the same layers as applications do, the application design should be somehow aware of their possible impact on the application's operability.

A firewall may be hardware or software based and there may be multiple firewalls controlling traffic between two networked parties. Firewalls are often placed at the perimeter of a network. In such places, firewall can have separate interfaces for external and internal networks having different set of rules based on the origin of the traffic. The technologies used by firewalls include for example *packet filtering*, *stateful inspection*, *virtual private networking* and *network access control*. (National Institute of Standards and Technology 2009, sec. 2-2)

Packet filtering is the most basic feature of a firewall. They manage access control by a set of defined directives that is often referred as a *ruleset*, which does not consider the content of packets. Instead, they inspect the source and destination addresses, used protocols, ports and the direction (*inbound* or *outbound*) of the packet and finds the first matching rule from their access control list based on this information that define whether the packet is allowed to pass or not. *Stateful inspection* extends the functions of packet filters by tracking the state of the connection and blocking the packets that differs from the expected state. The common use case for *stateful inspection* is to allow responding to connections that were originally initiated from the internal network, which means that firewall will allow traffic that is part of open connection regardless of the destination and port information. (National Institute of Standards and Technology 2009, sec. 2-3, 2-4.)

Some Firewalls may offer functionality to establish *virtual private networks* (VPNs). Firewall at the edge of a network may encrypt and decrypt specific network traffic flows between the internal and external networks. In addition, when user located in external network tries to connect to an internal network, the firewall may act as a gateway that negotiates whether the client will be granted access to the restricted network and to which specific network resources the access is granted. In general, firewalls and VPN services must be configured to work together in order to have working VPN connections. (National Institute of Standards and Technology 2009, sec. 2-7, 2-8.)

Network Access Control (NAC) or sometimes referred as *Network Access Protection* (NAP) is a firewall functionality that allows access to network resources based on the user's credentials and additional health checks performed on the user's computer. The health checks may contain requirements, for example about the latest update for antimalware and personal firewall software, security configurations, elapsed time since last system malware scans and other protection tasks against malware. Health checks require some sort of client software that can be controlled by the firewall. (National Institute of Standards and Technology 2009, sec. 2-8, 2-9.)

2.1.4 Software agents

The term "*agent*" is widely used by many people working closely related areas, whereas the exact meaning may vary across the research papers (Wooldridge, Jennings 1995, p. 116, Nwana 1996, p.5.). One definition agreed by agent technology researchers states that:

> *"An agent is an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives."*

(Jennings 1999, p. 280.)

Another definition of an agent is:

"A component of software and/or hardware which is capable of acting exactingly in order to accomplish tasks on behalf of its user".

(Nwana 1996, p. 5.)

Software agents are problem solving entities with well-defined boundaries and interfaces that are embedded into a particular environment. They have some sort of sensors that accept input information and they act through effectors. Agents are usually designed to fulfil a specific purpose. Software agents operate autonomously without external interaction having control over their internal state and their own behaviour. In general, a software agent is an entity to whom one can delegate a task and it will complete it autonomously (Jennings 1999, p. 280, Nwana 1996, p. 7).

A multi-agent system is a system that has multiple agents that are working together to fulfil the goals specified for the system. Agent-based systems provide a way of conceptualizing complex software applications that face problems involving multiple and distributed sources of knowledge. Multi-agent systems have the capability to provide several properties that are desirable for complex systems. These properties include robustness and reliability, speedup and efficiency, scalability and flexibility, cost effectiveness and reusability. (Zhang, Leung et al. 2005)

Figure 2 classifies software agents based on their basic characteristics. In the figure, agents are classified based on their emphasis on different characteristics. For example, a collaborative agent may have some learning abilities but their emphasis is on autonomy and cooperation. Collaborative agents work in autonomy while cooperating and communicating with other agents. Interface agents work as an interface between human users and machines trying to assist on tasks the user wants to complete, i.e. being some sort of personal assistants. Collaborative learning agents are capable of learning and cooperating with other agents, but does not operate autonomously. In general, they have very few practical implementations. *Smart Agents* try to have all the three characteristics, but in many practical purposes lesser implementation may be sufficient. (Nwana 1996, p. 6-7.)

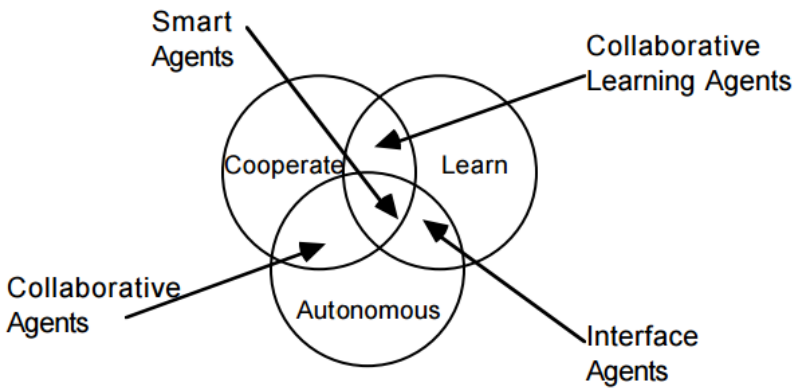


Figure 2. Agent type classification (Nwana 1996, p. 6.)

When applying the above agent classification, collaborative software agents are something to consider for a remote installation platform. To decide whether to select multi agent -based system or a single agent system the following formal equation may clarify the decision:

$$V\left(\sum agent_i\right) > \max\left(V(agent_i)\right)$$

Where V represents the amount of value added. The value could include several attributes, for example, such as worst-case performance, reliability, accuracy, adaptability or some combinations of these. (Nwana 1996, p. 10.)

Some benefits for having collaborative software agent systems include:

- to solve problems that are too large for a single centralized agent to complete due to resource limitations or the single-point-of-failure risk of having only one centralized system,
- to provide solutions to inherently distributed problems, for example, distributed sensor networks,
- to allow for interconnecting and interoperation of multiple existing systems,
- to enhance modularity that reduces complexity, increase speed achieved by parallelism, add reliability due to redundancy, enhance flexibility and reusability achieved by smaller autonomous (i.e. clearly separated) components,
- to help achieving graceful degradation where any failing components of the systems does not break down the whole system, but only partial functionality or performance that is in line with the percentage of non-working components.

(Nwana 1996, p. 7, 10.)

If the above properties provide essential value to the system, choosing a multi-agent approach in the system architecture may have a lot of potential. In addition, the static agent that runs on an operating system may be provided special privileges that allows it to alter the local system in ways that would not be easily or securely achievable with a single centralized agent that uses remote connection to gain access to different systems. Moreover, the agent can be native-like application to the target system which reduces the complexity of the overall system since one agent doesn't need to support varieties of system platforms. It is also not required to have constant end-to-end connection between the centralized system and a target system when agents are used. The agent can operate autonomously and if required, store any data to be sent later on when there is a working connection available. A single centralized system would be dependent to have a continuous connection to be able to complete tasks in remote locations.

Section key points summarized

- Designing software architecture on top of standardized and well-proved network procotols and architecture layers can help reduce the significant number of network related problems.
- Virtual private networks (VPN's) can be utilized to create secure connections between two networks through the open internet.
- Computers and networks may have firewalls that can restrict or block entirely the communication between two nodes in a network.
- Local software agent can provide privileges, access and information from restricted local resources to be utilized remotely over a network.

2.2 Problems in the remote installation system design

2.2.1 Describing believes, desires and intentions

Belief-desire-intention (*BDI*) software model is a model developed for programming software agents. When the system has multiple parts (components) that can change at any instant of time, it is possible that not all parties of the system know instantly what the changed state of the system is. Generally speaking, *beliefs* can be viewed as the informational component of the system state. *Belief* is the state where a party thinks that the system currently is. It may be true or false. *Desire* is information about the objectives to be accomplished including priorities. They can be thought to represent the motivational state of the system. *Desires* can be generated instantaneously or functionally. For example desire can be generated based on *beliefs*. *Intention* is the current course of action that the system or its component is taking. (Rao, Georgeff 1995, p. 312)

The behaviour of an actuating component that operates as part of the remote installation system can be described by using believe, desire, intention (*BDI*) paradigm. For example, when someone initiates a software upgrade procedure, the initiator has some sort of belief that in the system exist one or more parties that could be upgraded. The initiator desires to upgrade all or specified existing parties running an old version of the software. Its intention is to execute from one to many necessary steps that will lead to the desired state of the system. (Jennings 1999, pp. 277-278, 288.)

The target party running an old version of the software may believe that it has the latest version of the software until it receives new information about a software update. With information about important new update a desire rises that the party should acquire the new version and install it. The desire may lead to intention to fulfil the desire by

executing respective actions that will lead to the preferable outcome. Regardless of how to divide the system as multiple parts or as a single actuator, to manage and execute remote installations the system has believes about the current state of each running software instances, desires to reach installation goals and intentions to complete them.

Since there are no obvious limitations to describe a remote installation system as a combination of one or more agents, the *BDI model* can be applied here. For remote installation system, it is essential to get information about the underlying environment and the software applications running it. To maintain up-to-date *beliefs* based on the system state, updates of relevant changes in the system are required. To transfer information about relevant changes, some type of messaging is required. As described above, the change in *belief* may raise one or many *desires*. *Desires* should be fulfilled with *intentions*, which in remote installation system's case should be translated into a sequence of actions. The remote installation system requires ways to express and communicate these beliefs, desires and intentions both in the external world (i.e. system users) and internally to pass them from one component to another. Therefore the system design should solve how to represent and communicate all the beliefs, desires and intentions and how to take proper actions based on them.

2.2.2 Executing commands in remote location

Network access technologies are built on top of the idea that not all connections and their initiators can be trusted to have good intentions. By default, modern operating systems restrict external access and prevent users to alter the system itself unless they are explicitly permitted to do so. In addition, querying data about the system through the operating system's *application programming interfaces* (*APIs*) can be limited or even restricted completely. To complete installation tasks a remote installation system requires capabilities to execute system altering commands in remote system and means to query data about the current state of the system. Moreover, to be able to handle a variety of production environment systems, the remote installation system should work with the most common operating system versions that are running the target software installation tasks.

The remote installation system requires a way to pass its intentions to the target system and be able to either execute them directly or to get them executed by some proxy actuator. It also requires a way to extract data out of the target system. Both of these two cases require services from the target system's operating system but they also need to work with the end-to-end transfer channel. The operating system may provide access to functionality, but network tier may still restrict the access to operating system, which will prevent the overall system to work properly.

While the logical connection between the intention initiator and the target system may seem simple, direct link in the network architecture, the actual connection can have significantly more complex structure. *Figure 3* illustrates such complex architecture with components that can restrict the access between the communicating parties. The

connection route may have one or many firewalls or similar components that restrict the access to the target network resources (i.e. machines or applications). The firewalls operate on rules and policies that defines what kind of traffic is allowed and what is not. The rules and policies are defined by the parties that own the network components and they may vary a lot. In general, the rules are typically designed based on some guidelines and recommendations that help organizations to fulfil their network security goals and to protect against commonly known network vulnerabilities. Therefore, by expecting systems to follow some well-recognised guidelines that restrict network access will reduce the number of unexpected network connection related problems.

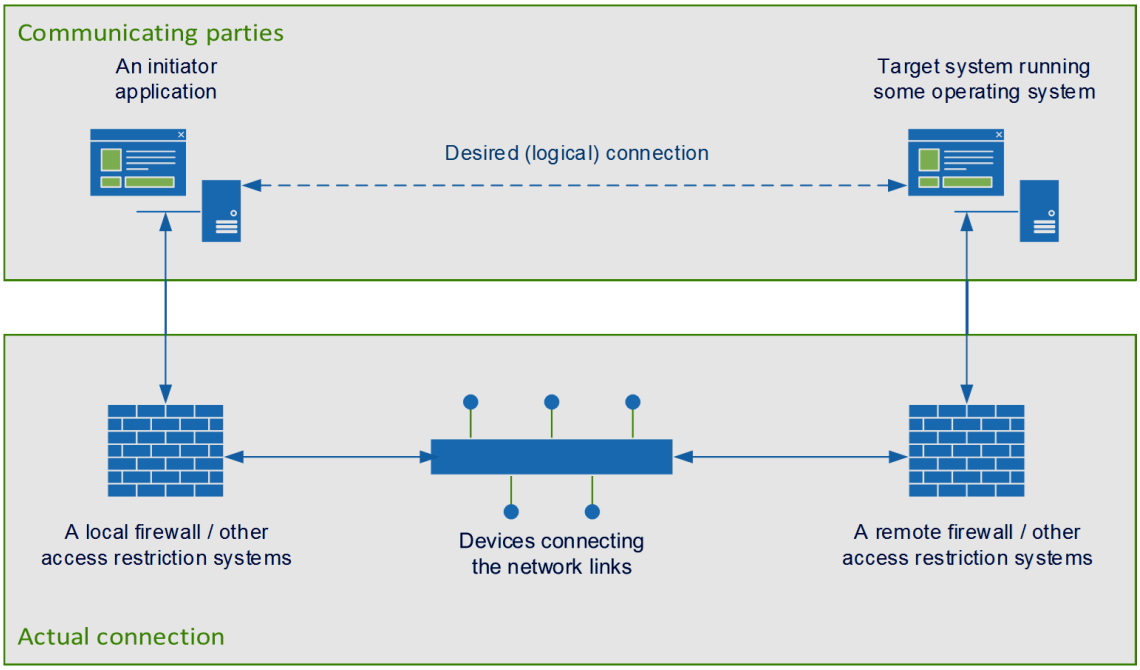


Figure 3. Example of underlying remote connection complexity.

Regardless of the underlying hardware or software infrastructure a remote installation system architecture should have the means to:

- express intentions that can be translated to set of commands understood by the target operating system,
- be able to pass the intentions through a network that may contain varying types of intrusion prevention systems and components that limits the access for most of the network traffic types,
- get the target operating system, desirable regardless of the specific version of it, to fulfil the intention and
- be able to get feedback from the intentions through the network channel.

The solution should adapt to the environmental conditions mentioned above and have a clear set of configurations, it requires from the underlying environment to be able to achieve the means listed above. The resource accessibility requirement list should be audible by third party security experts and include only items that can be realistically requested from professional parties that has high need to protect their system from unnecessary access. Every extra permission required by the system comes with a risk to be used by unintended parties. That means that the architecture should require the minimal set of different kinds of permissions to achieve its goal. Moreover, preferable solution would minimize the need to solve problems that are specific per network devices, security and intrusion prevention systems and operating system versions.

A system that includes multiple components (i.e. agents and processes) that communicate with each other has delays between the initiation of a command and getting the acknowledgement that the command was received or executed successfully. The system should tolerate such delays, but it should also define boundaries for each delays. If the system operates over packet switched network, it is probable that not all packets may find their destination. Also, it is possible that due to software design fault or anomaly in the system, a process or other software component may miss some messages that was sent to it. Therefore, in a robust system, there should be a time limit (a boundary) for the message sender party to wait for acknowledgement before it considers that the message was not delivered to the receiving party. To have such robustness it also requires that the messaging protocol used in each communication includes acknowledgements for each important message that is sent within reasonable time after receiving the message.

Summary of requirements introduced in this subsection

Req. 1	A remote installation platform should have means to describe intentions and transform them into executable actions.
Req. 2	The intentions should be transferable through the network passing all its traffic regulator mechanisms, e.g. firewalls.
Req. 3	The remote installation platform should have minimal integration or requirements for allowing its traffic.
Req. 4	The intentions should be executable in the target environment.
Req. 5	The actuator that executes an intention should be able to provide feedback to the party that creates the intention about the execution.
Req. 6	The messaging protocol should include acknowledgements and detection for non-delivered messages.

2.2.3 Updating system components' software

Software systems are subject to continuous changes. The need for making changes to the software may be due to changes in the surrounding environment, changes in external software components or changes in requirements. The change requests ask for new functionalities, for modifying existing ones, or for improving offered quality of service. To deal with the required changes, component-based distributed systems need to continuously evolve. Moreover, the process of updating software not only leads to improving user experience, but also has high importance for the safety and security of the system work as a whole. For example, in case of discovering a new exploit (a security vulnerability) the software producer can quickly correct errors in the application and minimize the resulting damage. Automated software updating plays an important part not only in desktop applications, but also in web applications. (Panzica La Manna, Crnkovic 2011, p. 1, Sitnikov, Rabasa et al. 2013, p. 106.)

When a software is upgraded, the process typically requires the software system to be shut down, updated and restarted. However, many applications require to offer continuous service. Therefore, it may be necessary to design the upgrade process as a part of the system architecture in a way that an upgrade does not cause discontinuity of any services provided by the system. The upgrade process design should consider at least the following things:

- *Correctness*: the entire update process must preserve any ongoing activities, which should complete their execution in the normal way,
- *Timeliness*: the delay needed for the update should be minimal,
- *Disruption*: service interruption should be minimal,
- *Human effort*: the update process should aim at being as automatic as possible.
- *Locality*: the update process should try to avoid depending on network resources and have all the information required to complete the operation locally available.

(Panzica La Manna, Crnkovic 2011, p. 1.)

As part of the correctness requirement, the update process should prepare for corrupted or non-working software upgrades and be prepared to do roll back to a previous working version. Rolling back to a previous version may be required in case the software stops working after the update is completed. Having update processes to complete quickly with minimal time consumption has a positive effect on causing minimal disruption to any ongoing usage of the system. Moreover, requiring no human effort can have the effect of the update process execution time. In addition, it also has decreasing effect on the maintenance effort required by the system and therefore can have a positive effect on the cost-effectiveness level of the system. The locality requirement decreases the changes of failures based on network infrastructure problems and has decreased effect to delays of data transfer. The locality can be achieved by preparing the update procedure by pre-loading any information required by it before starting the process itself.

To be able to update its own software a system should include *automatic update system* (*AUS*) component that takes care of keeping the software up-to-date without requiring human interaction. A modern AUS should meet the following requirements:

- *Safety:* The update procedure should be safe to from the operability point – it should never cause interference or interrupt any critical tasks or cause the system to become inoperable.
- *Sustainability in a case of unexpected errors:* The update process should be able to recover from any unexpected errors.
- *Scalability:* the update process should be able to scale as well as the system itself and not become a bottleneck.
- *Modularity:* the update process should be modular to cover all the requirements for different components of the system.
- *Extendibility:* the update process should be extendible in order to meet any changed requirements during the system lifecycle.
- *Integration into development process:* The update process should be integrated into the development process in order to make it easy to deploy new software version from existing deployment systems.

(Sitnikov, Rabasa et al. 2013, p. 106.)

Having the capability to update its own software the remote installation platform design should also include an architecture for managing component upgrades and being able to tolerate the disruptions caused by the upgrade procedure without losing operability. The solution should include ways to deploy (make available) new software versions, distribute them over the network to any remote system and to manage the procedure in a way that it does not cause failures of any other operations. In addition, the update procedure should be compatible with any other components and processes that are part of the system.

Summary of requirements introduced in this subsection

Req. 7	The system's own software's upgrade process should take minimal time and cause minimal disruption to the service it provides.
Req. 8	A remote installation system should have a capability to update its software without causing failures on ongoing operations.
Req. 9	A software update procedure should be fault tolerant and able to restore operability on unexpected errors.

2.2.4 Dependency for 3rd party components

Implementing a remote installation platform will most definitely include selecting a set of 3rd party software components that offer the required features in reusable form that

requires minimal integration effort to be applied in target software. Because of the cost-effectiveness, most of the software systems developed today are a combination of the application code and 3rd party libraries or components. Using 3rd party components considerably reduce the effort required for development of the software functionality in large scale application since the reusable components already provide varieties of features. However, using such components also bring new challenges. (Danek 2016, p. 457, Jezek, Ambroz 2015, p. 233.)

One of the challenges of using 3rd party software components in an application is the selection and composition of the right set of libraries. If the task is not done with care, the components may be incompatible, duplicated or redundant. The term *compatibility* is wide and the meaning usually depends on the context. Compatible libraries should behave as expected, which is often referred as to fulfil both functional and non-functional requirements. (Jezek, Ambroz 2015, p. 233.)

When software X uses software component Y, X becomes dependent on Y. In general, a dependency means that software X does not work without the component Y working as expected. When the software X has several dependencies of components that themselves have also dependencies it becomes very probable that there are some problems in the component interoperability. For example, some components may depend on different versions of the same dependency that is not backward compatible and therefore cause interoperability issues if the build procedure includes only a single version of the component into to the application deployment. (Danek 2016, p. 457–458.)

To be able to use a software component in a target software, it must be compatible with the software. To test if some software component is compatible with the software, the developer can use *compatibility testing*. In its simplest form, compatibility testing checks if software component Y can use or serve another software component X via published interfaces. (Ranganath, Crnkovic 2014, p. 469.)

As much as it is possible and in the control of the software developer, all 3rd party components should be evaluated to be sufficient to the purpose. In addition, a selection criteria should include also the compatibility of existing dependencies, e.g. by allowing only components that use the same version of shared dependency libraries. If the compatibility with existing components is not clear when assessing a new component, a good approach is to test it with proper *integration tests* before making the final decision to invest in integrating a dependency that the software will rely on. *Integration tests* are used to test how software components work together. (Jezek, Ambroz 2015, p. 233-234.)

Conducting integration testing is a good way to test how components operate together. However, integration testing is not sufficient way of revealing compatibility problems that are caused by dependency library collisions, since the integration test runner may use a different version of the dependency library than the actual application deployment and therefore be unable to reveal any runtime problems. Dependency library collisions should be analysed by other means as well, such as dependency graphs made out from the software code. (Jezek, Ambroz 2015, p. 233-234.)

Creating integration tests would require effort that may become obsolete later on if the candidate component will not be taken into use. In addition, the method's quality relies heavily on the test coverage. To reduce possible non-productive work, there are automated ways of running an analysis of the software dependencies that can help detect and solve most of the dependency problems in order to extract the list of candidate components from the less prominent ones. However, the automated tools do not guarantee that the component would work in the specific context. Therefore integration tests are still preferable to have. (Danek 2016, p. 457–458, Jezek, Ambroz 2015, p. 233.)

Software dependencies may also be other types than software libraries that are linked and shipped with the software itself. They may be, for example, tools (e.g. data compressor or script runner), frameworks (e.g. Microsoft .Net Framework) or hardware drivers that are required to exist in the target environment where the software is deployed. These dependencies share same compatibility issues with the library dependencies that are shipped with the software. The significant difference is that the software developer has almost no control over the software versions or the availability of these tools and components in the target environment. Therefore the developer should verify the compatibility list of different versions of these components and either implement verification tests for these dependencies in the software itself or provide a clear requirement list of the compatible 3rd party components that should exist in the target system. The best solution from user's point of view would be of course to have both, since that would provide the maximum visibility for the dependency issues.

Summary of requirements introduced in this subsection

Req. 10	Selected 3rd party software components should be sufficient for the purpose as well as compatible with each other and the target software.
Req. 11	The list of dependencies and their compatible versions expected to be in the target system should be known and available for the user.

Section key points summarized

Believe-desire-intention (BDI) paradigm can be used as a basis for describing, classifying and implementing software agent's behaviour.

Executing commands remotely require means to pass the commands, to get the target to execute them and to receive confirmations and feedback from the procedures.

Software technologies often come with dependencies to 3rd party components that are required to exist in the target system and where the software designer and programmer usually have very little or none control over.

2.3 Designing architecture for automated remote installation system

An automated remote installation system has a lot of complexity. There are some well-known techniques that can be used to ease the designing task for complex systems. The first technique is called *decomposition* which is the very basic technique – tackling large problems by dividing them into smaller, more manageable parts that can be dealt with in relative isolation. *Decomposition* limits the scope that designer needs to consider at a time. Practically, it means that when working on any portion of the full design, only a part of the whole problem needs to be solved at a time. The other technique is an *abstraction*. In *abstraction* a simplified model is used to emphasize some of the details or properties while omitting other complexities. This helps to focus on specific parts of the problem at the expense of the less relevant details. The third technique is *organization*. It is the process of identifying and managing the interrelationships between different problem solving components. *Organization* can be used to group basic components together and treat them as a higher-level unit of analysis. The method can help reveal the high-level relationship between component groups. (Jennings 1999, p. 282.)

The word “remote” in automated remote installation suggests that there is some distance between the installation executor and the place where the installation physically takes place. To occur, the installation requires some sort of input and control of the process. The input must be deployed somehow to the physical location. The control also requires some form of transaction between the executor party and the physical system where the installation takes place. These statements lead to a need for some sort of transfer channel, a network which connects the two parties and that is capable of transmitting data from one party to another.

The “automation” of installation requires that there is some pre-defined set of tasks that when combined, formulates a process that is considered as a single installation. These tasks may require input and have some conditional execution paths that require some decisions. Requiring user interaction during the automated process execution is

undesirable as it decreases the level of automation. All required decisions should have predefined answers as part of the input for the installation, or the system should be capable to resolve them otherwise. The kind of installation that has multiple different installation scopes per different systems is very likely to have conditional execution paths and therefore requires for some well specified control workflow to follow.

In addition for requiring input and transfer channel, automated remote installation requires some way of generating the proper input for a specific use case and some actuator that executes the process based on the given input. Moreover, since there is a transfer channel involved it is also required to be able to *serialize* the input data into a format that can be sent through the transfer channel. In addition, the data should be possible to be *deserialized*, hence restored to proper format in the receiving end.

Developing the practical implementation of a complex distributed system is a challenging task from a functionality and performance point of view. Dividing the problem of creating such system into a set of smaller problems is one known approach to make the task easier. Using *software agents* in *service-oriented-architecture* (SOA) based on *belied-desire-intention* paradigm can help reduce the complexity issues, support frequent changes, inconsistency and provide rollback functionality. (Kusek, Velásquez et al. 2009, p.48)

The following sections elaborate the subjects mentioned above. Each subject is part of enabling the automation and remote controlling the process over the network. They provide solutions for specific set of problems that must be solved in order to produce a system that provides the overall service for managing automated remote installations.

Summary of requirements introduced in this section

Req. 12	It should be possible to pre-define an installation process in order to automate executing it.
Req. 13	It should be possible to execute installation with input that applies the installation to a specific target system.
Req. 14	Installation input data format should support serialization and deserialization.

2.3.1 Remote installation system as service-oriented architecture

Having a system that executes remote installation over a network requires that there are at least two nodes in the network (*Figure 4*). The two nodes include the target node that accepts remote connections and input from the network interface and the source node that provides the information and sends the data. It is possible that these nodes are physically the same, but the two roles still exists. This smallest possible network of remote installation already suggests that in a system that handles remote installations,

there is more than one component that provide services to each other. The receiving end provides a service that accepts instructions and possibly report the results and the sending end provides services that are required by the receiving end as an input. When adding a user to the system, it suggests also to have some number of services that provide an interface for the user to control the installation and to receive feedback. Moreover, while the list of features and requirements grow the remote installation system becomes more and more complex.

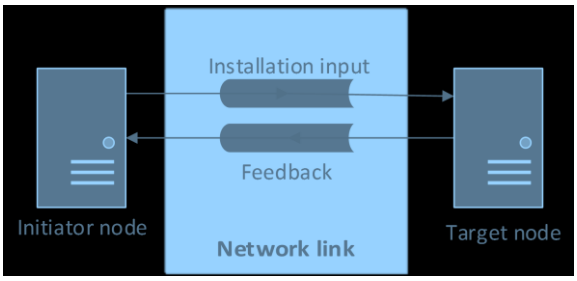


Figure 4. Simplified remote installation system.

When the system operates in a network environment, it already suggests that there are multiple components included. These components could interact with each other by providing services with clear and well-formed interfaces that accept specified messages with data payloads. Such system is using an architectural pattern called *Service-Oriented Architecture (SOA)*. *Service Oriented Architecture* was raising interest in the first decade of the 21st century because it was said to provide solution for having reusable and easily replaceable components. When the target product that is installed changes over time it is also likely that the requirements for a remote installation system may change over time. *SOA* can adapt to such changes. Small portions, single services, of the system can be replaced without modifying the whole system. That makes *SOA* architecture to be a viable candidate for remote installation system architecture. (Erickson 2008, p.44, Hau, Ebert et al. 2008, pp. 2-5)

To describe the architecture as a network of nodes connected to each other, let's assume that each machine of interest, server or workstation represents a single node in the network. In *SOA*, each node can have one or more components that provide services to each other. *SOA* adapts well for a remote installation platform since there are going to be at least one node that provides the input, e.g. new versions of the software, and from one to many nodes that needs to receive the input and to do something with it. When each of these nodes are clearly separated with interfaces and connectors, adding and removing nodes should become less complex. (Hau, Ebert et al. 2008, p.2)

Altering a single component should become less problematic when the whole system does not require changes to stay compatible unless the interfaces are changed. In a well-planned solution the interfaces should not change in a way that it breaks backward compatibility. The larger the network of nodes grows, it becomes less likely that all the nodes are available for updates when needed and hence it becomes harder to keep all

the nodes operational if some interfaces are suddenly becoming incompatible. (Hau, Ebert et al. 2008, p.2)

Summary of requirements introduced in this section

Req. 15	Updating the network platform software should not cause any nodes to become incompatible and not being able to become compatible.

2.3.2 Multi-agent systems

Multi-agent systems (*MAS*) are the group of agents which have different tasks and work in cooperation, coordination and in communication with each other (Mishra, Srivastava 2014, p.1). Moreover, multi-agent systems are distributed systems, composed of a number of autonomous software entities called agents (Poslad 2007, 15:1). They can support solving varieties of problems automatically. To work with most complex issues, solving the problem needs to be shared by many different entities. The advantage of *multi-agent systems* is combining intelligence with coordination of agents in the system. (Khue 2012, p.684.)

MAS represent a powerful model to solve distributed computation problems. The agents can adapt their operation in open and dynamic environments in which the content and workload are continuously changing. Multi-agent platforms are able to utilize other agents for cooperative distributed problem solving when individual agents don't wish to or can't perform tasks within certain constraints or do not have the competency to perform the set tasks by themselves. Software agents need to be able to transport messages, to discover which services and capabilities other agents can offer them, and to be able to utilize other providers' services. In addition, they need an execution environment and management services including storage and security. It is also likely that the agents need to be able to utilize non-agent software services. These kind of middleware services is often provided by the MAS platform, in which the agents are embedded. (Poslad 2007, 15:2.)

The *Foundation for Intelligent Physical Agents* (*FIPA*) is an international non-profit association of companies and organizations sharing the effort to produce specifications for generic agent technologies. FIPA has created a specification (FIPA97) where they describe a reference model of an agent platform (*Figure 5*). In the reference model, they have identified some key roles for managing an agent platform. The reference model contains three mandatory roles: the *Agent Management System* (*AMS*), the *Agent Communication Channel (ACC)* and the *Directory Facilitator (DF)*. (Bellifemine, Poggi et al. 2001, p. 105.). The Internal Platform Message Transport (shown in the *Figure 5*) was originally modelled as an agent, but was then changed in the later specification to be a non-agent functionality since it would be inefficient to separate messaging from the sender agent. (Poslad 2007, 15:13.)

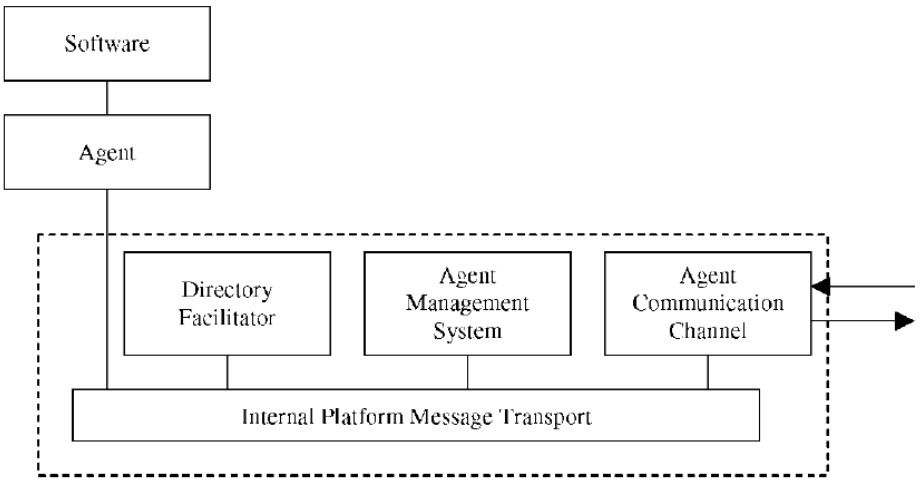


Figure 5. FIPA reference model of an agent platform.

The *Agent Management System* is the agent that possesses the supervisory control over access to and use of the platform. In addition, it is responsible for upholding a list of agents that are part of the platform and for handling their lifecycle. The *Agent Communication Channel* provides the communication methods and paths for basic contact between participant agents and between the outside world and the platform. In FIPA's reference model the *ACC* is the default communication method that offers a reliable, orderly and accurate message routing service. The *Directory Facilitator* is the agent that provides network information services, such as hostnames and addresses to the agent platform. (Bellifemine, Poggi et al. 2001, p. 105.)

Agent platform is capable of bringing together a number of service capabilities to form a unified and integrated execution model that can include access to external software, human users and communication facilities. The FIPA specification (FIPA97) defines *Agent Communication Language (ACL)* that is based on message passing. In the reference model agents communicate by sending individual messages to each other. The FIPA ACL is a standard message language for agents and it sets out the semantics, encoding and pragmatics of the messages. It does not specify any specific mechanisms for the transportation of messages. The messages are encoded into a textual form that allows the implementations to use different networking technologies and platforms. It is assumed that the agent has some means of transmitting the textual form of messages. (Bellifemine, Poggi et al. 2001, p. 105.)

The ACL semantics of the content typically defines shared information and tasks with respect to some domain, such as a specific service or application domain. The semantics of the communication context defines the relationship between specific messages in relation to context. The communication semantics can be for example the sender agent's current work flow, goals and plans, an agreed interaction protocol, or with respect to a receiving agent's beliefs, desires, and intentions. (Poslad 2007, 15:2.)

In their research paper "*AOCD: A Multi-agent Based Open Architecture for Decision Support Systems*" Zhang et al. (Zhang, Leung et al. 2005) present an *Agent-based Open Connectivity for Decision support systems* (*AOCD*) architecture (*Figure 6*) that has a unique component called *Matrix*, which combines the roles suggested by FIPA in their specification into a single component. The architecture they present, suggests that the *Matrix* component is the facilitator of the platform where all the other agents connect to in order to access centralized data source and to get information about other agents and their capabilities. In addition, in the reference architecture the user is connected to the platform through a user interface that is connected to the *Matrix* component. The architecture supports centralized, decentralized and hybrid network topologies and provides clear separation between the platform components. The presented architecture can be applied to a various kind of multi-agent systems. (Zhang, Leung et al. 2005, p. 2-3.)

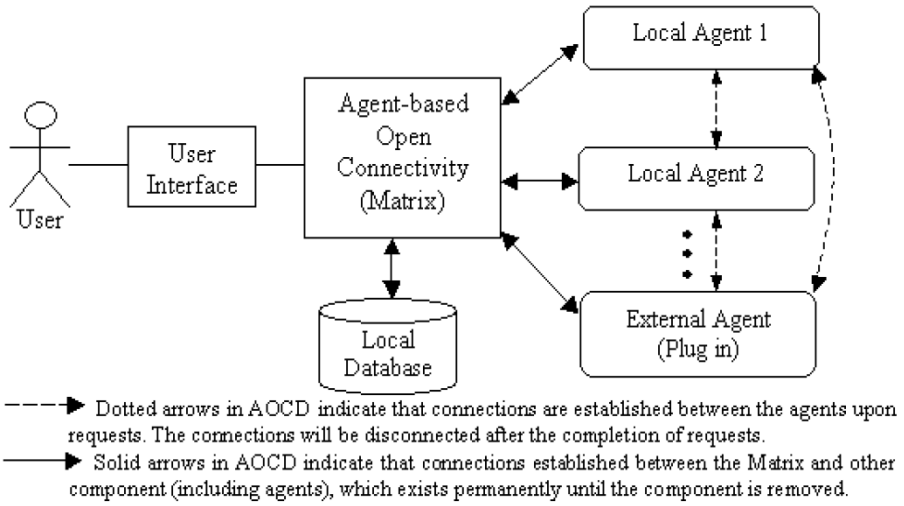


Figure 6. A conceptual view of the AOCD architecture. (Zhang, Leung et al. 2005, p. 2.)

In a research paper "*Development and Specification of a Reference Architecture for Agent-Based Systems*" Regli et al. (Regli 2014) present an agent system reference architecture (ASRA) that can be used in designing a multi-agent system frameworks. According to the ASRA architecture the functional concepts of an agent system are defined as the following:

- *Agent administration* facilitates and enabled control and command of agents, and manages resource allocation according to need.
- *Mobility* facilitates and enables migration of agents among framework instances, which, typically, but not necessarily are located on different hosts.
- *Security and survivability* allow execution of desirable actions and prevents execution of undesirable actions in the agent system.
- *Conflict management* facilitates and enables the management of interdependencies between agents' decisions and activities.
- *Messaging* facilitates and enables data and information transfer between agents that are part of the platform.

- *Directory Services* facilitate and enable the locating and accessing of shared resources.
- *Logging* facilitates and enables information about events to be recorded that occurs during operation execution for subsequent inspection.

(Regli 2014, p. 149.)

The ASRA architecture does not intend to define specific implementation details of a multi-agent platform, but rather describe potential interactions between the functional concepts mentioned above (Regli 2014, p. 149). The functional concepts listed above provide a clear way of separating an agent architecture into a smaller components, i.e. into smaller problems to solve at a time. The proposition divides the architecture into components that are considered from a functionality perspective. Therefore, when a functionality requires to be changed, becomes obsolete or needs replacing, in general, it should affect only a single component in the system.

All the items in the functional concept list are managing responsibility areas that are also likely to be relevant for a remote installation platform that is designed as a multi-agent system. Therefore the list provides a good starting point of component separation also for a design proposition for a remote installation platform.

Summary of requirements introduced in this subsection

Req. 16	A multi-agent platform should have at least three roles: Agent Management System, Agent Communication Channel and Directory Facilitator.
Req. 17	A multi-agent platform should enable agents to be able to communicate, discover and invoke services from each other as well as from some non-agent software services when the agent cannot complete a task individually.
Req. 18	A multi-agent platform architecture should divide the agent design into independent components that have clear responsibilities to manage the functional requirements set for the system and are easy to modify or replace.

2.3.3 Properties of a system operating in network environment

In its simplest form a reliable remote installation system that follows the BDI software model includes at least two types of messages. There are *request*-messages that should provide commands and instructions to the target machine. In addition, there are *inform*-messages that should provide feedback from the requested operations and reports of current state of a node. It is very likely that proper remote installation software requires several message types to handle all the communication requirements. Therefore, some kind of messaging protocol should be implemented that

supports passing the commands, feedback and some heartbeat traffic to detect dead nodes. In addition, the software that handles the protocol in the target nodes should have the ability to update itself since software may always have bugs and the requirements towards the system may change over time. (Kusek, Velásquez et al. 2009, p.46)

Resourcing and maintenance

Network aspect in a system design includes some key points that should be taken into consideration when designing the system architecture. In a robust solution, the software running the node instance, should have capabilities to recover or restore from cases like temporary network connection breakings, restarting of the host machine operating systems and unexpected errors caused by invalid input in network messages. If the network grows large in number of nodes, it becomes very quickly unbearable if a single node administration requires manual labour or resources from some centralized resource pool. A growing network of nodes where every new node consumes some amount of shared resources eventually leads into bottlenecks that causes scalability issues. Therefore, it is one good aspect to consider when designing a system that operates on a network, that how can a network node be as self-maintainable as possible.

In addition to the key points to include in the system design, the computing requirements and where to place the computing load should be considered. If a node operates in an environment (machine) where other software elements require significant computing resources from time to time the host machine will have computing resources available regardless of the new requirements that the designed system defines. Also, if the node operates in a host that is connected to a stable power source, i.e. not a battery-powered source, there are less restrictions to place computing resources in a node. In a such system, where every node's host has computing resources available, it makes more sense to design the system in a way where the computing resource usage is distributed to the nodes instead of having some centralized computing resources dedicated to the system.

Data transmission and network bandwidth requirements

In a remote installation system the amount of data required to be transferred dictates the network bandwidth requirements. The size of transferred data can be reduced by using some compression algorithms for the data. Typically, data compression is done by applying several different compression algorithms one after another to further reduce the size of the produced outcome. Each of these algorithms has their own time-complexity to apply and to reverse them. Adding a new algorithm to the compression operation can reduce the compressed data size, but it also increases the required computing effort on both compression and de-compression operations. In general, when using most advanced technics for compression, the combination of algorithms that produces the best possible compression ratio takes significantly more computing resources compared to some lesser technics that has a lower compression ratio. Formally, a data compression function with required computing and compression ratio increment does not have linear growth since different data compression algorithms or technics have different complexity levels.

Decreasing the data size transferred over the network through compression increases the computing resources required by the both ends of data transmission. The system design should define the acceptable levels for both required network bandwidth and computing resources and based on them, select the proper parameters for used compression rate. Typical compression software applications have some recommended settings that use the set of algorithms and memory resources that provides the best compromise for compression ratio versus required resources. Such settings can be used as reference when researching the best solution for compression. The Bottom line is that some level of compression for the data is preferable since installation packages that has database migrations and lots of plain text type of data tend to have a lot of potential for compression even with the most basic compression algorithms. In addition, the most basic compression algorithms do not require significant computing resources to compress and decompress compared to typical available computing resources of a *CPU* built inside last decade.

Data transfer reliability and security issues

In a network environment, any node can become temporarily unavailable. The Transfer Control Protocol is designed to handle cases where a single data packet transmission fails between two links in a network. The network layer protocol design does not cover longer periods of unavailability which should be handled by the application layer protocol that is the software running a node instance. The offline period can be caused by network level infrastructure malfunction or, for example, a software update for the node software that requires disconnecting the service. In order to recover from offline periods a node should maintain some status information about what data has been sent and received and an identifier for that specific data. Having an identifier helps building detection of the condition when a node has become unavailable and to get that node into a consistent state after the connection returns to working state.

The system operating in network environment should also have some consideration of security issues in case there is any possibility that it can be accessed from external sources. There should be at least consideration towards the data sensitivity and the content protection that is transferred between the parties. Also the authentication issue of how to verify that the correspondent party is really the real target node and not some unknown party should be covered by a solid security plan.

Summary of requirements introduced in this subsection

Req. 19	An agent network should have a messaging protocol implementation that supports passing commands and information from one party to another.
Req. 20	The agent software should be updatable in order to correct software faults and adapt to the changed requirements.
Req. 21	An agent should have ability to update its own software automatically when the agent software distributor has a new version available.
Req. 22	An agent software should have the ability to recover from problems like system restart, offline time and temporary network connectivity issues.

Req. 23	An agent node should be self-maintainable and hardly ever require manual maintenance done by human administrator.
Req. 24	An agent should not require more resources that are realistically available in any target environment it is installed into.
Req. 25	An agent implementation should consider the balance between available network bandwidth and computing resources.
Req. 26	Sensitive data transmission should be avoided through open network or the data should be protected by appropriate measures.
Req. 27	An agent should have a way to verify authenticity of the other party it is communicating with.

2.3.4 Defining input data structure for automation

The data that is required in order to achieve the goals of the remote installation system that operates on network environment has at least three states. Data in the different states may use same or different formats. The first state is manifested where the data values are defined. The usage in first state may prefer human-readability (i.e. human-comprehension) over machine-readability if the source for the data is an interface designed for human users. The second state manifests in the format where the data enters to the transmission channel. It requires that the data can be encoded (*serialized*) and decoded (*deserialized*) based on the data transmission protocol requirements. In the second state the human-readability is not as important as is the machine-readability since the data is interpreted by two non-human components. The third state is the format where the data is consumed as input for the actuator part of the system that requires the data in its operations. These states are covering only the data formats that are handled at the application layer, not the lower level formats described in OSI-model's, internet protocol suite's (TCP/IP model) and other similar models' specifications. In addition, more complex systems may have multiple data transformation phases between different formats, but in this definition, they are abstracted as part of the second state.

The transition from one format to another should be lossless, i.e. no data should be lost in translation from one state to another. It is not required to have a minimum of three different forms of the data to build a system with networked nodes. Having forms of the data that are more suitable per purpose helps optimizing the end-to-end transmission when the format is designed based on the characteristics of each component. With complex data structures it is rare that the same format would serve well both human-readability and machine-readability, and be efficient to use. On the other hand, it does not provide a very good user experience if the user interface requires providing the data in machine-readable format that is not intuitive for humans. Therefore, for efficiency reasons the data should be provided in human-readable format when human interaction is intended and for all other purposes, it should be maintained in a format that is optimized for machine-readability and can be efficiently passed through from one component to another with minimum overhead coming from interpreting (parsing) the data. From maintainability and software implementation point, the format should be

selected to be such that requires less effort in implementation. Preferably, the used format should have some well-tested existing implementations for transforming and interpreting done in the programming languages used by the system implementation.

The input data format can be designed to be able to describe an executable process as a control workflow, for example, an installation process workflow. The goal is achieved by including a *workflow process definition* as part of the input. *Workflow process definitions* are instantiated for specific cases. They are defined to specify which activities need to be executed and in what order. A workflow process definition defines the *control flow* of the process. (van Der Aalst, Wil MP, Ter Hofstede et al. 2003, p. 5.)

Having an automated process it is required that the automated process control flow is modelled into some form. If the process can be defined in the input data as a control flow it can be dynamic. Moreover, to be able to execute an automated process, the agent requires less pre-knowledge of the task, if the input data can describe the whole process flow with specific instructions for the execution.

The most basic workflow pattern is called a *sequence* where an activity in the workflow process is enabled after the previous activity has completed. A process such as an installation process is not always a static sequence of actions and can vary across environments. Therefore the process flow description data format should have a modular structure that can support multiple execution paths required by each variation of the modelled process. Such pattern in control workflow is named as a *multi-choice* pattern (Van der Aalst 2003, p. 6, 9.).

One way to achieve modularity is that the process description input format is formed into *execution blocks* where an execution *block* describes some part of the process, an activity, that can be further divided into smaller execution *blocks* placed inside the parent *execution block*. Each *execution block* should have a clear condition when to be included or excluded. *Figure 7* provides examples of some multi-choice workflows described with *execution blocks* and conditional execution paths.

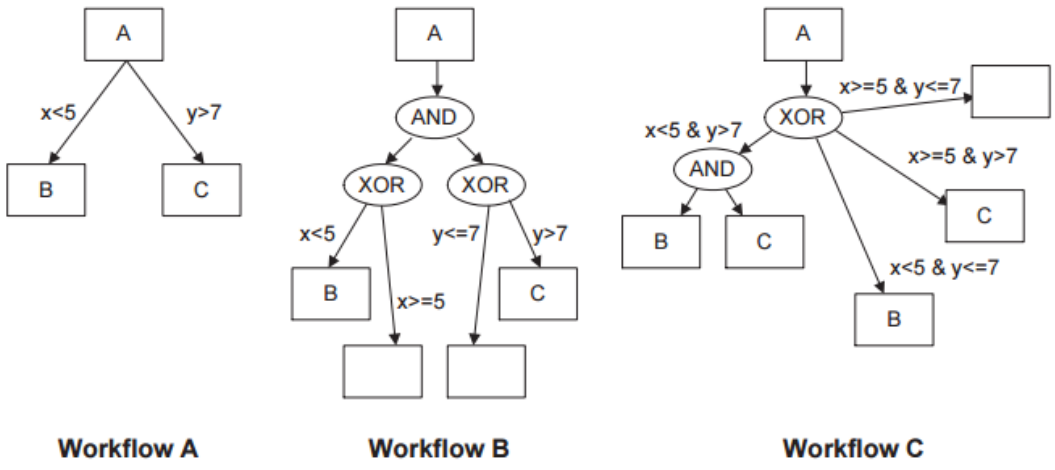


Figure 7. Examples of multi-choice workflows with conditional execution paths. (Van der Aalst 2003, p. 11.)

The examples provided in the figure above are rather simple. The actual installation automation process may require more complex control flow patterns in order to meet all of its goals. The selected workflow patterns are not in the primary focus of the research. More important goal is the ability to define such processes and to be able to transfer them as intentions in a multi-agent system that can be executed by the target agent in the same way the source intends it to be executed. For that reason, the ability to transfer the workflow describing an installation intention without losing any data in the process is the primary goal for the input data structure.

Summary of requirements introduced in this subsection

Req. 28	To improve efficiency the input data format should prefer machine-readability whenever it is not visible for human users.
Req. 29	When the input data is transformed from one form to another, no information should be lost in the process.
Req. 30	The input data format should be able to contain an installation process control flow for all the relevant installation cases.

2.3.5 Network topologies

The remote installation system can be designed by using different kinds of network topologies. Each topology comes with limitations, advantages and disadvantages. When the goals of a multi-agent system include communication effectivity and efficiency the consideration in designing the proper network topology should precede the designing of the agent network itself. Agent network topology analysis can enhance the agent communication efficiency of an agent network and provide efficient mobility and intelligence to the network. (Zhang 2006a, p.22.)

When deploying software components on multiple application servers at remote locations using software agents, a major challenge is to determine the number of operational agents to use. Deployment efficiency depends on having the proper number of agents. To manage multiple remote installations in a network environment, one of the following strategies can be applied in agent-based network-centric system:

- a single agent executes all required tasks on all application servers,
- an agent executes a single task on one node only,
- an agent executes all tasks on one node only,
- an agent executes a specific task on all nodes,
- tasks are assigned to agents in order to achieve maximal parallelism in task execution,
- a hybrid solution that combines some of the strategies listed above.

The simplest model is to have only a single agent that handles all the installation tasks for all the application servers. That would result in very simple architecture topology but also have set of limitations. The opposite approach is to have one or many dedicated agents per application server. The optimal number of agents depends on the conditions and requirements and lies probably somewhere between the two opposites. (Kusek, Velásquez et al. 2009, p.48-49)

Local Area Network (*LAN*) theory generally defines four basic topologies, which include star topology, bus topology, ring topology, and tree topology. More complex topologies can usually be divided into these simple topologies. Based on the *LAN theory* the simple agent network topologies can be classified into the following categories:

- centralized agent network topology (i.e. a star-like topology)
- peer-to-peer agent network topology (i.e. a mesh-like topology)
- broadcasting agent network topology
- closed loop agent network topology
- linear agent network topology
- hierarchical agent network topology

(Zhang 2006a, pp. 23-26.)

In a *centralized agent network* topology (*Figure 8*) there is one centralized facilitator that initiates operations in other nodes and one dedicated installation agent per each application server. Apart from the centralized nodes, agent nodes are not directly connected to each other. A Star-like topology is one of the common cases of centralized topology. It is a simple topology where each agent is clone of each other from the software perspective and there is only one centralized facilitator system that acts as a system operator for each agent. The total number of connections in *centralized agent network* topology is the total number of agents in the network minus one. The most obvious limitation of such network topology is that there is a single point of failure. If some offline time for the whole system is acceptable, then the single-point-of-failure disadvantage may not be that critical. (Zhang 2006a, p 23.)

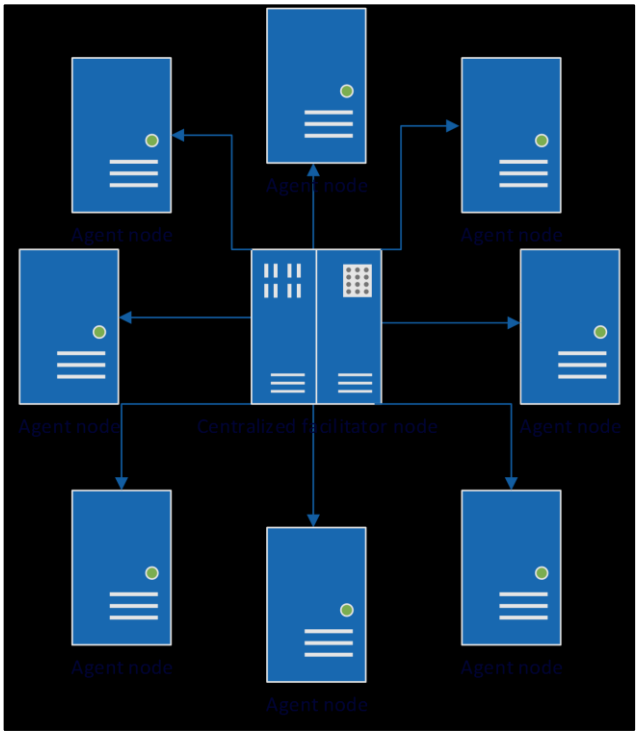


Figure 8. Star network topology applied for centralized agent network topology (Zhang 2006a, p.23).

A single, centralized resource may become a bottleneck for the whole system's scalability. However, there are technics that can be used to reduce the size of that problem. For example the centralized system can include load balancer that

distributes the resource requests to multiple available back-end resource providers (e.g. servers). From the network topology perspective, that kind of details does not change the architectural structure.

Peer-to-peer agent network topology has multiple direct connections between different agents. It may be fully connected where each agent has connection with each other or partially connected, where some agents have connection with each other and all agents have connections to some agents (Zhang 2006a, p.24). For this specific remote installation system a limiting factor for selecting between network topologies is that nodes which are located inside one customer's environment cannot be directly connected to other nodes that are located in other customer's environments. That restriction limits usage of mesh-like peer-to-peer agent network topologies. In addition, the remote deployment provider is not responsible for the network connection conditions in any customer's network. Therefore, they have very little weight on decisions for remote deployment system design.

In *broadcasting agent network* topology all agents are connected through a common medium and there is no direct connection between any of the two agents. In such networks, every agent has the same role in the network. Bus topology is a common example of a *broadcasting agent network* topology (Zhang 2006a, pp. 24-25). In broadcasting agent network topology, all the communication messages are available for all the agent nodes. That may not be preferable for a remote installation system from the same multi-customer issue perspective that prevents using *peer-to-peer agent network* topologies. Moreover, for the similar reasons ring-like *closed-loop agent network topologies* and chain-formed *linear agent network* topologies are not applicable topology options for the remote installation system.

An alternative architecture for a remote installation system can be built on top of treelike *hierarchical agent network* topology (*Figure 9)*. In *hierarchical agent network* topology a number of agents form a group, which is connected to an upper level agent. In such topology an agent is not connected to other agents, except to its upper level agent. Total number of connections in *hierarchical agent network* topology is same as in centralized topology which is the total number of agents minus one. (Zhang 2006a, p. 26.)

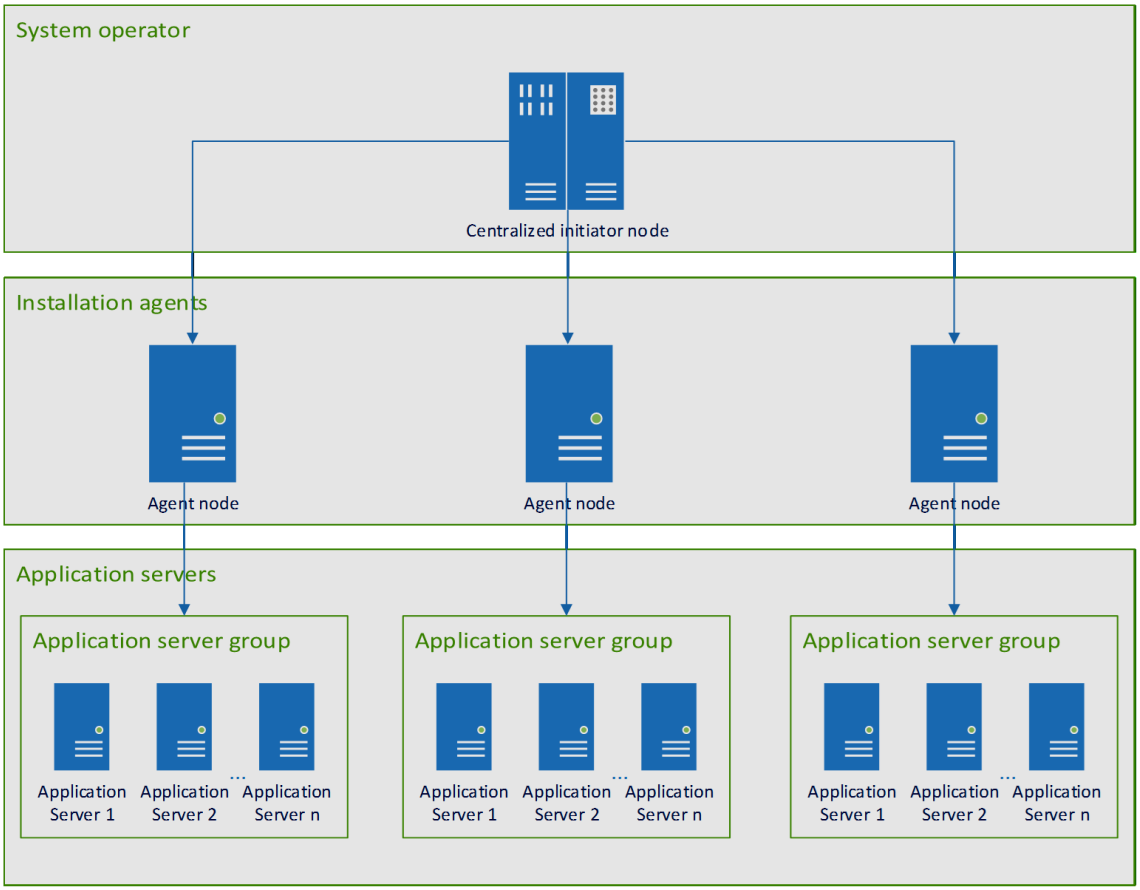


Figure 9. A tree network topology applied from a hierarchical agent-based remote installation system network topology (Zhang 2006, p. 26).

Using a hierarchical agent network topology provides a chance to use more hierarchical network structure where nodes can belong to different layers that have hierarchical order. In hierarchical tree topology a single installation agent placed in customer network can handle installations for multiple application servers that are part of the same network. The network can have restricted or non-existent access from outside of the network. In such a setup, the agent requires only a single open port to the internet that the system can use for connecting.

The architecture in the tree topology figure (*Figure 9*) can be extended further with new hierarchies that can increase the resources and number of agents inside a closed network. For example, a customer environment could have a root node that is connected to operator in order to get installation packages and other input data. The root node would then distribute the same data to lower level agents that handle installation for different application server groups. One advantage for such setup could come from the security point of view. If the internal network is wanted to keep fully isolated from the open internet, single root node in a customer environment, placed into the demilitarized network zone, acts as local centralized facilitator. Then the installation data from the main central node could be imported manually to the local root agent which requires small amount of human interaction. That data could then be distributed automatically to multiple agent nodes that handles the installation of the whole internal network.

Redundancy is one of the issues to consider in the network topology selection. In a network that describes application system, a single link may include multiple devices that provide some level of redundancy from the hardware perspective. In addition to hardware redundancy, the remote installation system should consider having some redundancy also for the software instances. For reliability reasons the system should include multiple agents that can install the target software to the same application servers. For example, if the time window when an installation must be completed have to be very specific, the system should include secondary agents that can be initiated to do the installation if the primary agent is unavailable for any reason.

The remote installation system architecture could abandon the idea of a centralized system that requires more resources for each agent node connected to it. To get rid of centralized distribution system a *peer-to-peer* (*p2p*) technology could be used to have all the participant nodes to be included in the software distribution effort. The benefits of *p2p* increases when the network size and the amount of data transfer required for the distribution grows very large. The problems in such architecture include security issues. For example, knowing that the source is reliable and the content has not changed from the original form becomes more complex issue. Moreover, it could raise some customer concerns about having direct connections from their system to the competitors' systems, which can become a mean for company espionage. In addition, the p2p network may not have the stability and robustness required by the professional software distribution system designed for company customers.

Summary of requirements introduced in this subsection

Req. 31	Agents running in one customer's environment cannot be directly connected to other agents running in other customers' environments.
Req. 32	Having a strict installation window the agent network should have a level of redundancy of multiple agents being capable of completing the same goal.

2.3.6 Choosing the software component dependencies

Today's drive for cost-effectiveness in software development business demands that the company considers carefully which parts of the software they want to implement themselves and which components can be bought or get for free of charge by using some of the available open source solutions. The cost of developing software in-house does not limit only to the labour costs of time spent on development and testing the software. The software also requires maintenance. In addition, unless and sometime regardless whether the software is licensed as open source, all the effort required to maintain the software is covered by the company itself. A benefit of having parts of the software done by some 3rd parties is that some of the testing and maintenance effort is then handled by others.

Using 3rd party software components that have long maturity, larger user base and possibly has reference based proof of scaling abilities can lead to more robust and higher quality products. On the other hand, there may be some uncounted risks included when using them. One risk of using 3rd party software components is that there may not be any guarantees that the components are supported and maintained in the future. It may lead to long term security issues or other types of severe faults in the components.

As well as the remote installation platform itself, also all the dependency components should be maintainable and have the possibility to be updated in order to correct faults, improve performance characteristics, adapt to changed environment or to improve their maintainability. In general, software deployment and maintenance strategies are important in order to configure distributed systems, for example, multi-agent platforms, according to their requirements. In addition, for performance and cost-effectiveness reasons the configuration and re-configuration (i.e. update) setup times should be minimalistic in order for them to have minimum influence on the system operability. (Kusek, Velásquez et al. 2009, p. 46.)

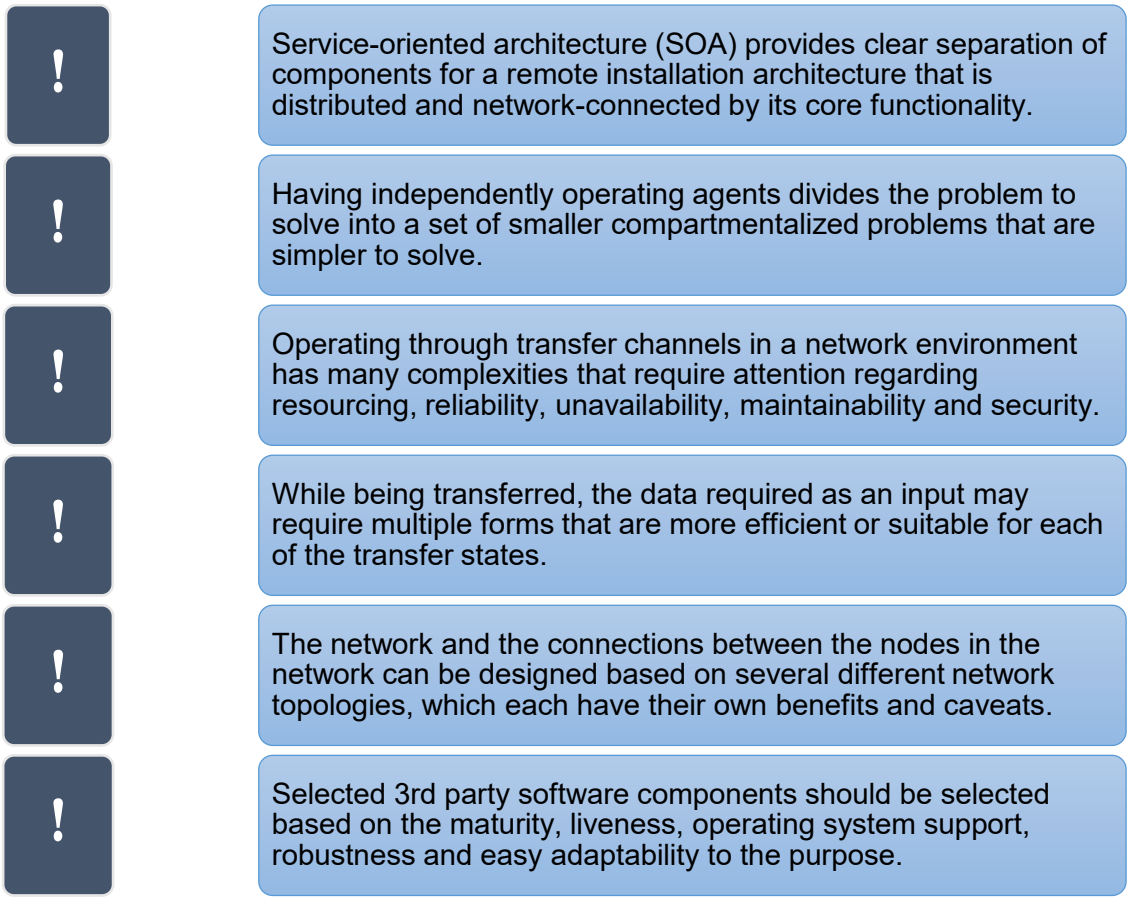
Because of the possible risks to the software product quality, all the included dependencies should be considered carefully. The risks can be diminished with some long term considerations. In addition to being sufficient for the purpose, including a dependency component comes with less risk if it can be replaced relatively easily, for example, in the case that the component's support ends or it become insufficient for the purpose.

Modern software applications are expected to run on a variety of platforms with diverse software and hardware level features. The component's cross-platform support should also be evaluated per need. For example, if the component is web technology related, it should support all the target web browsers. In case the component is an application binary component it should have support for all the target operating systems, including desktop and mobile operating systems if they are or can become essential in the near future. The component includes less risk, if it has large cross-platform support or if it is fully platform independent. (Choudhary, Jalote 2014, p. 642-643.)

Summary of requirements introduced in this subsection

Req. 33	The solution should use 3rd party software components when there are good ones available, but the decision of using each component should be evaluated and though carefully.
Req. 34	Selected 3rd party components should be compatible to the maintenance strategy thought for the platform and not cause severe problems or offline time.

Section key points summarized



2.4 Chapter overview

In the previous sections in chapter 2 there is a list of key points summarized after each section. In addition, all the requirements presented in the previous subsections are listed after each subsection. This section provides an overview of the requirements and does not dig into any requirement in detail.

The chapter 2 presents 34 requirements in total that are set to be fulfilled by the remote installation platform implementation. The requirements are created based on the goals, expectations and good practises made for such multi-agent system that can execute remote installations in an automated manner. All the requirements listed in the chapter 2 can be divided into the following categories:

- Installation automation capabilities (6 *requirements*)
    - Requirements: 1, 4, 12, 16, 17, 30
- Remote installation capabilities (*5 requirements*)
    - Requirements: 2, 5, 13, 14, 19
- Platform robustness and efficiency (*11 requirements*)

    - Requirements: 6, 7, 9, 10, 22, 24, 25, 28, 29, 32, 33
- Platform extendibility and maintenance (*9 requirements*)
    - Requirements: 3, 8, 11, 15, 18, 20, 21, 23, 34
- Platform and data security (*3 requirements*)
    - Requirements: 26, 27, 31

Only less than one third of the requirements set to the remote installation platform design are directly linked to the primary functionality, remote installation and automation capabilities that the platform is expected to provide. More requirements are set to make the platform operate in a reliable and efficient way while serving its purpose. In automation platforms the value produced by the system is easily lost if the system cannot maintain the level of automation over a longer period of time or if the automation does not make the process more efficient so that the increased output can match the invested price over a reasonable amount of time.

More than one quarter of the requirements are linked to the extendibility and maintenance of the system. The driver in these requirements is to reduce the time, effort and cost of a long term operability. It is very likely that over time the platform will extend by the number of included agents and by the functionality requirements set to it. Therefore, having a platform that is easy and cost-effective to extend and maintenance makes it more sustainable. Having these properties also provides longer life expectancy for the platform, hence not becoming obsolete over changed operational requirements.

The security related topics were out-scoped from the focus of this research. Therefore, less than one tenth of the requirements are linked to security issues. The need for proper security measures is acknowledged, but the field is only surfaced by the requirements. It is clear that all the security issues require further research and proper solutions. The proposed design should provide easy extendibility to improve and fix all the security issues that it may and will be having in the future.

3 Research

This chapter presents an implementation of a remote installation platform that can be used to automate installations of one specific energy data management system platform. While it is designed for a specific purpose the architecture model and design goals can be applied or used as a reference to build other similar remote installation platforms as well.

This chapter is structured as the following. It begins by explaining the existing installation characteristics of the target EDMS platform environment. Then the chapter continues by modelling the installation process as units that can be used to define an installation process control flow for automation purposes. Presenting the installation design that can be automated is followed by addressing the most important problems in remote installation and installation automation subjects for this specific purpose. After going through the problems, the design proposition for the remote installation platform is presented, first from the overall perspective and then going into specifics of the different types of agents that the proposition includes. In the end, the chapter discusses the aspects of feedback generation of the system. The feedback, which is the data collected by the platform is considered to be a very important part of the system in order to gain a significant portion of the potential value the system has to offer.

3.1 Defining typical EDMS platform installation

In the context of this thesis, the “EDMS Platform” refers to an Energy Data Management System platform of an anonymous *Company X* that is a real modular energy data platform. The platform has approximately 300 running production installations. The installations are serving energy industry companies that operate mainly in European markets with varying regulations and industry standards.

The figure below (*Figure 10*) shows the EDMS platform’s server components in high level. The platform is based on multitier architecture where data storage, application server machines and clients are separated. The database is a single instance data storage that uses Oracle database engine. An Oracle database instance can run on Windows or on Linux operating system.

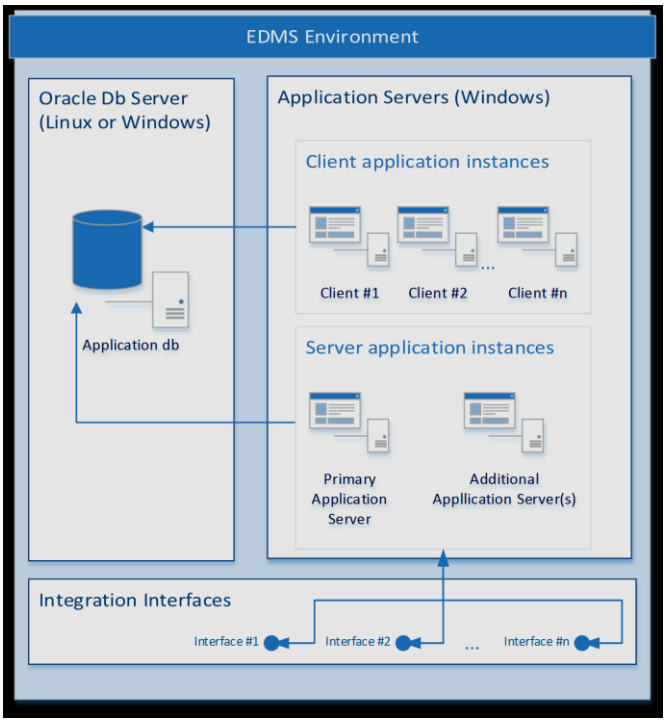


Figure 10. Reference EDMS platform high level application architecture.

There may be a single or multiple application servers that run daemons (*Windows Services*) on Windows operating system. These daemons run web interface listeners and some data processing activities that are executed based on various (event driven and periodic) types of triggers. Some of the activities are computing and or memory intensive operations, in which case the execution time is bound to the available resources.

The sufficient number of application servers and client machines varies based on the size of the energy company. The data managed by the system must be processed in faster or equal pace where all the regulated time constraints are met. The aggregated data magnitude that an energy company needs to manage in the energy market is mainly dictated by the number of metering points (electricity, gas and water utilities) that the company manages and the measurement frequency of each metering point.

The application servers host many types of interfaces for 3rd party software integration. Some of the interfaces operate on a file basis (shared directories) and some are web service based. Some interfaces have requirements for network policies, for example, network share access privileges and communication port openings. A typical energy company has multiple key systems that need to be connected to each other. The EDMS system is just one part of the whole puzzle. Therefore the EDMS platform has an integration layer that connects to other systems and data sources.

3.1.1 What is included into an installation

In the context of this thesis the term *installation* refers to the installation operation that may be a fresh installation of a new system or an upgrade installation that updates the existing software with the newer version. The EDMS system installation includes two main operations, *file installation* and *database installation*, which are present in some form in every installation case.

File installation

The *file installation* operation may contain some or all of the following tasks:

- backing up existing binaries to external directories or as compressed file archives,
- stopping daemons and processes that are part of the EDMS platform and therefore located in the binary directory,
- installing (copying) the new binaries to all application servers and optionally on all client workstations as well (as opposed to using separate Microsoft Windows Installer setup package for clients),
- configuring XML-formatted configuration files that are part of the application binaries,
- registering the Windows *Component Object Model* (COM) servers in each application server that are part of the application,
- starting the stopped daemons and processes.

The combination of binary and other files that are included in a single EDMS platform application instance can differ from each other. The combination depends on the target version and the included software licenses that are linked to a customer account. From the installation system planning perspective, the algorithms and logic that selects the proper combination of files to include per installation case is already built into an existing installation package creation software and there is no evident reason to change that.

The package creation software creates installation packages, which each contains a manifest file that determines the files to install. The installation files are packed with an installer software that can install them accordingly. Given the required input details and administrator privileges, the installer software can execute the file installation process on a local system, including all the tasks required. Therefore, from the remote installation platforms point of view, being able to provide the required input and local execution privileges for the installer software is all the remote installation platform is required to do in order to be able execute the file installation in any installation case. For more controlled execution, it also requires the ability to get real time feedback and being able to abort gracefully any ongoing activity.

In the existing installation procedure the XML configurations are updated manually whenever required. The problem in these configurations is that they contain some

environment or customer specific values that should not be overwritten by any installation. On the other hand, the new version may have some new values where the default value should be placed in the configuration in order to get the updated software to work correctly. Therefore, a fully automatic file installation operation should include some configuration merging algorithm that would leave the existing non-default values intact while adding the new changes to the files. Such operation should be fully automatic whenever possible, traceable from the change perspective and therefore undoable, and have close to none maintenance overhead from the software development perspective.

The *Component Object Model* (*COM*) server registration is a Windows operating system specific procedure that allows creating and sharing binary software components from one application process to another (Microsoft 2016). The COM-servers must be registered with the operating system where they are accessed by any COM-client application that utilizes the specific objects, which is the case of the target EDMS system is the same application servers. From the remote installation perspective, it is an important note that with a proper set of execution privileges, it is possible to execute the registration operation over the network.

Database installation

The *database installation* operation may contain some or all of the following tasks:

- stopping database daemons for the duration of backing up the database,
- backing up existing database to external directory or as a compressed file archive,
- starting the database daemons that were stopped before the backup operation,
- saving some module-specific option flag values to the database that has effects on the installation execution,
- registering the included application modules with a command line tool that will auto-generate some database tables and data rows,
- running database schema creation, alteration and data migration scripts with an intelligent SQL execution engine, that will generate proper SQL statements that transforms the database schema from the current state to a specific target state table by table and column by column,
- running a command line -based configuration tool that imports data objects and configurations from XML input files to the database as data rows and relations,
- running a smart diagnostic tool that will search the database for Foreign Key columns and generate any missing indexes for the referenced columns.

Similarly to the file installation operation, the existing installer software has ready functionality for all the database installation steps. The order of executed steps is designed for manual, user-centric use. To some extent, it can work with automation use, but for automation purposes it may use far from optimal execution ordering.

The figure below (*Figure 11*) visualizes an example versioning of the target EDMS platform software. In the fresh installation case, the database schema is created based on a database schema image of a release branch that is closest beneath the target version. Then the final target version is achieved by running patch release related modifications, version by version until the target is reached. In the upgrade installation

case, the database state is updated by going through each version, including all the releases and patch versions between the source and target versions until the target version is reached. The installation supports only going forward in the versions and not any kind of downgrading.

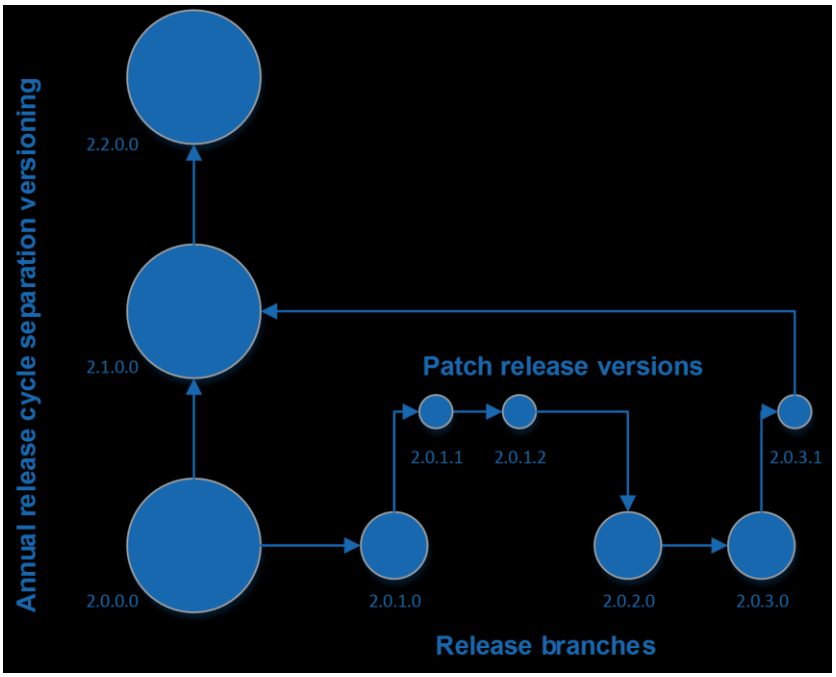


Figure 11. The reference EDMS platform software versioning.

The target EDMS platform has been developed to adapt and to support several market areas, market and customer specific requirements, laws and regulations. Therefore the database installation has many variations caused by the above dimensions that create complexities of the installation process. The complexities manifest in different types of data and schema dependencies per installation case. They can cause incompatibilities in database installation executions that frequently produces errors in many database installation cases of novel versions. Therefore, database installation errors are expected to occur frequently and reporting them quickly with enough accuracy, details and context information is an essential part of an installation software in this particular case.

A single EDMS platform system instance may hold information for several energy utilities (e.g. electricity, district heating, district cooling and gas) and for multiple energy companies (e.g. in a national data hub instance). In the reference EDMS platform, all the above data cases are handled fully in separated entities called *domains*. The *domains* that are created as separated schemas in the database are completely isolated and unaware of each other. For interoperability purposes, each domain may own one or several sub-schemas that are referred as *external schemas*. In the existing installation procedure all of the *domains* and their *external schemas* are installed sequentially, which causes the aggregated installation time to grow linearly based on the number of similar domain setups included in the installation. This will lead to a case

where a fresh installation of a national mid-sized data hub with 16 domains to take approximately one and a half days' time.

Other installation tasks

In addition to the main file and database installation operations, an installation may contain some other maintenance tasks as well. For example, if the installation is used to run some automated tests or to test the installation procedure itself, there can be a need to restore the environment into a specific state before proceeding into the actual installation. In addition, if creating backups before each new installation is part of the procedure, to keep up the free disk space, the installation can include some clean-up operations to remove old backups based on some predefined rules. Moreover, in the automation case, the installation package creation operation should be considered as part of the installation procedure. In brief, the installation operation includes some extra operations to keep the target environments operational and capable of executing future installations as well.

3.1.2 Installation as a set of tasks and sub-tasks

For simplifying parts of the installation operation and for supporting the modularity of an EDMS platform installation in environments that are used for different purposes, like automated testing, manual testing, development and production, this thesis describe the installation operation as *tasks* and *sub-tasks*. These *tasks* and *sub-tasks* could be interpreted as the description language that can describe a full installation procedure. Each of the *tasks* and *sub-tasks* represent an *execution block* in the installation process control flow.

Installation plan and its sub-item hierarchies

One installation process is described as an *installation plan* that contains the *tasks* and *sub-tasks* that are specifically intended to be included into an installation. In theory, any agent that is part of the remote installation platform should be capable of translating an *installation plan* into a sequence of actions that will transform the target environment into the intended end state.

Installation plan can contain one or more *installation tasks* and an *installation task* can contain one or more *installation sub-tasks* (*Figure 12*). These three concepts are modelled as data structures and they are referred as *plan objects*. Each *plan object* can be serialized and deserialized into and from a form that can be transferred through HTTP protocol using RESTful application programming interfaces. In the reference implementation, the format selected for serialization is JavaScript Object Notation (*JSON*) as it is compact and currently most used serialization format in modern web applications. In addition, there are many efficient JSON parser implementations available for free for all the most common programming languages.

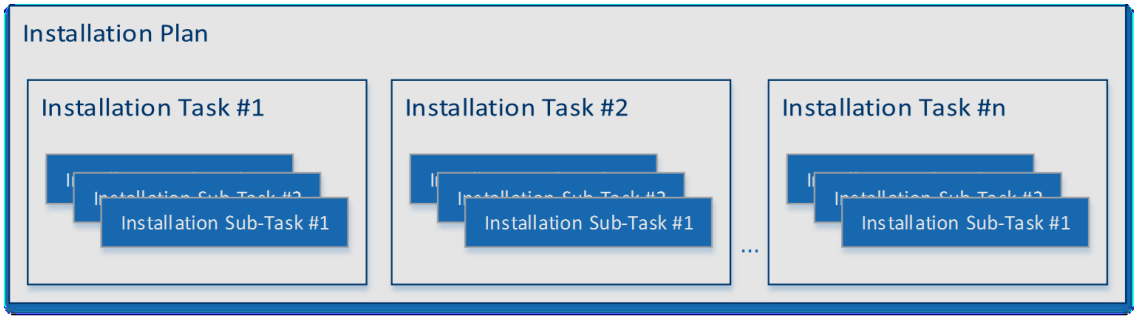


Figure 12. Data structures that can describe a single installation process.

The installation is divided into the following *tasks* that can be selected as part of the installation:

- environment restoration,
- installation package creation,
- file installation,
- database backup,
- database installation,
- after-installation operations.

There are some dependencies between the tasks. If the tasks would be executed only sequentially the direct order that the above list is presented in would be the one that satisfies all the dependency requirements, i.e. what operation should have been executed before a specific *task* can be executed without having unwanted effects on the outcome. To avoid unnecessary complexity, the smallest size of a *plan object* that should be given to an agent to execute is a *task*. That provides an agent implementation the freedom to decide the best approach to execute the set of included *sub-tasks* based on the requirements, conditions and available resources. In addition, it reduces the complexity of required execution synchronization and the number of dependency issues since the number of items requiring synchronization across agents is kept smaller.

The base for above division in installation operation is driven by dependency thinking, the party or a role who should execute the task and the combination of actions that may be left out from some installations. For example, environment restoration that includes restoring a database or binary directory to a certain state is not part of most of the installations since in general, versions are intended to go only forward. The package creation in upgrade installation case depends on the current version of the environment's software installation. Therefore, it cannot be executed before the environment is in a state where the actual installation start version can be resolved. In addition, the package creation should occur close to the data source rather than close to the installation target environment. Having separated file and database installation tasks comes naturally from the fact that they are the two primary operations of any EDMS installations in this particular case.

The file installation operation includes a *sub-task* for backing up files before installing the new files. In database installation case the backup operation and installation operation are in separated tasks. In the latter case, there are two main reasons for

such separation. In file installation case, the backup operation and file installation operation both have the same pre-requirement that the daemons are stopped before the operation can complete. In the database installation case the daemons must be stopped for backing up, but they must run in order to execute the database installation operation. Therefore the database backup includes several steps to be added to the installation operation where the file backup operation can cope with just one. Since both of the backup operations are optional and not always included in an installation, for control flow management reasons, there should be a single *plan object* controlling each of backup operation's inclusion state.

Common characteristics of the plan objects

All *plan objects* are data structures that hold input parameters and data that is added during and after the execution. Therefore an executing agent can receive a plan object from any level (*installation plan*, *task* or *sub-task*), execute the proper set of actions based on the input, and add some details and results from the execution. After the execution the agent can then return the same structure back to the initiating agent as an execution report. By using the same data structure, the results and details are bound to the correct context, a specific plan object execution, without significant extra effort. In addition, the data can be transferred using the exact same technologies and optimization logics to both directions. Moreover, when the *plan object* parsing and executing logics are separated into their own libraries, the software code can be reused as is in all the relevant software components that are working as part of the remote installation platform.

All *plan objects* have some common properties that are defined in each object. The common properties include *Identifier*, *Runtime Identifier*, *Enabled* flag, *Executed* flag, *Succeed* flag, *Execution time* and three arrays of messages (*Information*, *Warnings* and *Errors*). The *Identifier* is unique per plan object type and it is used in data structure parsing to detect the type of a plan object. *Runtime identifier* is unique per plan object instance and it is used to connect the data structure that was sent as execution input to the returned data structure that has the execution results filled. It is possible to have multiple instances of same plan object type in a single installation and therefore type-based identification is not sufficient. The e*nabled* flag helps creating temporary modifications to the execution flow and the other flags are there for reporting purposes. The three message containers hold the execution details that are meant for helping human users to have an understanding about the overall installation process that was executed. They separate the message into semantic categories to help produce more efficient reporting methods.

The container *plan objects* (*installation plan*, *tasks*) hold the sub items (*tasks, sub-task*) and their aggregated results and *enabled* states but no actual input for the execution actions. Executing a container object is done by executing each of its sub items. The implementation is required to include a way of determining the correct order of execution for each sub item. The *sub-tasks* are the actual units of execution that each represents a request to execute sequences of actions to get the environment to an intended end state. They hold all the required input parameters that are needed to execute the specific actions. In this context, an action is a simple task, for example, copying a file, stopping a daemon, checking some condition, etc. Since the *sub-tasks* can differ from each other significantly, the input parameters per *sub-task* are also

different and classified as *sub-task specific parameters*. Many of the *sub-tasks* may depend on input information that is constant per installation or per environment. Therefore, some of the parameter values are classified as *shared parameters* and they are shared between *sub-tasks*, also over *task* boundaries. Examples of such input are source installation package location in the current environment, target data source and environment-specific user credentials for accessing resources.

Sub-task classification and dependencies

The sub-tasks are classified into four different types: *validation*, *before-action*, *action* and *after-action* based on the nature and execution moment related to the operations that the task is included in. The classification is visible in the sub-task *identifier*. Validation sub-tasks are auto-generated by the executing agent based on the included *action sub-tasks.* They are the basic condition check -type of operations that check for conditions that, if not met, will fail the task with certainty. The validations are meant to provide quick feedback of certain failure conditions with minimal effort required to restore (roll back) the environment back to an operational state. A typical validation checks input data correctness, tests credentials or data source connection, or required disk space availability. *Before-actions* prepare the environment to be ready for the actual operation. Example of a *before-action* is a sub-task that stops or starts some daemon service. *Actions* are sub-tasks that execute the target operation, for example, file copy operation for each target, a list of database script executions run with intelligent code-generation engine or a configuration set import operation. *After-actions* are the post-procedures that restore the environment into an operational state. Example *after-action* is a sub-task that starts daemon services.

While some of the *tasks* have dependencies from other *tasks*, also the *sub-tasks* have dependencies. *Figure 13* describes *sub-task* dependencies inside the two most important and time consuming tasks in the typical EDMS installation. The dependencies were deduced by investigating the design principles and architecture of the specific EDMS platform solution of the company *X*. The two *tasks* themselves are designed to run without having any dependencies to each other apart from the overall end result of installation successfulness. The figure shows with arrows the dependencies where the outcome from arrow target must occur before starting the execution of arrow source. The arrows that are drawn with dotted lines are not real dependencies, but are added to help optimizing the process execution.

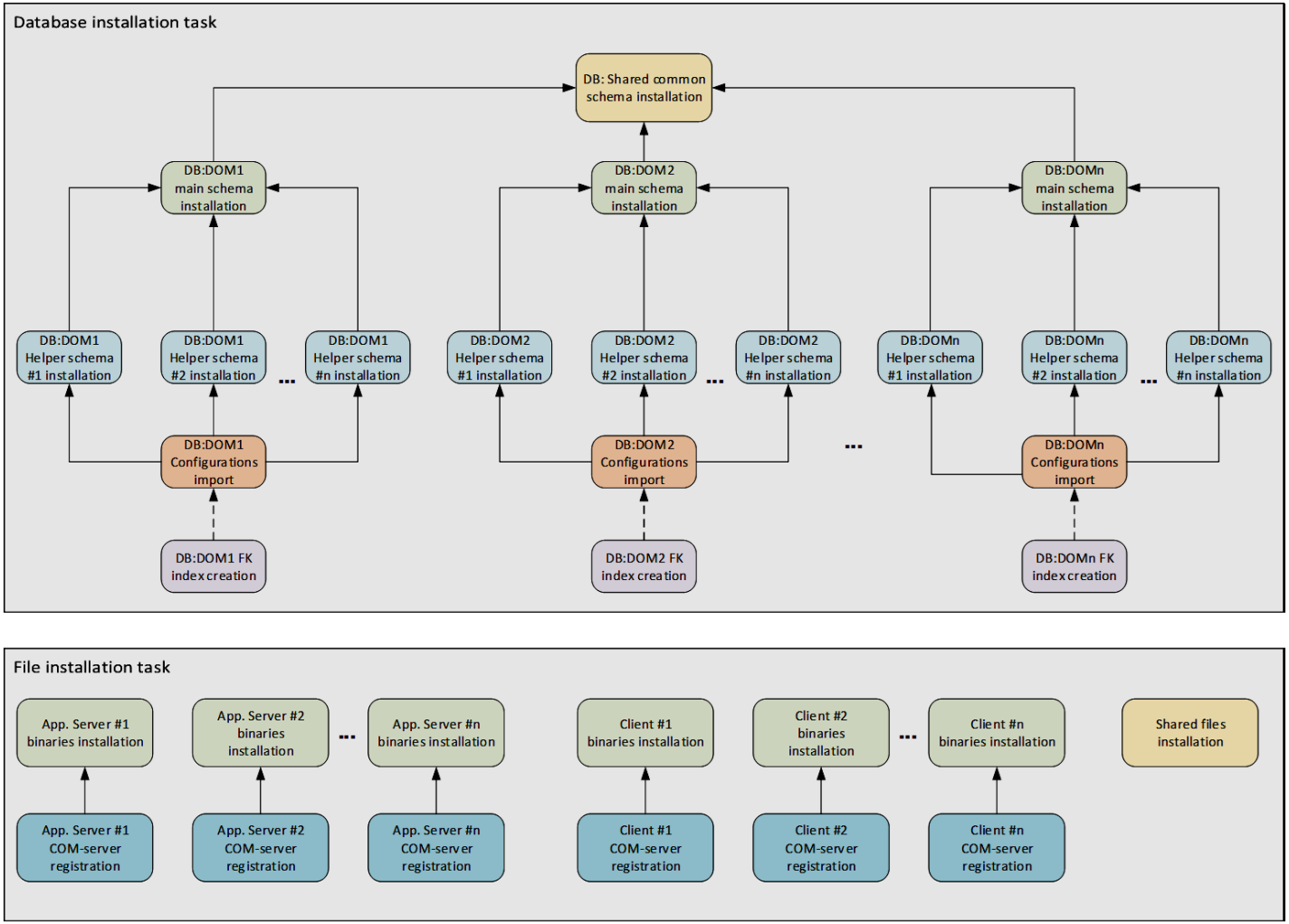


Figure 13. Installation task dependency diagram extracted by inspecting several installations and the installer software source code.

The dependency diagram helps understanding what needs to happen before a specific operation can be executed so that the outcome is as it is expected to be. For example, in the database installation task (*Figure 13*) the installation software is constantly analysing the database state and altering its operations based on the results. Therefore the execution is dynamic rather than static. In addition, to succeed, most of the database-altering tasks depend on the precondition that the preceding items have been executed successfully because the underlying database is relational and it has relations (dependencies) between data objects.

Each of the *tasks* has many different types of sub-tasks that they may include. The complete set of sub-tasks included in each task are listed in the appendix "*List of installation plan tasks and sub-tasks*" (*Appendix 1*). An example of an *installation plan* input data structure is visible in the appendix "*Example installation plan in serialized format*" (*Appendix 2*).

3.1.3 Environment specific installation information

An *installation plan* data structure discussed in the previous subsection is designed to describe an installation procedure for a single customer. In reality, for single customer there may be several different dedicated EDMS system environments, including for example, production environment, customer's *Software Acceptance Testing* (SAT) environment and software vendor's *Factory Acceptance Testing* (FAT) environment. Even though in theory, these environments should be clones of each other, it is not usually the case in the real world. Reasons for differences are, for example, that the production environment usually has a lot more load and therefore more resources and performance. In addition, the testing environments are updated to the next software version a lot earlier than the actual production environment.

To be able to reuse the same *installation plan* for multiple environments that are maintained for the same customer, there are list of installation parameters that should be possible to define per environment. Moreover, because the environments may have a different usage purposes, there should be possible to define some boundaries and reporting options for the differentiating installation procedures. Ideally, the configurations that define installation procedures for a single environment should not need changing after the first time *fresh installation* turns into a continuous upgrade procedure.

To meet the goal of the *installation plan* reusability, the environment specific installation parameters are defined in a data structure called *installation subscription*. *Installation subscription* subscribes an environment to apply installations based on a specific *installation plan* in a continuous fashion. The *installation subscription* based installation can be triggered periodically, manually or remotely whenever a new version is available and wanted to be installed to the target environment.

The installation subscription has the fields listed in the table below (*Table 1*). The *Installation plan UUID (universally unique identifier)* value can be used to link

subscription to a specific *installation plan*. If the *installation plan name* field has value, but no *uuid* is defined, any locally stored enabled *installation plan* with a matching name is selected when executing subscription based installation. Therefore, it is possible to have several complementary plans configured for customer environment that can be switched quickly, which can be useful when having an environment for testing new development.

Table 1. Installation subscription fields.

Field	Description
Installation plan name	Human-comprehensive name of an installation plan that is linked to the subscription.
Installation plan UUID	Unique identifier of an installation plan that is linked to the subscription.
Branch name	Name of the version branch from where the fresh installation should start.
Update over release branch version flag	Flag that permits updating to next release branches.
Data source	Connection information (connection string) for the target db.
Maximum version	Upper boundary version where the subscription allows installation.
Customer package name	Name of the customer license package that defines installable components.
Installation scopes	Array of categories that are included from customer license package (e.g. “Default”, “Test”, ProdEnv”).
Email report recipients	Array of email addresses that receives report after each installation.
Slack report recipient web hooks	Slack application integration web hooks that reports installations to slack channels.
Enabled flag	Flag that can be used to temporarily disable the subscription and prevent it from being used.

Branch name field defines the release branch from which the latest available version is installed when the subscription based installation is executed for the first time, if no EDMS software instance exists before. For the next upgrade installations, the *update over release branch version flag* controls whether only greater version from the same release branch are accepted to be installed or if the instance is allowed to be upgraded to next release branches as well. *Maximum version* defines the upper boundary for the version that is accepted. That value can be linked to a customer license information in a way that it is automatically updated when the specific customer is allowed to get future releases. *Customer package name* and *installation scopes* are also customer installation package content related information that is linked to data managed by a system called *License Manager. License Manager* holds customer account licenses and their contents from the installation component scope perspective. Subscription also has fields that define the recipients of automatic installation reporting and flag to enable or temporarily disable the subscription. Reporting is covered with more details in section 3.6.

The subscription does not include information about the local environment paths. The paths are tried to be kept as standardized among the environments and the subscription design tries to force configuration creators towards that goal. It is possible that in the future, it is required to add at least the root drive label to be configured in the subscription. In addition, the user credentials required by the installation are likely to be environment-specific. In the reference implementation of this thesis, they are included in the *installation plan*. The long term plan should be to allow the customer to insert the required passwords locally in the target environment which would then be encrypted and linked to the subscription in a way that the local installation agent only can decrypt the password when they are needed and the credentials data would never leave the target system. Implementing such feature in a secure way was left as a potential topic for future research.

The subscription execution logic takes care of applying the subscription based values to the linked installation plan when the execution is passed to the executing agent. The new version availability is checked from installation package source API with specifying the boundaries defined in subscription and extended with the information about the current instance version. If a new installation package is available, the installation continues, otherwise subscription execution ends. An example of an *installation subscription* data structure is visible in the appendix "*Example installation subscription in serialized format*" (*Appendix 3*).

3.2 Installation related problems

When changing the installation process from existing, manually executed wizard-like installer execution that runs as an application process in the target environment into a multi-agent based remote installation platform that handles automated installations over network, it is expected to have new problems to solve before the platform can be counted as operational. Section 3.1 discusses about how to transform the installation process into a format that can be used to define target-specific, full installation procedure which is able to be transferred through multi-agent network. This section addresses the problems that are new to the remote installation platform compared to the old manual installation process. In addition, it addresses the automation related problems that are relevant for the specific installation problem discussed in the Section 3.1.

3.2.1 Addressing problems related to remote installation

The remote installation in the context of this thesis means that the actuator (a person or a computer software) that initiates the installation is not directly connected or present at the target environment (computer) where the installation process is executed. The connection is indirect, which may mean connection through a network interface or a

mobile agent that is configured by the actuator, then transferred to the target system by installation media (e.g. in a memory stick) to be executed and then returned to the actuator party after the execution is completed.

Since the Section 3.1 already covers how to be able to configure an installation process that does not require further input to launch and execute an installation, the first new problem to be addressed is how to deploy the software package and installation configurations to the target environment. The typical solution for an installation engineer is to use a compressed archive that contains the data, which can be copied through various channels to the target environment. The same method works well also for the remote installation system, which can easily transfer the archive file over the network with reliable transfer protocols (*TCP/HTTP*) or by using installation media such as the memory stick. Since there is a chance that the data gets tampered by broken hardware or unexpected 3rd party, the process should include a data integrity check mechanism that uses for example a hash-based checksums (*CRC*, *SHA-1*) to verify that the content has been transferred as unchanged to the target site. The selected hash algorithm has little effect to the verification process and can be changed later in case it is found to be not sufficient for the purpose. In addition, the transferred data should not contain any sensitive data, for example, customer-specific information, user credentials or target environment details in unsecure (i.e. non-encrypted) format that could have harmful effects if leaked to the public.

In the remote installation case, the controlling party does not have the same kind of visibility about what is currently happening in the target system as in the previous installation case where the user had wizard-like user interface that displayed installation progress and visualized feedback for each user's actions. The remote installations have lots of variables that have an impact on the total execution time. Therefore, it is not usually clear how long an installation process is expected to last. The remote environment needs to provide acknowledgements for the execution requests and status and progress reports to the executing party while the execution is expected to be ongoing. Therefore, there should be bidirectional communication channel open during the installation that can be used to transfer the messages between the parties of the remote installation. The communication capabilities should include the possibility to abort the ongoing operation since human users tend to make errors that they want to be able to cancel efficiently. Since the connection may be required to be open for hours and the exact timing of the passed messages is not known in advance, the communication channel should be implemented with technology that has low overhead for long-term open connections that are used infrequently. Currently the commonly recommended technology for such connection usage pattern is *WebSocket* protocol defined by Internet Engineering Task Force in RFC 6455 specification *(Internet Engineering Task Force 2011)*.

The remote installation target party (agent) should never trust that the provided input is compatible and correct for that specific environment. Therefore, all the input data should be validated and compared to the current application instance state in the receiving end. The validations should be executed by comparing what does the input expect from the environment and the application instance, and what is the actual current state. For example, if the input expects that the local filesystem has a drive with the letter D (Windows-specific drive specification type), the local filesystem should really have such drive. In addition, if the installation input states that the installation

should update the application from version “X” to version “Y” and the current version is “X-3”, the validation should detect such version mismatch and prevent installation that would fail or cause unexpected results.

The initiating party in remote installation may not have the full access to the target system. Therefore, the remote installation agent should create sufficient logging and trace information about all the important steps and any occurring problems that causes the operation to fail. The data is required in order to have possibility and effective means to get a good understanding about the problem and executed steps to be able to investigate the problem without direct access to the system. In real life it may not be always possible to be fully independent of the target system and never require connecting to the target system for fetching more information about the problem. That is due to the fact that many of the unexpected problems are caused by some external conditions that are present only in the specific environment. Therefore, they can be fully unknown by the remote installation agent.

To prevent the unnecessary data transmission, the output produced from an installation operation should be collected per installation session basis. When all the logs and reports are created per installation, it is easy to collect, report and archive the installation output per installation basis. Based on the general feedback from target Company X’s installation engineers’, the installation scope of logging data is usually the exact scope of interest when someone investigates problems occurred during a failed installation.

In a packet-connected network the connection between the communication parties is not usually physical or direct. Therefore, there can be both temporary and permanent connection periods of unavailability between the installation initiator and the remote installation agent. The communication logic in the sending party should have an implementation for buffering and re-attempting sending the data in order to provide higher tolerances of temporary unavailability. In addition, the failed attempts to communicate should be logged in a way that it can be inspected later to understand why the operation failed. Moreover, the implementation should either not rely on any specific order of the messages being delivered from the sender to the receiver or it should guarantee the order, which can be a complex and costly feature (performance-wise) in a network environment.

The continuous remote installation procedure that is done by permanent remote installation agents has a dependency to the agent software. As do every software, the agent software itself requires also maintenance, updates and bug fixes. Since the remote agent should stay operational in long term use, it should be capable to update itself with available software updates. The updating operation may mean limited functionality or offline time, which should be planned and included in the installation platform design. In addition, the system should be capable to operate in a condition where not all remote installation agents are running the same version of the agent software. The software change management solution should either include a full backward compatibility or the communication messaging should include means to detect the outdated agent version and be able to request an agent to update itself before continuing the operation. The latter option is easier to design since it is very hard to predict what kind of changes are needed to be done for the software in the future to cover all the requirements.

3.2.2 Addressing problems related to automating installation tasks

An important factor in any fully automatic operation is that there is no user, a decision maker, available during the operation. Otherwise the operation would not be considered as automatic. Not having a user does not mean that the ongoing procedure could or should not require problem resolutions. To solve them, it is required that the automation logic have means to resolve and provide the resolutions autonomously. The problem solving capabilities can be provided by either using some predefined resolutions for each specific case included in the installation procedure input data or by having some algorithm that can come up with the resolution based on any input it is given. If the latter approach is used, the implementation should consider:

- How to know the safest option to select from the available alternatives?
- What are the worst case scenarios for each option?
- Is the worst case of an option that continues the execution more acceptable outcome compared to abort or stopping the procedure?
- Is there a probability that is acceptable to lead to an unwanted end scenario?

If the resolution to a problem is predefined in the input data, the decision and responsibility to weight the alternative outcomes is left to the user who configures the automated installation procedure. When designing an algorithm that can provide a resolution to any unexpected problems, the most intuitive solution could be to always abort when any problem occurs. In a real world use failing on every error can be very ineffective. To reveal all the problems, it would require as many iterations as there are problems. For example, in the past EDMS platform installation, there have been frequent and the number of problems in database installations when varying SQL scripts have been executed to different systems with existing data. The experience has shown that in most cases, it is a lot more effective to let the installation process continue to the end also in the case when errors occur. After getting the full list of errors from the installation, the software developers and installation engineers resolve which problems were caused by the previously occurred errors and which were caused by other unrelated problems. In that way a single installation can reveal all of the problems and there is significantly higher chance that the next installation will succeed when all problems are addressed at once. The example does not suggest that all errors should be handled in the similar manner, but that there should be more intelligent approach to handle the problems based on some type or case qualification.

One of the most basic requirements of an automated installation process is to have the ability to decide what to install. The decision requires collaboration from the remote installation agent and from the central facilitator agent. The remote installation agent has the most accurate knowledge about the target system and the current state of the application instance that is targeted for installation. For example, if there is no existing instance, the upgrade case should be ruled out. On the other hand, if there is existing instance running and the installation process works in the way that the current version is rather migrated to the new version instead of replacing it entirely, the current version

information is mandatory for making the decision of the new installation package version range.

Being a centralized source for information that is gathered from other back end systems, the facilitator agent has the best knowledge about what versions are available and which software components are meant to be installed into a specific customer environment based on the agreements made with the customer. From the software vendor's management perspective, it is better to have the control of installations in the software vendor's secure and protected environment that has limited outside access instead of distributing it into all of the remote agents that are part of the platform. Therefore the remote installation agent should be an information provider and have such capabilities, but the decision making should be done by the centralized facilitator. This approach does not remove from the remote installation agent the need to validate and confirm the decision made by the central facilitator in the context of the target environment. In addition, the remote installation agent should be responsible to decide when the environment is ready for the installation.

The automated installation requires some level of isolation that protects from any external interference while executing an installation operation. The remote installation agent should have clear signals to indicate its busy state and also the busy state of the target system. During the busy state, no other installation or other interfering operation should be executed to the target system. While it may be easier to implement prevention system for interference caused by the installation platform and compatible software components, it is rather hard to prevent interference caused by human users and scanner software products (e.g. antivirus software) that is installed in the target system. These interference sources can possess exclusive locks on files and resources that prevents installation, to proceed or even to kill or alter processes that are controlled by the remote installation agent. The past installations have shown that during installation operations users have been running applications from installation target locations or changed the running states of daemon services that the installation depends on. In these cases the human user running manual installation could detect the problem and communicate with the responsible party to solve the problem. The same method is not viable to work with automation platform. There are methods to decrease the chances of having interference based problems, but it is realistic to expect that some installations will encounter such problems nevertheless.

In the real world, some of the automated installations will always fail when the target software is as complex and modular as is the reference EDMS platform. Therefore the automation scheme design should include a plan on what to do when the installation operation ends with failure. One option is to do nothing and risk leaving the target environment in a non-operational state until someone eventually fixes it. Another option is to do a full rollback and return the environment in the state where it was before the installation operation failed. That would naturally require to have the ability to roll back the original state. The latter option would leave the system in an operational state, but it may be difficult to investigate the problem in order to find the problem solution, if the environment having the state where the problem occurred does not exist anywhere. The third option is a combination of the two where the remote installation agent would first preserve a copy of the environment in the state where it failed and after that roll back to the original state. The third option has the highest cost of resources (disk space, IO operations) and time. Depending on the operational requirements and

environmental conditions, the remote installation software should apply some of the recovery plans and accept the cost it includes.

3.3 Remote installation platform architecture as multi-agent system

Based on the installation process described in section 3.1, the problems identified in section 3.2 and the requirements set for the design, this section proposes a multi-agent system architecture for remote installation platform. The proposition is done by adapting a matrix-agent framework presented by Zhang et al. in their research paper "*Matrix-agent framework: A virtual platform for multi-agents*" (Zhang 2006b) to the purpose addressed in this research. The adaptation presents the agent matrix role that is responsible to coordinate agents and their resources as a *central facilitator agent* (CFA) that is part of the multi-agent platform. The idea of having a facilitator role is also present in many other agent development frameworks and architectures, such as JADE (*Java Agent Development Framework*) and The Open Agent Architecture (Bellifemine, Poggi et al. 2001, p. 107-108).

The following subsections discuss about the remote installation platform design and architecture in high level. The purpose is to give an overview of the proposed system before going into the details of the different agent types that are part of the platform. A reference implementation is done based on the following design as a part of this research in order to provide proof that the proposed concepts work in practise and to reveal any unexpected problems that it may have.

3.3.1 Platform overview

To meet the goals and requirements set for the remote installation platform, the proposed multi-agent system has two types of agents. The first agent type is a *domain installation agent* (DIA) that is operating in the same *environment* (i.e. a computer running an operating system) where the target EDMS platform software is installed. The DIA has access to the local resources and is able to alter the target environment by installing or updating software applications.

The second agent type is a *central facilitator agent* (CFA) that has access to the centralized resources that are required to manage automated installations. The CFA is connected to the DIA*s*. It can send commands and provide information and input data to the DIA*s*. The DIA*s* are *intelligent agents*, which means that they are independent and in control of deciding when to execute the desired commands and what to do with the received input. Although, it is their desire to serve the platform and therefore execute any command as soon as it is possible without causing any problems to any existing operations.

The agents are connected to each other by using the network connections of their environment. In order to be able to communicate via messaging protocols, the network environment needs to allow the inter-agent messaging. That includes allowing the TCP connections from agent-to-agent in all the firewalls that are placed between the agents. The end-to-end connection between agents may also require setting up a Virtual Private Network (VPN) connections if some of the agents are operated in separated networks. The design assumes that there is a logical connection between agents and abstracts the details of how it is configured.

In the reference implementation created for this research there are one CFA and many DIA*s*. The DIA*s* are not directly connected to other DIA*s*. They are always connected to the remote installation platform by connecting to the CFA. In the future, the platform could be extended to have multiple CFAs to share the load coming from the DIA*s*, if the only CFA is becoming a bottleneck for the system scalability. That would require synchronization and cooperation mechanisms between the CFAs.

In the overall command hierarchy, the user has the highest authority, the CFA has the second highest authority and the DIA*s* have the lowest authority. In general, that means the user has the ability to override and, for example, abort the current operation in any DIA regardless when the command was given by a CFA. In addition, the CFA has the ability to send a command that can abort the existing operation in any DIA. As having the lowest authority in the system the DIA does not have any ability to cancel or control any ongoing operation done by a CFA. Any decision that would cause the CFA to alter its current processes should be made by the user or the CFA itself.

The figure below (*Figure 14*) shows an overview of the proposed architecture components placed in their operational contexts and the connections between them on a high level. It has three domains. The first one is the target environment where the EDMS platform software is running, the second is the backend systems and the third one represents the human users that are in control of administrating the process.

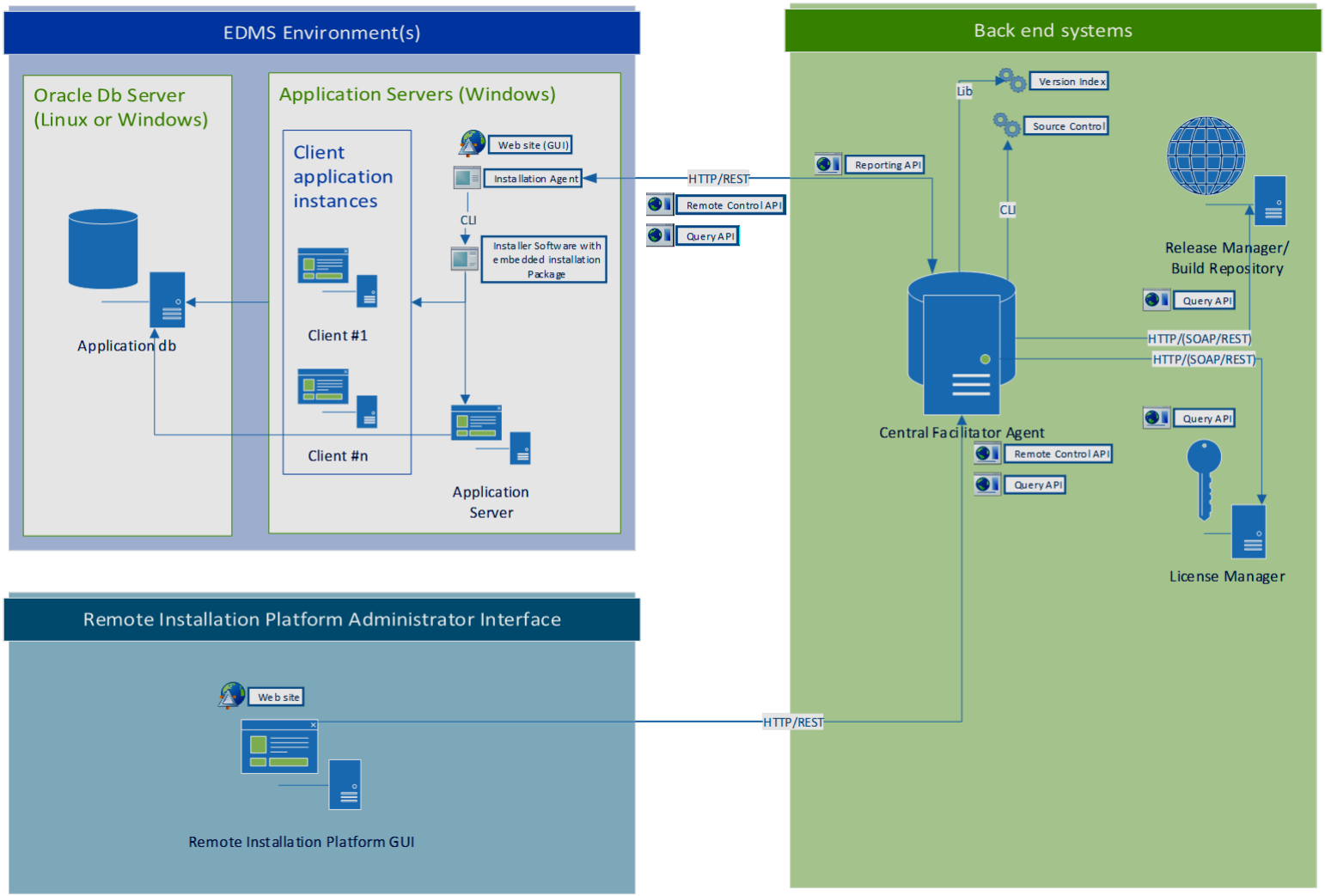


Figure 14. A component overview of the proposed system architecture.

EDMS environment

A DIA operates in one of the EDMS environments. It is connected to the CFA by using HTTP/REST API connections for the data exchange. It has a management user interface for human users and it is responsible for installations done to the target environment. For the installation operation, it uses a command line interface of an external installer tool that is embedded to each installation package delivered to the DIA per single installation. The command line interface is designed to be used by the remote installation platform and it supports remote installation platform's *installation plan* data format, graceful abortion by using a file-based signal and real time progress reporting towards the DIA through operating system's *standard output*.

An EDMS environment consist of a database server running on Linux or Windows operating system and a separate application server that runs daemon services and exposes interfaces to 3rd party services to connect to the EDMS system. In addition, the environment includes a number of client installations that connects to the EDMS system using the client software. The applications and client software are currently supporting only Windows operating system.

Back end systems

A CFA exposes various backend services to the remote installation platform in a controlled manner. It connects to other backend systems and combines their information to manage and administrate new and previous installations. It is the agent where any top level integrations to the remote installation platform should be implemented. For the DIA*s*, it exposes a set of REST APIs that can be used to query and report information.

While all the agents have some relational database where to persist the agent state and past information, a CFA requires a backing database engine that has enough performance and stability to support multiple parallel queries and data insertions. In addition, the database requires enough scalability to not be the first limiting factor for the number of agents that the platform can manage in the near future. For the reference implementation, a *PostgreSQL* relational database was selected since it has a long maturity, wide support for programming languages, references of having good performance in production use and it is free to be used in open source and commercial products. (The PostgreSQL Global Development Group 2016.)

A CFA connects to several existing backend systems by using their existing APIs and available connection methods. These backend systems include:

- *version index* that contains list of all the available EDMS platform system versions and their locations in a *Build Repository* located in a centralized file server,
- *source control system* that is centralized repository for source code files and documents that are part of installation source files,
- *Release Manager* and *Build Repository* that have source code based build outputs and metadata linking a specific build into a *release version*.

The release version is a term for a labelled build that has been given a version number (e.g. 1.2.3.4), which identifies that exact build and its place in the software version tree hierarchy. In addition, the backend systems include connection to the *License Manager* that holds information about customers and the platform software components that they have bought licenses to. To construct a single installation package for a specific customer, combined information from all of the above systems is required.

Administrative interface

The primary interface for human administrators to manage and control the remote installation platform is the administrator interface, a management GUI (Graphical User Interface) that connects to the CFA APIs through HTTP/REST protocols. The administrator interface is a *single page web application* (*SPA*) that works in a typical modern web browser. The application is accessible from anywhere in the typically closed network. It provides means to create, command, follow and inspect agents and the installation tasks that are part of the remote installation platform.

API design principles

All the REST APIs are designed to be asynchronous in a way that they don't block the executing thread. In addition, the implementations try to avoid any long running tasks inside a single request-response communication and use asynchronous *callbacks* instead. A c*allback* refers to a policy where the responder returns the response data by using a web service endpoint in the requester's end to pass the results when they are available.

In many multi-agent platform frameworks the communication layer that handles the inter-agent communication is separated from the API layer that allows communication with the non-agentized world, i.e. the other applications (Bellifemine, Poggi et al. 2001, p.107-108). In this model the same APIs can be used by all the parties since the same data is often required by agents as well as some other applications, for example, a user interface implementation. Having redundant methods to access the same data would add extra complexities to the system without adding any value to the platform.

Agent mobility levels

From the agent architecture perspective, DIA*s'* mobility level is high. Since they need to only maintain connection to the CFA*,* they can be easily moved to a new host if required as long as the stored connection information for the CFA that the agent currently has works from the new host as well. This is an important feature to have from the platform maintenance point, since a host can become non-reliable from the hardware point of view when it becomes old. The CFA's mobility level is not very high since all the other agents should be able to know its location at any times. The mobility can be increased by using alias network addresses that can be updated to point to the CFA's location nevertheless what is the actual physical location where it is running.

3.3.2 Agent lifecycle management

A multi-agent system architecture may have agents that live only for a short period of time until they have served their purpose. In the proposed remote installation platform, whenever it is possible, all agents are meant to be static and integrated as part of the environment where they operate. The agents are installed as daemon services that starts automatically when the system starts and runs continuously if not stopped on purpose by human administrator. Having agents as static help the goal of reusability. The same agent can run multiple installations and keep a track record of all the past installations. In addition, the overhead of creating an agent is minimized when the task needs to be done only once per environment. Subsection 3.4.3 discusses more of the case when an agent cannot be static.

The figure below (*Figure 15*) shows the stages of a DIA lifecycle. It starts with the *agent creation* where the agent is installed and integrated into the target environment. *Agent integration* stage is the stage when the agent sends a notification message to a CFA that it exists, which establishes the connection between the DIA and a CFA. Based on the notification it gets accepted as part of the platform. At *agent maturity* stage, the agent collects information from the platform and is configured to serve its purpose as part of the platform. At this point the agent becomes ready to be utilized in task execution. Running an installation is the common case of task execution and occurs on *agent maturity* stage. When any task execution starts or stops, the DIA reports the change in its state to the CFA so that the CFA knows when the activity is ongoing.

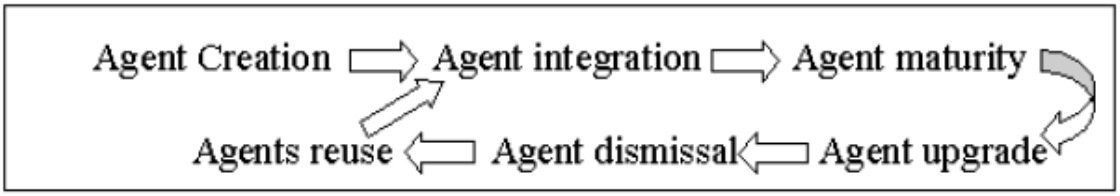


Figure 15. DIA lifecycle pattern adapted from matrix-agent framework model (Zhang 2006b, p. 440).

At the *agent upgrade* stage the DIA updates the knowledge it has collected from the CFA and the target environment to represent the current, possibly changed state. The agent then reports the CFA about any changed state. *Agent dismissal* stage occurs after the installation is done and the agent has updated its knowledge. It's the dormant stage where the agent does not have any ongoing activity. In the proposed platform, the agent maintains a periodic heartbeat that is reported to the CFA in order to the CFA to know which agents still lives and are responsive. In addition, while being in dormant stage, DIA checks periodically for new software updates to its own software and can update itself when one becomes available. *Agent reuse* stage occurs when the DIA is given a new task to execute. It becomes then active again and starts updating its knowledge about the current state and therefore cycling back to the *agent integration* stage.

The CFA has very similar lifecycle. After it is created or restarted, it connects to the platform, and builds up its knowledge about the environment and backend systems for any not already handled events. After becoming mature, it keeps track of the changes that occur in the environment. Therefore, in its dormant (dismissal) state, it needs to keep monitoring the environmental changes and be ready to receive any new reports from the other agents.

3.3.3 Creating a new installation procedure

One of the most common tasks that the installation platform administrator is required to do is create a new installation procedure that installs the EDMS platform to a new environment. If the procedure is hard and time consuming, it cuts the cost-effectiveness and productiveness factors of the system and therefore the value that the automated platform should produce compared to the manual installation procedure that has been the used method to complete installations. In addition, the people are not commonly very eager to change their work methods if the new procedure has a high learning curve. Creating a new installation procedure is one of the first tasks that they are required to learn while they move to use the new proposed solution. Therefore, it is the feature that will produce many of the first impressions towards the new technology. The past experience has shown that people's reactions toward new working methods and tools in *Company X* has been less enthusiastic since they always feel that they are in such hurry that they don't have time to learn other ways of working.

Having a good understanding of the feature or features that will be providing the first impressions can also be a valuable opportunity. If these specific features are designed and implemented with care to be quick and easy to use and to not require any technical expertise to do, the good impression will help the users to have a positive attitude toward learning also the more complex parts of the system in order to get all the benefits out of the new technology. Moreover, if the work hours required to complete the most frequently used tasks can be reduced to minimal that will have the greatest impact on the monetary value that the new investment will produce.

The remote installation platform design will provide a single modern and user friendly user interface for users to create and modify the *installation plans* and *subscriptions* that are the data structures, which affects to the installation processes. The design supports creating templates for most common installation procedures that can be easily selected as a starting point when creating installation procedures for new environments. These installation management tasks are placed into the web application that is running in the CFA. This research does not focus the usability and user friendliness factors of a user interface, but states only that they matter and should be designed carefully.

3.3.4 Environment based dedication and resource pooling

For customer environments, it makes sense to dedicate one DIA per environment since that makes it easier to separate and manage customers' environments. Dedicating an agent to a single environment causes the agent to have a very low utilization rate. That may be acceptable in a hardware environment that has been primarily acquired to do something completely different, i.e. serving as an application server for the EDMS platform, and having the DIA running only as a side product. On the other hand, if the hardware has been dedicated to run primarily the installations, the low utilization can be costly and wasting good resources.

Hardware that is dedicated to run installations is acquired for example for the *continuous integration* (*CI*) usage. In *continuous integration* the installation process and the *installability* of the target product is tested by running the installation using different customer packages. The goal is to see, if any errors occur in the installation after the latest changes made into the software source code or configurations are included. The successfully installed instance can be then used to run some automated tests to test the product and its functionality before removing the installation.

When the installations are not made to be persistent or to be utilized on any long term use, it is not that important to keep the history and trend of all the executed installations in one place as it is to provide quick feedback and clear reports when the installation or the product itself is not working properly. Therefore, it is better to have a group of non-dedicated DIA*s* that can execute an installation with any input in order to run any queued installation tasks whenever the agents are idle. That leads to a higher utilization rate per agent and hardware, and quicker beginning for the new installations since the task execution is not depending on a single agent availability. In addition, a higher utilization rate for a hardware leads requiring less hardware in order to produce the same value, which increases the cost-effectiveness.

In the reference implementation, an agent can belong to several resource pools. The agent advertises the participation to a pool by advertising its capabilities. The agent can be configured with a set of *tags* where the tag can be a resource pool name or a capability to execute a task. For example an agent that is part of a *CI* system and can execute any fresh installations has tags "*CI*" and "*FreshInstall*". The tags are reported to the CFAs in the periodic heartbeat reports. In that way the CFAs have an up-to-date list of DIA*s* and their special capabilities. In addition, each installation task that is configured to be run as part of the *CI* has a list of tags that it requires. In that way when a new installation task is available to be executed in a specific pool, the CFA can search the list for matching candidates that are free to run the installation task.

The DIA*s* that are dedicated to an environment are not required to have any tags that links them to a specific customer. They register their *installation subscriptions* to the CFA. Based on the subscription a CFA knows to which agents it should deploy any new versions that becomes available. Using the *installation subscription* provides more control and configurability over the installation for the DIA*s* that are managing installations in a continuous manner. These properties are required by the real world environments.

Having an agent pool is easier to implement for fresh installations. Fresh installation cases do not require any existing application instance that is in a specific state. In fresh installation case any agent that is part of a pool requires only an empty database instance that can be used in any fresh installation.

Depending on the included scope, a single installation process may last from minutes to even days. From an installation point of view, the upgrade installation case is more complex than a fresh installation. There must be an existing application instance available running a specific version of the EDMS platform software before the upgrade can start. Moreover, if a new version becomes available when previous upgrade installation is still ongoing or not succeeded, the upgrade task cannot start before the previous installation has ended successfully, which may lead to longer feedback loops. The complexities in upgrade case can be solved by managing repositories for different application instance versions that can be restored to any agent's environment. However, it consumes significantly more disk space resources and produces high disk I/O and network traffic, since a single application instance size is few hundred megabytes at a minimum, also in a compressed state.

3.4 Designing the domain installation agent that completes the installation tasks

3.4.1 DIA design principles

The *domain installation agent* (DIA) is the agent type that handles installation process execution into a target environment according to the instructions and installation input provided by a *central facilitator agent (*CFA*)*. The DIA can either advertise its installation capabilities to the CFA*s* or it can subscribe to be interested in specific versions of the EDMS software, and let the CFA notify the DIA when a new version is available. After a new version is made available for the DIA*,* it decides when it is a good time in the target environment to run the installation. While the installation is ongoing, the agent keeps monitoring the process and produce progress feedback and reports about the ongoing installation process to the CFAs.

In the reference implementation, only a single dedicated DIA manages all the installations for one environment. If multiple DIA*s* would be used, there needs to be a process to synchronize the agent's operations in a way that a single installation process can operate in isolated fashion, and no other agents are installing anything to the target environment while the installation is ongoing. Otherwise the target environment could end up in an undesired end state that does not represent any version of the product and therefore is not operable by any standards. Although, it is not intended in the current proposition, it is possible to have multiple agents utilizing the same data source, since all the data sent to a CFA is linked to a unique data source id that identifies the origin of the data. Therefore the CFA can detect the same data sent by multiple different DIA*s*.

The proposition represented in this research does not include automated installation of a new DIA instance into a new target environment. It requires manual work to install it as a daemon service, and to provide it administrative user credentials that allows altering the target system. The reference model is designed to run on Windows operating system and it utilizes a technology that allows it to run as console application or to install it as a daemon service with a single command. When the DIA is starting, it runs self-diagnostics about the ability to operate in the target system and initialization procedures to create or update its own data source to represent the target agent version. The following subsections discuss the DIA design in more details.

3.4.2 DIA component overview

The *domain installation agent* (DIA) design has multiple component modules that operate autonomously from each other (*Figure 16*). Each module has public interfaces that the other modules can use to consume services from the module. The logic separation has been done in a way that a module could be replaced with another module that has the exact same interfaces without requiring to change anything in the other components, hence following the service-oriented architecture.

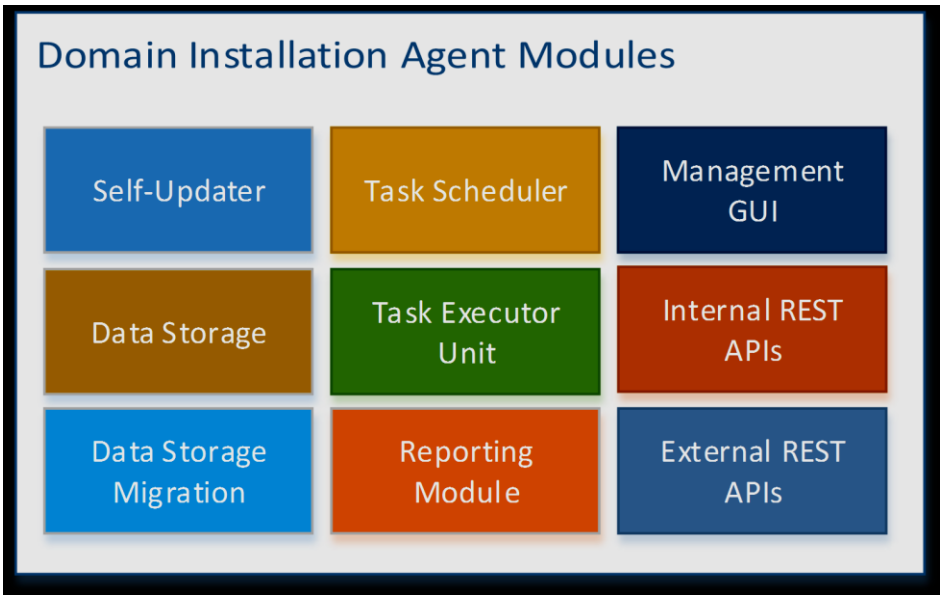


Figure 16. DIA component modules.

Self-Updater
Self-Updater module manages updating, rolling back non-working versions and tracking of changes of the agent software. It is the auto-update system (AUS) that runs in a separate daemon process that can stop and start the agent daemon for the update and rollback operations. It has no knowledge about the other agents in the platform. The communication with other agent components is done through the local file system.

New software versions are installed when the other agent components provide new agent software packages to the updater and when the agent daemon does not have busy signal turned on, i.e. not having a lock file placed in the file system. The updater daemon also monitors the agent daemon running state and can start it if it has been crashed for some unknown reason.

Data Storage

Data Storage module provides access to a local data storage that persists objects into a relational database. The implementation uses 3rd party software component that is compatible with multiple database engines without requiring changes to the application code. For the reference implementation the performance requirements are not very high. Therefore the SQLite database is used for its simplicity and small dependency footprint. All the data insertions and modifications are done through the *upsert* (update or insert) interfaces that takes care of selecting the proper method based on the target object type's unique identifier and the existence of such object. The data access layer should use the connection pooling mechanism to restrict the number of concurrent users and decrease the overhead from connection creation actions.

Data Storage Migration

The Data Storage Migration module is executed when the agent initializes itself. It takes care of upgrading or downgrading the internal database (data storage) schema to match the running agent version. The selected 3rd party migration component requires that all the database schema modifications are done by implementing a type of migration classes, and it takes care of all the complexities involved in the database change management procedures. It also supports several common database engines out of the box.

Task Scheduler

The Task Scheduler module manages triggering periodic tasks' execution when they are due for execution. These tasks include heartbeat reporting that reports agent state and liveness to the CFA, checking available agent software updates and possibly scheduled subscription execution that will install the target software if a new version is available. Task scheduler triggers all tasks to be run asynchronously in their own thread. The threads are using a thread pool that prevents exceeding the available resources of the system.

Task Executor Unit

The Task Executor Unit is the module that handles task execution process. It executes installations according to *installation subscriptions* and *installation plans*. The installation is executed by executing an external installer application that is embedded into an installation package. The executor unit handles communication with the external process and monitors the execution. It is also responsible to kill the external tool processes if they don't provide any progress within predetermined timeout boundary.

Reporting Module

The Reporting module handles creating installation reports and reporting the results through various channels. In addition, it handles producing status reports that are provided to the CFA and management GUI.

Management GUI
Management GUI (Graphical User Interface) is a module that provides user interface for human users to manually access the installation process functionality and created reports. It has two web servers, one for hosting a web application and the other for providing *web socket* connections to the web application clients. The web application is designed as *SPA* (*Single Page Application*) that is a fully functional application made with HTML5 and JavaScript that runs in a modern web browser. The SPA can work without having a constant connection to the backend web server. It connects to the agent functionality through *Internal REST API's* module and web socket connection that provides progress messages. The management GUI allows user to upload installation packages and data structures manually, start and abort the installation processes, to manage installation settings and to see real time progress information from any ongoing operation. In addition, it shows reports and lets user to manually download output from any past installation.

Internal REST APIs
Internal REST API's contains web APIs for serving the *Management GUI* module for fetching the information and controlling the agent operations. Internal REST APIs are meant to be used only through the Management GUI module and not by the CFAs. The data format accepted by the API's is a specific *JSON* format designed for the use. The *JSON* (*JavaScript Object Notation*) format is convenient to be used with JavaScript that is used in the web application. All the modern web browsers have built-in support for it.

External REST APIs
External REST APIs provide controlling and reporting functionality to be used by the CFAs to send commands and query information from the DIA. The data format is also *JSON*. It is a compact and widely used serialized data format that works well with REST API's and is compatible with the other installation platform components' design.

3.4.3 Working in offline environments

Many energy industry companies are very traditional and protective about their data and systems. Therefore, usually their production systems are not directly accessible from the internet or from any non-company networks including the software vendor. These energy companies are not likely to change such security based policy in the near future. Therefore, some of the environments that require installations of the EDMS platform software cannot work with the model where DIA*s* are continuously connected to the CFAs. Such environment that is not able to connect to the CFA is referred in this context as an *offline environment*.

Because the *offline environments* are a remarkable portion of the target environments, it is an essential part of the design to include process with minimal manual effort to support agents that operate without physical (constant) connection to the CFAs. To be able to work in offline environments the DIA needs to become an *intelligent mobile agent* where *intelligent* refers being able to work as a standalone. As opposed to *static*, *mobile* refers being able to travel and run on the target environment without being

integrated into the target system. The designed DIA can be installed as constantly running a daemon service that connects to the CFA and stores its data to a common application data location. In addition, it supports also running as a portable console application that:

- stores all of its data into a data directory that is located in a relative path close to the agent software (preferably in the same media),
- can be preconfigured with all the data required to execute an installation process through its *Management GUI,* while running inside the network that is connected to the CFA,
- can be deployed in any installation media or file share methods to work as a portable application,
- can be run by an installation engineer in the target environment to execute the installation with single button click trigger mechanism,
- collects all the installation data into its data directory,
- can be sent back to the vendor to be connected to a CFA to report back all of its installation results.

To manage an installation in *offline environment* the process requires a small amount of manual work to preconfigure an agent, deploy and run it in a remote environment, trigger the installation with a single click and to deploy it back to the vendor's network to report the results. The process can be made relatively easy. To create and preconfigure an *intelligent mobile agent,* by having wizard-like user interface and *installation plan* template for the target environment simplifies the task. Although most parts of the installation operation can be automated also for *offline environment*, it still requires some manual work that relies on the human user. Regardless of the required effort, the proposed installation process should still provide significant improvement to the previous manual installation process. The difference is that with the remote installation platform, the process requires a very small amount of work time and technical expertise from the human user and is therefore more cost-effective method compared to the previous installation methods.

From the CFA and the platform perspective, the *intelligent mobile* DIA produces the same outcome as the DIA that is connected directly through the network produces. Therefore the only design that needs to consider the *offline environment* use case is the DIA design. From the CFA's point of view, the process works in a way that the CFA knows about the installation only when it is already completed. Process-wise that makes very a small difference to the CFA. In reality, the transmission channel has some extra reliability issues. It relies on the human users' ability to deploy the agent software to the target environment and to upload the results back to the CFA. The CFA can only register the installation if it receives the data generated by the DIA. Therefore the CFA cannot reliably track the states of *offline environments*.

3.5 Designing the central facilitator agent that coordinates the system

3.5.1 CFA design principles

The *central facilitator agent* (CFA) is the agent type that coordinates all the installations, It provides information and all the installation input data collected from various backend services to *the domain installation agents* (DIA*s*) in a format that they are compatible with. In the reference implementation there is only one CFA instance running, but the proposed system is designed in a way that it can be extended in the future to have multiple CFA*s*. Multiple CFAs could share the load generated by the extended amount of DIA*s* connected to the system.

In the proposed solution the CFA is designed to support the installation automation. It provides an administrative interface to create and manage installation automation tasks that are executed according to the instructions whenever a new installable software version becomes available without requiring manual triggering. As part of configuring a new installation task execution of any sort, the administrator decides who will receive reports from the installations. After the task configuration is done, there is no need to manually participate in the installation process, except in one of the following cases: *a)* an installation fails on some unexpected problem, *b)* there is a resource problem, e.g. the available disk space is getting low, or *c)* some process configurations are wanted to be changed.

Since there is only a single CFA in the reference implementation, it is vital to the whole system that the CFA is robust. In addition, the connectivity between the CFA and a DIA is required to be robust. The CFA is recognised to be a single point of failure to the whole remote installation platform. Therefore, there are some practises in place to increase the robustness of the system. The first practise is that if the CFA cannot connect to its own data source (a database service) or to one of the backend systems it relies on, it tries to maintain the operability of the system and reduce the effect of a temporary failure in connection. This is done, for example, by having cached data in the memory to serve the DIA*s* with knowledge of the latest known available versions. Therefore the DIA*s* can make pending requests that can be completed after the back end connections are working again. In addition, if a connection attempt from a DIA fails towards the CFA, the agent tries again several times after short intervals before accepting that there is no connection available.

The CFA design follows the service-oriented architecture and has similarities to a multi-agent systems. It is designed to consist of modules. Some of the modules are working independently and collaborating together toward the common goals. The design has some similarities to the whole remote installation platform in a micro scale. The difference is that there is no redundancy in the modules and the platform operability requires that all of the modules work properly.

It is a matter of future research to increase the robustness factor of the CFA. The long term production usage may reveal some frequent robustness problems. In addition the

data caching patterns for increased performance and reliability can be a focus point for further optimisation. The following subsections describe the details of the CFA design.

3.5.2 CFA component overview

Similarly for the DIA design, also the central *facilitator agent* (CFA) design is built on component modules that operate autonomously from each other (*Figure 17*). The *Data Storage*, *Data Storage Migration*, *Internal REST APIs* and *External REST APIs* modules are fundamentally following the same design as the component modules with the same names in DIA's design described in the section 3.4. The other modules are like micro processes working autonomously inside the CFA agent process and communicating with each other through the *Event System* that supports sending events combined with argument objects.

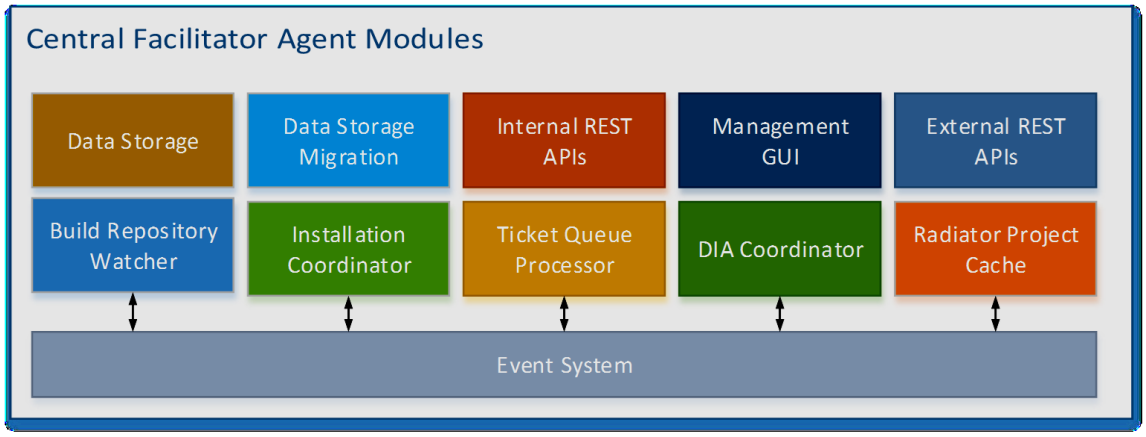


Figure 17. CFA component modules.

Event System
Using the Event System provides a loose coupling between modules that depend on each other. The events are used to notify other components when something important has happened, for example, when a DIA sends status report (e.g. about becoming unavailable) or when a new build has been detected in the build repository. The events are sent and received asynchronously, so that sending and receiving an event does not occur in the same application thread with the component process. Therefore, sending events is not blocking the executing of the sender component's process.

During the CFA initialization phase all the modules register as a broadcast receiver to the events that they are interested in. By doing that the event system will notify all the registered modules when an event occurs. With an event the module's event handler receives a container parameter that contains the arguments that are sent with the event in order to execute the proper actions based on the event.

Repository Watcher
The Repository Watcher module integrates itself to the centralized build repository. The build repository is essentially a file system location with subdirectories for each release branch that contains subdirectories for each build. In addition, repository watcher integrates itself into a *Release Manager* application that provides information about new version releases. The watcher component registers a file system watchers through the operating system APIs to receive notifications when a new directory is created, deleted or renamed. Based on the notifications the watcher module detects when a new release branch or build to a branch becomes available or unavailable. It sends events based on the changes to notify other modules.

To track the state of each release branch and builds in the build repository the module creates a metadata directory inside each build directory. In addition, it stores the information about builds into the data storage. By having two-way book keeping the module can handle offline periods and collaboration with other CFAs monitoring the same repository. Other modules will use only the data stored in the internal data storage about the builds and release branches. The build repository watcher tracks the builds from which it has already sent events previously in order to cover any offline period.

Radiator Project Cache
The Radiator project cache is a cache that works as an information source for 3rd party *build radiators*. A *build radiator* is a visual indicator that shows latest statuses and progress information of build tasks and other similar tasks. The cache is built during the initialization by collecting statuses from all the connected DIA*s* and it is updated based on events that reflect on a possible change in a specific *installation project*. The *Installation project* term refers to a set of all the installations done based on specific *installation subscription* or to a specific release branch.

The cache is optimized to be updated only when it is likely that a status change has happened, or if no status update report for an ongoing installation has been reported during a specified interval. The update operation for a single *installation project* requires connecting to a DIA to query the information. Therefore the implementation has been optimized to reduce unnecessary queries. The project cache is needed because there may be dozens of build radiators that polls the information periodically on a frequent basis and it would take too much time to query the data from all the DIA*s* every time it is requested.

Installation Coordinator
The Installation Coordinator is the core component of the entire platform that coordinates all the installation tasks. It has many responsibilities. For each release branch the administrator can configure the number of *build branch tasks* that are each linked to an *installation subscription* and *installation plan*. When a new build event occurs, the *Installation Coordinator* creates an *installation order ticket* per each of the *build branch tasks* configured for a single release branch. The ticket is put in an *installation queue* to be scheduled for execution when a suitable agent is available. The *installation queue* is a table in a data source that requires always to be accessed synchronously in order to avoid any concurrency problems.

The second responsibility for the installation coordinator is to track new version releases advertised by the *Release Manager* application, create installation packages based on the versions to all matching subscriptions registered to have an interest towards the version, and to notify the subscription owner agents when the installation package is available for download. The third responsibility is to update the *installation order ticket* states that are created for each build-based installations whenever there is a change in the installation states. The important changes in installation processes are reported through the *Event System* to other interested modules.

Ticket Queue Processor
The Ticket Queue Processor is running periodically when the *installation queue* is not empty. The operation goes through the queue and tries to offer each of the *installation order tickets* to a DIA that has the required tags for the ticket execution and is in idle state. If a DIA responds that it accepts the installation order based execution, the queued item is removed from the queue and an event is sent to notify that the specific ticket has been accepted to be executed in specific agent. Based on the information the *Installation Coordinator module* can update the ticket state.

Management GUI
From the technology and design point of view the CFA*'s Management GUI* is identical to the DIA*'s Management GUI*. It provides the administrator capabilities to create and manage installation tasks in a centralized way. In addition, it allows to quickly search and check statuses and information of the DIA*s* from a list view that has searching and filtering capabilities. It also provides quick access to open the management GUI of a specific DIA and to inspect installation results from all the reported installations.

DIA Coordinator
The DIA Coordinator module is responsible for providing and managing the connections to DIA*s*. It communicates with the DIA*s* and can command them to execute specific tasks or operations when required. For example, it can request that all the DIA*s* need to fetch a new updated software version when it is available. It also manages the list of live agents and their states and monitors which agents have not been reporting their statuses periodically. After verifying that the dead agents are unreachable, the coordinator can report the dead agents to the platform administrators.

3.6 Feedback generation from the system

The feedback that the system provides is a central part of the design proposition. The remote installation platform integrates many different tools and software components. Therefore, it is likely that there will be many unexpected problems with their co-operation. In addition, the past installations of the complex EDMS software indicate that it will be common that an installation ends with errors. Therefore, even though the remote installation platform is expected to reduce significantly the human effort required to execute the installations it is very unlikely that it would completely remove the need for human assistance to complete all of the installation tasks.

The remote installation platform can provide good reports that lead the human users in finding and fixing the problems more efficiently. Moreover, the delay how quickly the interested parties get the information that an installation has failed can be shortened by optimizing the process. Early detection of the fatal problems also limits the time and required effort to restore the system to an operational state.

Collecting feedback from the installation process itself can help improving the process further in order to: *a)* produce more accurate estimates for the task execution, *b)* detects bottlenecks that could be solved and *c)* further extend the automation to cover new problem cases that can be solved with some intelligent algorithms. In addition, the collected data can help having a better understanding of the whole process and its cost-effectiveness potential.

The feedback generation has a lot of potential to increase the value the remote installation platform can produce. Therefore, as a part of the proposition for the remote installation platform, also the feedback generation process is thought carefully. The following subsections discuss different aspects of the feedback generation processes.

3.6.1 What data to collect from installations

The first task when planning a feedback generation process for the remote installation platform is to decide what kind data should be collected, i.e. what data is valuable and easy to collect from the process without large investments into the implementation. In addition, when the data is collected from an installation that occurs in customer environment all the collected data should be something that the customer will easily agree on to be sent to the vendor. In addition, the data collection coverage should be configurable in a way that a customer has the ability to completely prevent or limit the data collection to some specific level. Moreover, the collected data should not contain information that may become security issues or otherwise harm the customer in case the data is leaked to the outside world. Otherwise the risks can outweigh any gained value the data collection has to offer.

In the proposed design, the following information is collected:

- success state and execution times of each task and sub-task that is executed as part of the process,
- overall operation success state and total execution time,
- database script execution times per statement,
- all errors and warnings produced by the installation software and validations that are formatted to not contain any business or customer critical information,
- log files and console outputs generated by integrated command line tools and the installation software,
- for each integrated command line tools the argument list that was used in execution having any sensitive data removed,

- database connection information, application locations and internal IP addresses that are valid only in the local network connection usage,
- operating system versions of all the servers that are part of the installation,
- CPU, the amount of memory, disk drive types and other system resources of the target servers, and
- the current version of the EDMS platform before and after the installation.

The justification and the goal of collecting the suggested data is:

1. To identify where the installation fails, what tasks were included and how long the operation takes in order to help preparing the future installations.
2. To have more accurate knowledge where the time is consumed and how that could be improved.
3. To find and prioritise target scripts for SQL optimizations since SQL execution is known to take the major part of the total installation time.
4. To have better knowledge of the shares of different operating systems that utilize the company's product in order to help resource planning for the future development.
5. To help providing better hardware recommendations based on actual data.
6. To have the list of versions with detailed information of the product scope that are used by the customers.
7. To provide faster support capabilities for SLA support personnel in order to let them know quickly where the software components are installed in the specific customer environment whenever there is need to access them.

The collected data scope is designed to cover the needs of fixing and investigating failed installations, to improve the system stability and performance. In addition, it provides value to the customer in the form of better SLA (*Software License Agreement*) support. For the company itself, it provides insensitive, but valuable business intelligence data that has potential in helping planning targeted sales and more efficient development effort based on the customer systems. The data was not easily accessible in the previous installation process in an aggregated manner.

3.6.2 Collecting the data

The data that is linked to a single installation is collected by the DIA before, during and after each installation. After collection, the data containing multiple files (logs and metadata files) is stored into a directory located in the local filesystem. The path pointing the location is stored as part of an installation record. An installation record identified by installation start timestamp is stored into the data source after each installation. If the DIA is directly connected to a CFA it will compress the directory content and send it automatically to the CFA with a data source identifier and installation time stamp after the installation completes.

The report with the installation data may contain a special flag that is set if reporting any previous installation has failed and the CFA has not yet received that information. In that way the CFA can also query for previous reports that were not sent because of some temporary network problems. The CFA stores the installation reports with the data source identifier, installation time stamp and agent identifier being able to easily check if received installation report matches any existing record in order to prevent duplicate entries.

If the DIA operates in an *offline environment* and is therefore unable to reach any CFA, the installation record and associated data is kept in the agent data directory and a reference to it is in the data source in case the agent is connected to a CFA in the future. The written installation instructions that are deployed with the installation software to partners and installation engineers include a request and steps to send to the vendor company the installation report after each installation. Each DIA has the graphical management user interface that has a list view showing all the past installations. From the list, it is possible to download any installation's data as a compressed archive, which can be uploaded to a CFA from CFA's management user interface.

The installation data is compressed whenever it is requested for the first time and then the data is kept in compressed format. The DIA has also a routine to archive (compress) old installations in order to keep the disk usage as a minimum. The data is not compressed right after the installation since it is very likely that administrator users that are report recipients may want to access the files often right after the installation, especially when there may be problems requiring investigation.

3.6.3 The reporting methods

The data collected from the installation processes is only useful if it reaches the interested parties and is represented in a format that is understandable, clear and can display quickly the most important facts. In addition, the reporting channels should be considered carefully based on the purpose and the information receivers' roles. Using only a single channel may cause unwanted delays or have irritating effect to the receivers.

The proposed solution has four channels that are used to report the data. Each of the channels has different targets and they serve different usage purposes. For analysing the aggregated data and searching any history records the management user interface in the CFA provides tools and graphs to access data gathered from multiple DIA*s.*

First of the four available reporting channels that reports installation results is an installation project source API that provides installation task's information in a widely adapted *CCTray XML project* format. The format is compatible with most of the *build radiator* applications. A *build radiator* is typically a dynamic web page that shows multiple projects in a grid view where each project has a box that is coloured based on

the latest status. Active projects may have animated back colour where a project with latest status set as successful is usually coloured with green background. Failed status is represented in red colour. A build radiator screen can be placed into the corridors and team rooms where everyone can see the overall statuses of different tasks. The primary target for build radiator is a manager that is not interested in the details of a failure, but only the overall status of all the installations, and anyone in a development team in order to get the overview.

The second reporting channel is called *Slack's incoming webhook integration*. *Slack* is a popular modern messaging web application designed primarily for software developer teams for sharing and receiving information in a single place *(Slack Technologies 2016)*. The Slack application has chat-like channels that support posting formatted messages with rich content (e.g. links and images) by any 3rd party application through something called web hook URL addresses that are essentially REST-based web service interfaces. The remote installation platform supports posting messages to specific slack channels. The messages posted before and after installation operation are targeted for software and quality assurance engineers. These formatted messages contain included tasks which are coloured based on the end status with red and green colours. In addition, for providing quick access the messages contain other environment based details and links to the results and used installation packages. The Slack application integration is meant primarily to be used with installations that are part of the *continuous integration* system.

The third reporting channel is the traditional email. A formatted email that contains the same information as the Slack channel messages extended with a list of details and all the errors and warnings can be configured to be sent to a list of recipients. The email channel is targeted for users who do not prefer the Slack channel. In addition, it serves as a secondary channel for all the interested parties who wish to be kept up-to-date with the installations for specific environment. The email channel has a high risk to be ineffective since typically the email is an overused messaging method in companies and the amount of emails per day may be high.

The fourth channel is creating a static html formatted report that looks similar to the email. The report is stored after each installation as part of the installation output. The management user interface of a DIA has a link to the reports in the installation list view. It is provided as a complementary report for the email in *offline environments* where emailing capabilities are not available. In addition, it provides a way to check the installation summary quickly for an installation when an installation engineer is starting to investigate a specific installation output. When the installation output is archived, the html report is included in the archive to be available for later views.

It is very likely that the radiator status or the Slack channel message is noticed quickly after the installation operation. Therefore the feedback is expected to reach its most important target audiences with minimal delay. On the other hand, these channels do not cause load to the email boxes, which can be an appreciated feature for many of the report recipients.

These reporting methods have been selected based on intuition and the existing information distribution methods that are used in the company. Since the reporting is

considered to be a very important part of the remote installation platform and its efficiency has significant impact on the value produced by the system, the reporting channels should be improved continuously based on a real feedback collected from the platform administrators and users.

4 Analysis and discussion

This chapter discusses the proposed implementation and how well it matches the requirements that was set to it. It also analyses the operational results collected from some real life use case EDMS platform installations that were executed by using the reference implementation done based on the propositions of the thesis. The results are reflected to the operational results from the past installation methods. No accurate comparisons were made since the previous installation methods were lacking some functionality and the manual work was not tracked in any continuous manners. Therefore, any comparison would be based on inaccurate and incompatible measurements.

The chapter begins by reflecting the remote installation platform architecture proposed in Chapter 3 to the list of requirements presented in Chapter 2. The architecture inspection is followed by the analysis of operational results gathered from installations executed by the reference implementation. The analysis continues by presenting the identified limitations that the proposed remote installation platform has and is followed by analysis on the quality of the used data in the contrast of the actual intended use for such system.

4.1 Proposed architecture vs requirements

The Chapter 2 lists 34 requirements for the remote installation platform design in order to be able to fulfil the remote installation and automation requirements that were set to it in a robust, scalable and maintainable way. The proposed solution fulfils most of them with a proven solution. The security related requirements were acknowledged to be not sufficient. However, the security issues are not in a focus of this research and were decided to be solved in a future research. Matching the remote installation and automation requirements were proofed by several remote installations that were executed with the reference implementation. The reference implementation is following closely the design proposition.

The requirement categories defined in chapter 2 are *installation automation capabilities*, *remote installation capabilities*, *platform robustness and efficiency*, *platform extendibility and maintenance*, and *platform and data security*. The installation automation requirements are achieved by using data structures, *installation plan* and *installation subscription* that can together describe an installation process control workflow, its input and the general rules of automating a continuous installation process in a target environment. The data structures support lossless serialization and deserialization that are used in data transferring the installation from one location to another. The platform uses satellite agents (DIA*s)* to achieve the remote installation capabilities.

The requirements placed into *platform robustness and efficiency* are also covered by the design. Retrying connection attempts and providing the possibility to fetch older data on a query basis when the reporting functionality has not been working provides the system a possibility to become eventually consistent. The consistency refers to the centralized storage being able to collect all the information that any single agent holds assuming the agent will eventually have a working connection to a CFA. In addition, the design addresses some of the bottlenecks that may limit the scalability capabilities of the platform. However, the design uses a level of abstraction that leaves the final responsibility of making the system to be robust to the implementation. Although, using standardized protocols can help reduce the uncertainties produced by the network environment, in the end it is up to the implementation to recover from unexpected problems.

Platform extendibility and maintenance requirements are addressed by having the agent software distribution procedures and automatic update capabilities in the agents. The offline period caused by an agent software update procedure is limited to be short enough (less than 30 seconds) to be fit into the retry period of any connection attempts towards any agent. Therefore, from the timing perspective the platform software can be updated in any agent without having effects, other than a small delay, to the operability. In addition, the design proposition has a single place for a maintenance view that provides the administrator users an easy access to most frequent maintenance tasks they may need to do. Dependencies and 3rd party components can have an effect to the maintainability. The design does not name any 3rd party components to use, but only the functionality they are expected to provide. The thesis suggests to evaluate carefully any included dependencies from the maintenance perspective. The actual responsibility to select maintainable 3rd party components is left to the implementation developer.

Platform and data security requirements are covered only partly and with minimal level of sufficiency. Therefore, the design proposition should not be taken into use in an unsecure network that has agent connections that use the open internet before addressing the security issues; for example, implementing secure methods to pass sensitive data and authenticate the other communication party with up-to-date security measures. In addition, the design would benefit from a more fine grained security analysis of each component and the platform itself. For example, when the DIA requires administrator privileges, it has the responsibility to prevent harming the environment with any input provided by a non-trusted party. In addition, when running external software tools it should reduce the running privileges on per need basis and prevent passing the administrator privileges to any unknown external software processes.

From the overall perspective, the proposed architectural design addresses all the key requirements and provides a prospect solution for them. The security issues are acknowledged to be not sufficient for using the platform in a non-secure environment without first analysing and extending the security level of the design. However, the system design proposition provides the means to implement a working remote installation platform that can be used to automate the EDMS platform installation procedures, which was the goal set for the research. Appendix "*List of all requirements and the solutions presented by the thesis*" (*Appendix 4*) has a table listing all the

requirements presented in Chapter 2 and the corresponding solutions in the proposed design to fulfil them.

4.2 Operational results of the proposed system

Operational results of the remote installation platform are analysed from the installations that were executed using the reference implementation during the testing period that was conducted for this research. All of the executed installations that are analysed in this section were made to fill an actual purpose and the results were used also in the EDMS platform product development. The analysis covers only inspection whether the proposed system can complete the goals set to it. The analysis does not include any results that were affected by a software defect in the remote installation platform implementation and could be fixed in the code without diverging from the proposed design decisions.

The analysis is done by verifying that the remote installation platform is able to execute installations according to the *installation plans* and *installation subscriptions* in an automated manner. Some of the installations were triggered automatically after a new build of the EDMS platform software was completed by the *Continuous Integration* system; others were triggered with a single button click. Requiring one click to trigger the installation in this context is considered as *automated* (not *fully automatic*) installation. In 1-click installation the person who triggers the installation controls when the installation is executed, but he doesn't need to know anything about the current state of the target environment. When a fresh installation or upgrade of an existing EDMS platform instance is completed and the next version that matches the configured *installation subscription* becomes available, to consider the installation process automated, also the installation to the next version should require only 1-click or less depending on the case.

The other goal for analysing the operational results is to verify that the status and problems the remote installation platform reports match the actual situation. The installation status should be successful only when the operation was completed without errors and the installed instance state represents expected outcome. In addition, if the installation operation was failing, then stopping on error mechanism should match the configurations defined in the target *installation plan*. Whenever a problem occurred during the installation, the root cause was inspected to see if it was related to the remote installation platform or to the target software product. If the problem was caused by a software defect in the remote installation platform the bug was fixed and the installation retried.

4.2.1 Executed installations

The analysis reveals that the remote installation platform works, producing the value that was expected from it. The platform is able to execute the installations according to expectations and the user has control to trigger and abort operations without more than a few seconds delay. The reports are delivered according to the configurations and the actual results match the reported outcomes.

During the testing period conducted for the research, the remote installation platform agents were installed to more than dozen environments. Several hundreds of successfully executed EDMS platform installations proof the concept of the designed platform's capabilities to execute remote installations in an automated way. The number of agent nodes used in this analysis that is based on actual installation operations does not represent the near future number of agents. However, it demonstrates that the remote installation platform can operate and manage with several concurrent installations ongoing.

The selected number of DIA*s* was dominated by the number of available real installation cases during the testing period. The candidate cases were considered from the business risk perspective and were included only if the risk was minimal. Therefore the installations included in this analysis cover *Continuous Integration* based installations, some development and testing environments and only a single *offline environment* that was set up with a business partner to be a playground for their needs. The latter environment was the only environment that was not an in-house environment for the software vendor company. Table 2 lists statistics collected from analysed installation data.

Table 2. Analysed data collected from EDMS platform installations that were executed between 20.4.2016 - 30.8.2016.

	Count	% of all
Total number of executed installations	463	100%
Number of participating DIAs	15	100%
Different customer package scopes included	13	-
Failed installations[1]	249	54%
Upgrade installations	255	55%
CI-based installations	55	12%[2]
Failed in file installation task[3]	34	7%
Failed in db installation task (excl. conf. import)[4]	8	2%
Failed in db configuration import task	136	31%
Executed with all main tasks included[5]	306	66%
Succeeded with all main tasks included	154	33%
Shortest failed installation duration	1s[6]	
Shortest succeeded installation duration	6m 53s	
Longest failed installation duration	21h30m24s	
Longest succeeded installation duration	23h59m9s	

[1] Succeeded installation refers that no errors occurred, failing does not imply that the remote installation platform does not work as expected, but that the target software has problems.
[2] CI-integration feature was completed in the middle of august and after that 37% of executed installations were CI-based.

[3] File installation task failed occasionally for insufficient disk space but most commonly because some of the target files were in use by other parties during the installation operation.
[4] Database configuration import was never successful when database installation task failed.
[5] All main tasks include file installation, database installation and db configuration import. The number indicates the variating nature of different installation scenarios.
[6] Execution failed on pre-validation

Some of the installations were used to execute a fresh installation to an environment without any existing EDMS platform software instances, and others to execute continuous upgrade installations to the same environments according to *installation subscription* boundaries. The end state of the target software was not noticed to be any different compared to the previous installation methods. Since analysing the performance is not the main focus of the research, repetitive installations were not executed to compare performance variations between runs.

Over half of the executed installations ended with failed installation status. The results and the reported errors were compared to results from installations of same content done with the previous methods. In all of the cases, the results were as accurate as or even more accurate than the results from the comparable installations. Having more accurate results were due to fixing some software defects in the installer software while extending it to have remote installation platform compatible execution mode.

The outcome where number of failed installations is greater than the number of succeeding installations was in the line with the expectations. It is the reason why as many installations are required and that testing the installation is important to include as part of the automated test done for each new version of the complex EDMS platform product. Only a very small portion of the installations failed on an interference based problems where the end result can be different if the installation is retried. Almost all of the non-successful installations failed on a problem in the EDMS platform software. Fixing them required changes in the target product. Most of them was caused by problems in some of the database installation scripts or in the data importing command line interface that is part of the product.

4.2.2 How well the tools and preparations worked for the task

How well the new method for managing the EDMS software installations works for the purpose, is analysed by comparing the outcome to the previous situation. The most significant change is the value that the new system was designed to provide. It provides the capability of remote installations and the new level of automation that was not previously available. In this light the design and the implemented reference platform provide clear and tangible value and solution to the problem presented in the research.

The installation process itself has some differences to the previous installation methods due to changing the order of some actions. However, it does not add parallelism nor skip executing any previously included actions that are relevant to the installation tasks. Based on the results, the installation execution time was noticed to be remarkably faster in some *Continuous Integration* based cases that are comparable due to using

same input installation version and target environments. The core reason was not inspected thoroughly as it is not part of the analysis goals. Since the used environment and the outcome were the same, the expectation is that the installer software runs faster in the mode that does not use user interface during the installation compared to the previous installation method that shows the Windows compatible user interface. Refreshing the user interface requires a lot of synchronization operations between process threads in the code and lacking that need is likely to be the reason for having better installation performance. The decreased installation time was an unexpected but tangible benefit gained from the remote installation platform. The non-user interface execution mode was implemented to the installer software in order to get it work reliably with the remote installation platform.

The differences between an upgrade and a fresh installation are very minimal in the remote installation platform. After preparing an environment with initial prerequisites, the same *installation plan* and *installation subscription* can be used to first execute the fresh installation and then continue to keep the target environment up-to-date with upgrade installations without changing anything in the remote installation configurations. The automation included in the platform handles matching the correct installation version range for each installation, which makes the operation to be very convenient for the installation administrator person compared to the previous installation method.

The analysis did not reveal any properties of the installation process that would have been noticeably degraded compared to the previous installation process. According to the free feedback provided by the users the performance was improved, the usability is better due to the modern web user interface. Moreover, after configuring the process, it requires very little knowhow to trigger and manage the installations in an environment. Therefore, the installation is less dependent on specific persons and their availability. In addition, having the logs and results easily available in a single place made the investigation of a failed installation significantly easier and quicker. The previous method did not manage the logs and outputs per installation session. It used same log files continuously for each installation that were growing larger over time.

4.3 Limitations of the proposed platform

The design proposed in this thesis has some limitations in its current form. Some of the limitations can be removed or reduced by extending the design with new functionality; others may require replacing some of the existing solutions with different ones. Some of the components are so critical that having them malfunction or stop working can make the whole remote installation platform become inoperable or cause some severe damage to the existing environments.

The current design based implementation cannot handle all of the EDMS platform installations to all of the existing environments. The fresh installation cases require some manual work to prepare an environment with some software and configurations. In addition, some production environments have some heavy fault tolerance

mechanisms provided by different vendors that cause interference or prevent any installation process execution with the remote installation platform. These cases require human interaction.

4.3.1 Recurring Problems that require human interaction

Encountered issues that require human interaction to be solved degrades the level of automation that the remote installation platform provides. The less human interactions are needed, more the system can scale without adding more human resources to tender the problems that may occur. In addition, requiring human interaction decreases the throughput of the system and can reduce the utilization level of the resources when they cannot be reused until the administrative task has been completed by some human. Therefore, all the human interaction needs and time spent on them should be tracked in order to make it possible to enhance the performance and operability of the remote installation platform with solutions that can reduce the human interaction need on known problems whenever that is possible.

Not all the manual work is worth automating. If the need for any manual task is not frequent the payback period for the implementation and extended software maintenance effort may be overwhelming compared to the produced value. Therefore, the cost may not be paid during the remote installation platform's lifespan. For example, some of the existing environments that are running the EDMS platform have fault tolerance systems. These systems monitor services and resources and turn them on if they are stopped without using the fault tolerance systems' interfaces. That can cause problems with the installation process executor unit that relies on standard operating system functionality to execute such tasks. Since there are many different kinds of fault tolerance mechanisms that would require specific and possibly complex integration implementation to support the fully automatic installation process.

The production environment updates are not as frequent as the new installations occurring in testing environments that are used to prepare production installations. Therefore, any implementation costs that can be utilized only in a single, less frequently updated environment may not be justified, if the required human effort is only e.g. few hours per year. Extending the automation to cover such environments should be considered case by case.

The proposed design and its reference implementation do not include solution for service configuration management that is part of the file installation operation. It remains as a manual task for some fresh installations that include such services. It is also a manual task for some upgrade installations in case there are changes to the configurations included. The reason for not including it as a part of this research and its implementation was the complexity of creating a safe and valid algorithm that would cover all the requirements. In addition, the product management office of the target EDMS platform software suggests that such services are probably going to be deprecated in the standard product in the near future, which reduces the worthiness of automating these configuration tasks.

When the implemented remote installation framework has been used, some EDMS platform installations have failed. The failed installations that are not caused by problems in the remote installation platform operability require always some administration work. The most common causes are identified to be a database upgrade related that are caused by either the incorrect order of applying database changes with dependencies or data related problems where existing data in the database cause some problems. These are complex problems to solve in generic automated manner. When the remote installation platform has operated longer and more data is collected of the failures, it may be possible to identify patterns or find ways to expose the problems earlier. The goal should be to increase the rate of succeeding installations as well as to decrease the execution time of an installation before detecting problems that will lead to a failed installations. If the problems are revealed quicker the fixing can start earlier and that will benefit both the customer and the software vendor.

4.3.2 Known sources for errors

This subsection discusses the errors and problems in the remote installation platform proposed in the thesis and the problem sources that can cause the errors. Some of the errors were encountered while using the reference implementation that was developed based on the design proposition introduced in this research. Other possible error sources are acknowledged and accepted as part of the platform.

The external applications that are executed as child processes are always a risk to become a source of problems. During the period when the reference implementation was used for this research, there were some issues caused by the external tools. For example, continuous executions of file archiver tool (*7-Zip*) caused some random file locking issues that required changes in the implementation for better compatibility. In addition, a data and configuration importer tool used in the database installation had a defect where it reported no-error exit code while the operation was encountering some failures that should have caused the installation status to become failed. Due to the problem in that external tool, which was the only failing operation, the installation was reported falsely as succeeded. The problem was only caught by the manual verification effort. In another case an installation tool failed to report some problems with console output, which is the general behaviour of the tool and in that specific case it wrote the errors only in its own log. The resulted problem was that the remote installation platform reported the installation status as a failure, but did not provide details in the installation report why the installation failed. That caused confusion for the users.

These problem cases that occurred with the used external tools demonstrates that any external tool used by the remote installation platform can have a direct effect to the robustness, reliability and quality of the remote installation software. Moreover, it is likely that not all of the problems are revealed while the remote installation platform implementation is tested. The above kind of problem especially is very hard to detect in an automated manner and it almost always requires additional testing methods in order to be revealed. Therefore, the platform administrators should do some manual inspections and verifications on continuous manner in order to find any unexpected

behaviour that is not visible in the reports. Also, the external processes should be executed always with some timeout value of maximum expected execution time in order to detect any deadlocks, so that no external software is able to halt the installation indefinitely. In addition, some external tools can unexpectedly require human interaction that will halt the execution in an automated system that is not configured to provide any input.

Human users can also cause interference with the installation process. They may accidentally terminate processes or lock files that are used by the installation executor process. During the testing period for the reference remote installation platform some of the installation target files were in a shared network location and locked by some users' active processes. The implementation does not contain neither the means to identify the users responsible for the interference nor to unlock the target files so that they could be updated (replaced). In some target environment the chance of the problem occurring was noticed to be quite high due to the working methods the owner team was practising. The problem can be reduced by providing notifications with a request to close any relevant processes in the system to target audiences before all installation processes are started, but any notifications are not enough to remove the actual problem. Finding more efficient solutions for the problem should be a topic for improvements.

One unexpected problem discovered in the reference implementation was related to the proxy server configurations set for each machine in the company's network. The agent software uses web sockets in communication from the agent web server towards the management client application to send progress notification messages during installation. The company's proxy application's rule set required adding the web socket protocols to be bypassed by the proxy in order to get the web socket connections to work. The proxy software seemed not to work with web socket protocol's traffic. The same problem was not detected in another network where the agent software was tested, but the problem is expected to occur eventually in some customer networks. Therefore, the prerequisite document was extended to include web socket support or similar configuration as described here. Changing the progress messaging implementation from web socket based server push mechanism into a client based poll mechanism was considered to add extra complexity and more unnecessary network traffic compared to the magnitude of expected problems that some web proxy software web socket compatibility can cause. The web socket protocol technology is maturing and spreading in network applications and it is to be expected that the support for it will grow rapidly.

Support for different web browsers is a common problem in web applications. In the remote installation platform the management clients are web applications. Apart from some minor issues in table data ordering functionality and the format of expressed date formats there were no issues in cross-operability among Mozilla Firefox and Google Chrome web browsers. Internet Explorer browser required adding some security exceptions when the default browser security setting level was used. Other browsers were not tested with the platform applications since Firefox and Chrome browsers were the main target for support as they are the most used web browsers in the Company X.

The DIA software was defined to require Windows administrator privileges to run. It was expected to have some credentials and privileges related problems in some of the

environments. The agent software has a self-diagnostic procedure during the initialization phase, which confirms that it runs with administrator privileges and provides detectable error message if that is not the case. During the test period of the remote installation platform, there was no privileges related problems noticed that were not due to any software defects, which could be fixed with patches for the agent software. However, the user who configures the installation procedure is responsible to make sure the agent software is granted access and enough privileges to all of the required resources, e.g. installation target network locations. The remote installation platform aids in the process only by providing clear details of the cause of failure in privileges to the administrators. It does not interfere or modify the security policies and settings in the target system even though it may have been granted privileges to do so. Altering security and access policies is not part of the remote installation platform design. Such functionality could result some anti-virus software to consider the remote installation software to be a threat to the system, which could be very problematic to solve.

4.3.3 Performance requirements

The remote installation platform design shows that all the installations depend on the backend CFA services. For the reference implementation, only a single CFA is used. It means that every DIA depends on that one CFA. Moreover, the CFA utilizes resources, for example, a shared file system through SMB protocol and a PostgreSQL database in addition to some web service API's of other applications. From the architectural point of view, the CFA is a viable candidate to be the first component that can become a bottleneck that eventually prevents the platform scaling.

A reference expectation of the maximum number of DIA*s* that the remote installation platform may include is less than 1000. The number is based on the number of customer installations the EDMS platform has currently added to the number of employees the Company X has and including optimistic 50 % growth possibility in the near future. At any point of a DIA's lifecycle, it should be highly unlikely to exceed the 10 concurrent active connections towards the CFA. In general, the design suggests creating any connections based on triggers that occurs only when it is really needed. In addition, the few tasks that are set to run periodically and require connection to the CFA, should run less frequently than once per minute. The number of employees in the Company X is included in the calculation since they may have a radiator software that connects periodically, once per minute by default, to the CFA to fetch the installation project statuses.

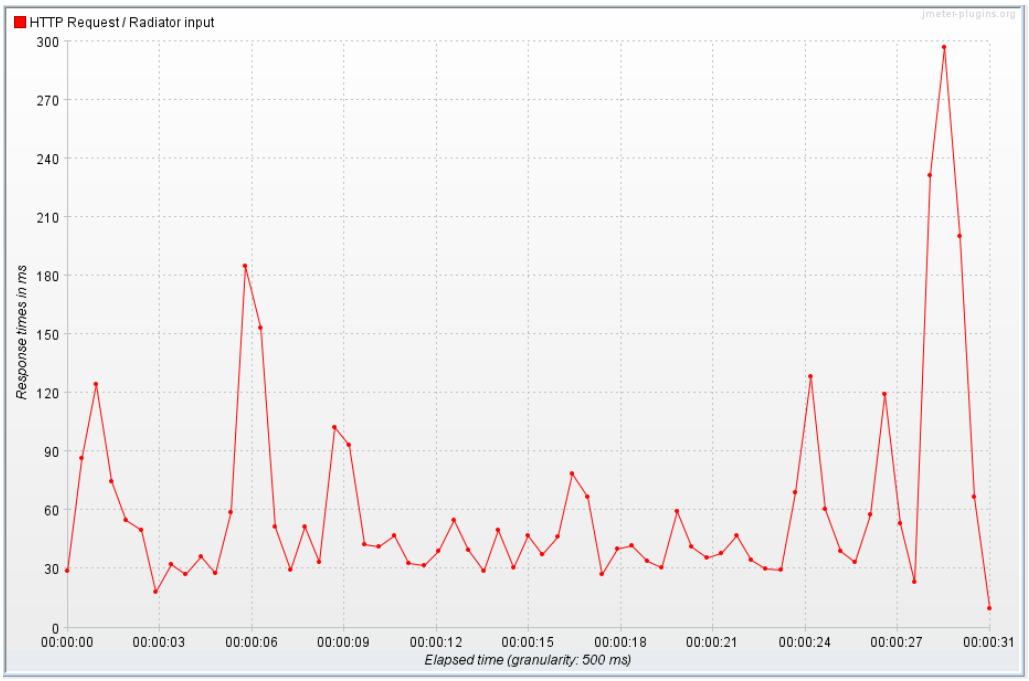
In the reference implementation, a typical connection towards the CFA can tolerate 60 seconds waiting time before timing out. Based on these assumptions, the scaling of the CFA should be sufficient for the near future, if it can handle 10 000 connections per minute. Moreover, most frequently occurring connection requests are served from the caches that do not require services from the backend systems.

The design does not define any requirements for the file system that provides the installation packages distributed by the CFA. Typically creating and downloading an installation package takes a few minutes. Compared to the total installation time, the time spend on acquiring the package is only a small portion of the overall process. Therefore, if the I/O operation time would increase 100% or more during the peak moments that would not cause any significant impact on installation times. In addition, the network bandwidth the *Company X* is having is not considered to be a bottleneck for the performance or scaling capabilities of the remote installation platform. If proven otherwise, the I/O throughput and network bandwidth resources can be increased to serve any changed performance requirements that the remote installation platform may have in the future.

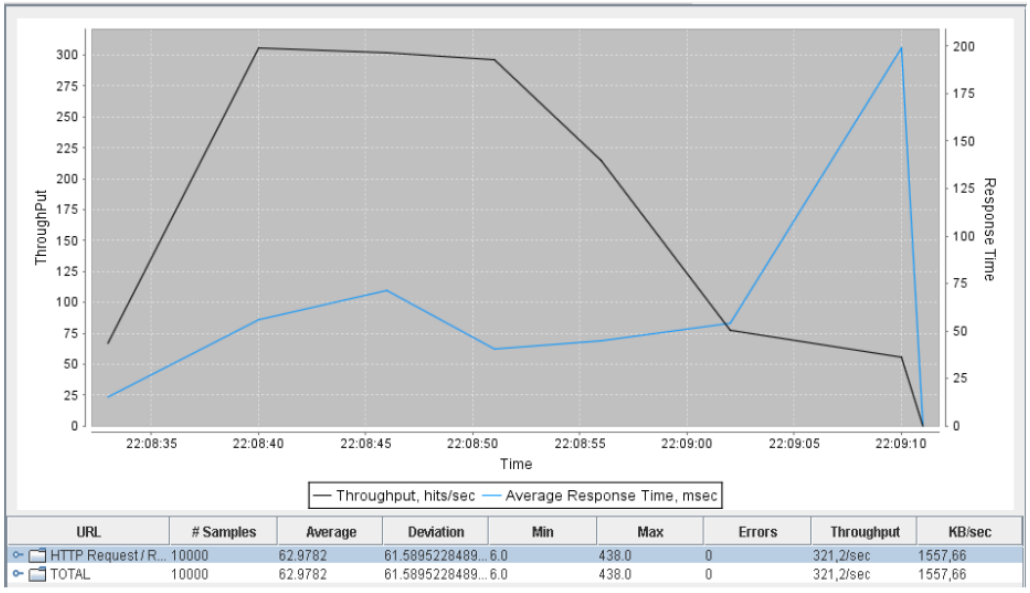
For the only CFA in the reference implementation, a PostgreSQL database was selected as the backend data storage. With enough CPU cores and memory available and proper indexes in place, the database engine is expected to provide enough performance to support very large numbers of concurrent connections. It can be configured to utilize the available CPU and memory resources to serve the queries. In addition, for the future scaling, PostgreSQL has extensions that can distribute the database over multiple machines to scale horizontally. Since the amount of data that the CFA needs to access is not particularly large, distributed database is not considered to be needed for filling the current scalability requirements of 10 000 connections per minute.

Appendix 5 (“*Executed performance test scenario setup with Apache JMeter application*”) has details about the executed performance test. The test was executed just to give a reference level of the performance capabilities with a virtualized database server and application server hardware having a few cores and HDD disks. Since the hardware has a significant role in the data fetch performance, the purpose of the test is just to show that with enough hardware resources, it is possible to serve the expected load with the proposed implementation. Exhaustive performance testing is not in the main focus of this research. Therefore the performance test was run only for a single API (“radiator API”) that collects data from multiple sources to be able to show ongoing activity and latest result indicator in a screen views called “status radiators”.

The figure below (*Figure 18*) shows the results of the executed performance test that uses parameters described in appendix 5. The picture “a” in the figure shows that the response time was maintained below 300 milliseconds during the whole time, which is within acceptable delay a human user could tolerate in such status monitor. The picture “b” shows that all the 10 000 requests were responded properly without errors and the throughput was high enough to meet the goal. The test used an assertion for the response value to verify that the received payload had expected data format instead of only relying on expected HTTP status codes of the response header. Overall, the test result shows indication that the system can handle the expected peak loads. However, it also shows that with increased load the response time starts growing rapidly leading decreased throughput. Therefore, with the used hardware setup, it is expected that the upper limit magnitude of requests that the system can handle is close in line with the goal set for the executed test.



a) Response time variations over elapsed time.



URL	# Samples	Average	Deviation	Min	Max	Errors	Throughput	KB/sec
HTTP Request / R...	10000	62.9782	61.5895228489...	6.0	438.0	0	321,2/sec	1557,66
TOTAL	10000	62.9782	61.5895228489...	6.0	438.0	0	321,2/sec	1557,66

b) Throughput and average response time for 10 000 samples.

Figure 18. Performance test results for testing 10 000 requests inside 60 seconds period.

The time consumed by an installation process for a single environment is considered to be a performance issue in some cases. One of the test installations for the remote installation platform is a national data hub installation with 16 domains for different energy companies and utilities. The current installation process that the installation software supports is mostly sequential. The executed fresh installation of such system with minimal EDMS platform scope takes approximately 22 hours in a valid testing environment. When extending with the actual delivery scope the installation process can take more than 1.5 days. Having that long installation time is non-ideal not only

because the feedback interval is long, but also because for *Continuous Integration* purposes, testing nightly builds will require more than one environment just to test that one specific installation scope for each build. For these reasons a data hub type installation process should be improved to utilize some form of parallel execution.

4.3.4 System robustness

System robustness is measured by how well the system can tolerate unexpected and expected problems and remain operational. It is the ability of a system to resist change without adapting its initial stable configuration (Wieland 2012, p. 890.). The remote installation platform has many potential risks that can degrade the operability and cause problems.

Analysing the system robustness is important in order to prepare and prevent any problems that may occur. One means of analysing the platform system level robustness is to analyse could the system be moved from one set of hardware, a single component at a time, to another without having any effects to the continuous operability. If the system can operate continuously through the whole move operation, it has a high enough robustness for tolerating at least one missing component at any time without having critical problems. The problem can be considered as critical, if some task will be left undone and the system will not eventually recover to a state where the non-operational phase had no effect to the end state. A robust system should redo any missing actions or achieve the same outcome in other means after a component that was unreachable returns operational.

Based on the design architecture analysis, the remote installation platform would not presume full operability during a complete move to a new hardware. There are many dependencies in the remote installation platform that are critical for new installations to start. That means the system has many single-point-of-failure components. The problem could be fixed with increased redundancy by duplicating the backend systems, but it can be also considered as an acceptable condition. The condition can be acceptable, if it can be tolerated until any defecting system can be fixed or if it doesn't cause any critical problems to any ongoing operations.

The DIA*s* have been designed in a way that after they have acquired an installation package and started to install it, they don't require any connection to the agent platform in order to complete the installation task. In addition, any effect of a failed connection attempt is recovered by other means so that eventually temporary connections won't have any significant effect. For example, if a DIA fails to report its liveness with the heartbeat reporting the CFA tries to first connect to the agent before removing it from the list of active agents. In addition, if an agent reconnects to the platform after being flagged as inactive, it will retake the same role and identity that it had before. The remote installation platform tolerates failing components or losing backend systems as long as they are restored at some point. The temporary missing component can cause delays to some operations to be completed, but as not expected to occur frequently, they are considered acceptable in the platform design.

The continuous software development practise that brings new functionality and changes to the system after it has been taken into production use includes some risks for the system robustness. During the implementation and testing phases of the reference remote installation platform, some DIA*s* became inoperable after updating their software. The problem was caused by an incompatible data access library that broke the agent updater software. Solving the problem required updating each DIA software manually. If the number of agents becomes high enough that kind of problem can cause a huge cost in repair.

The case showed how critical part of the system is the updater's capability to recover and restore operability in any case. Moreover, the design does not protect the system operability from mistakes in the deployed agent software, if the software changes have an effect on the update deployment mechanisms. The size of this kind of problem depends on the number of affected agents. For example, if only a few agents need to be fixed every now and then that may not be a significant problem as would be if all agents suddenly require manual tendering. To prevent high risk problems caused by new agent software deployment, there should be another remote installation platform instance that is used only to test any changes in the platform software before deploying them to the production platform. In other words, the same software development practises that are used with any production systems should be applied to the remote installation platform as well.

One robustness factor of the platform is the selected technologies and 3rd party components. If a technology or component is not reliable or actively maintained it can have a negative effect to the system robustness. The selected technology or component should be mature and proven already in similar use, possibly standardized and well accepted. Alternatively, the 3rd party component can act in a role where it is not very vital to the system and possibly can be easily replaced at any time without significant effort.

Any 3rd party component can become non-optimal or even unavailable in the future. Therefore the investment and the level of integration should be thought carefully, also from the perspective of long term robustness of the system. The remote installation platform uses standardized protocols like TCP/IP and web sockets, widely adopted web technologies like Java Script, *JSON* and REST, mature C# programming language and the .NET framework developed by Microsoft Corporation. All of these choices are relatively safe since they have a large support and user base. In addition, the platform design uses a component model where all the functionality is compartmentalised and therefore support easy replacement. The graphical user interfaces use Java Script libraries that not all may not be widely used, mature or actively developed but they fill the requirements. These small user interface related 3rd party components are not considered to be critical and have only effect to the user experience, not to the system operability. In such cases, selecting less safe components for the long term usage has very little effect to the robustness. Based on the analysis done for the selected software technologies and components, they are not considered to cause problems with the remote installation platform robustness.

The most significant risk to the system robustness comes from the external tools used in the installation process. The tools are mandatory to achieve the goals of the remote installation platform. The cost and effort of implementing all the required functionality

in-house go far beyond the risk coming from the selected external tools. Therefore the risk must be accepted, but minimized with compatibility and integration testing means.

4.4 Measurement quality analysis

This section discusses how well the use of the remote installation platform done for this research represents the actual intended use of the system. For instance, in order for the results to provide any insight to the real world use, the set of executed installations should represent reliable amount of variations that occur between some typical installation cases. In addition, the input data and the existing data in the existing EDMS platform application instances used in the installations should be similar or exact match to data that is stored in the real target environments.

The operational results are based on analysis that is done by inspecting the output from installations that were serving actual purposes. The installed testing environments, performance testing environments and development environments were taken into use or continued to be used after successful installations. If the installations failed, they were fixed so that the environments could be utilized.

Some of the installation results were gathered from *Continuous Integration* installations where the use case is to get the installation results and to determine whether the installation works without errors or not. In this sense, the data were collected from actual installation cases that the platform will continue to execute in the future as well. However, the results do not cover any installations done for production environments that have production magnitudes of data and performance of the hardware. From the remote installation platform concept point, production environments may reveal some new problems that need to be solved. However, they are not required to be used to prove that the remote installation platform serves its purpose and matches the goals, hence are not vital to the research. Moreover, one hundred percent coverage of all the installations was not set as a goal for the system.

The installation results contain enough variety to cover most frequent installation cases and to provide proof that the design works for the requested tasks. When counting the numbers, most of the installation cases are done to run C*ontinuous Integration* based testing or to update a manual testing environment. The Quality Assurance goals for the EDMS platform includes installing only successfully tested, installable versions to the production environment. Therefore, the focus in analysis should also be in the more frequent installation cases rather than on the production environment installations that represents only the minority of all the installations in real world use. Therefore, lacking production installations in the analysis input data is not considered to be diminishing feature for reliable analysis.

5 Conclusion

5.1 Research results

This section presents solutions to the research questions. It is structured by listing research questions that are followed by paragraphs presenting the solutions the research proposes for the above questions. Some solutions address multiple research questions. In these cases, all the addressed research questions are mentioned after the sentences. At the end of the section, research results are concluded.

How to run fully automated fresh installation of the product to a Windows machine that is accessible through connected network? (RQ1)

How to have a fully automated upgrade of the product to an existing system that is run on a Windows machine that is accessible through connected network? (RQ2)

The purpose of this thesis was to design a remote installation platform that can run fully automated installations of a specific EDMS platform software including fresh installations to an environment without any existing instances (*RQ1*) and upgrade installations on existing instances (*RQ2*). The proposed remote installation platform is done by using a multi-agent system architecture that follows the service-oriented-architecture paradigm. The platform includes one *central facilitator agent* (CFA) that provides the human administrators a management user interface. In addition, it includes a sufficient number of *domain installation agents* (DIAs) that each are responsible for running installations into a single target environment. The agents communicate and transfer data through a network by using standardized and widely adopted web technologies.

For proving the concepts and ideas presented in the design works well in practise, a reference implementation was implemented. The implemented platform was not only targeted to be used for the research, but to serve as an installation management platform in a long term real world use for the C*ompany X* that invested in the research. The design was proofed to be working for the intended purpose. The remote installation platform managed to run first a fresh installation into a several prepared environment and then continue installing succeeding new versions according to the intended plans to the same environments whenever they become available (*RQ1, RQ2*).

How to define installation that can be automatically installed without required human effort during the process? (RQ1.1)

What things in the installation process can be changed to improve the automatic and remote installability without having an effect on how the application to install works? (RQ1.2)

To achieve the automated remote installation goals, the installation process steps were modelled as tasks and sub-tasks and placed into a dependency graph. The graph shows the required order of actions to not change the outcome compared to the previous installation methods (*RQ1.1, RQ1.2*). The order of executing installation tasks or the actions they include was not changed to achieve the goals of this research. The identified dependencies can be utilized in the future research if the goal is to decrease the installation process execution time by increasing parallelisation in the process execution.

How to reduce required effort to put up several copies of the same environments with same installed application scope close to the effort that is required to put up one such environment? (RQ4)

Installation plan and *installation subscription* data structures were designed to define an installation process workflow specified in a target environment, with required input data and version installation boundaries when the installation of a specific version is allowed and required to occur. The *installation plan* data structure defines a specific installation process workflow that can be applied to several environments with the same kind of target maintenance software setup (*RQ4*). The *installation subscription* provides the target environment based values and boundaries for accepting new versions of the target software. Cloning, reusing and creating new *JSON*-formatted plans and subscriptions for installations are made simple. The frequent management tasks can be made easy for human users with an intuitive user interface for managing the data structures (*RQ4*). RQ4 is addressed also in the paragraph discussing solutions for RQ3.

How the upgrade installation differs from a fresh installation? (RQ2.1)

By using an *installation plan* and an *installation subscription* data structure, it is possible to define automated continuous installation intention that can be transferred and executed in a target environment. The target environment can be physically located in a remote location that is connected to the same network with the remote installation platform or with few manual steps, an active connection is not required. In addition, by using the remote installation platform and the data structures based installation process workflow, for the installation administrator person, there is no difference between fresh and upgrade installation apart from the different installation execution time magnitudes (*RQ2.1*). The differences in the installation process are hidden inside the automation functionality.

How to remove completely or minimize the difference in installation procedure, whether target system is located in local area network, on a network connected through the internet or on a virtual machine running in public cloud? (RQ3)

How to minimize new problems introduced by remote installation? (RQ3.1)

How to minimize or abstract the requirements from local environment? (RQ3.2)

The remote installation platform can be extended to handle a new environment by installing a new DIA into the target environment. The installation procedure requires less than 10 minutes of manual effort (*RQ4*) and only very basic level technical

knowledge. By using the DIA*s* that run as a normal process in the target environment, the remote installation platform is allowed to have the same capabilities as there is available in a locally occurring installation (*RQ3, RQ3.2*). The significant difference is that there is no human user or human interaction available during the operation since the installation relies on automation. Therefore, the installation process must have a fully predefined execution workflow that can always complete (end) without requiring new input from the user (*RQ3.1*).

The agents that are part of the platform communicate via a network using standardized protocols (*RQ3.1*). Therefore the platform does not know or care how the network is implemented physically or logically, with hardware or software, or if the target environments are running on top of actual hardware or as virtualized machines (*RQ3*). All of the participating agents are *intelligent agents*. They can operate independently without full time connectivity and include mechanisms to tolerate connectivity network related problems (*RQ3.1*).

What type of errors occur during typical installation and can some of them be avoided, automatically resolved or detected earlier in the installation process? (RQ4.1)

The typical problems that lead to certain failure in installation and can be easily detected, include lacking of resources (e.g. disk space), missing connectivity between systems and data sources, and invalid input data used in the target environment (*RQ 4.1*). The installation process includes automatic pre-validations for all the currently known environments based failure conditions that can be easily detected in order to provide quick feedback from certain failures (*RQ4.1*). In addition, to extend the pre-validations in the future, the platform collects data from each installation that can be used to discover new conditions, which can lead to certain failure in the installations (*RQ 4.1*). In general, the remote installation platform does not try to fix these problems automatically, because there may be a risk for customer agreement violations that cannot be detected by the platform software.

Besides the savings in human effort, what extra benefits does the automated installation provide as a side product compared to manual installation? (RQ5)

What valuable or measurement data can be collected from the installation that can be used in analytics purposes to grow business potential or efficiency? (RQ5.2)

Besides saving human effort the automated remote installation platform brought also other features of value. The collected, aggregated data from installations can be used to develop analytics functionality that can help identifying the best focus points for future development. The feedback not only reveals the most frequent problems and more accurate execution times of individual actions in a specific environment, but also for example, the share of different operating systems that are used to run the EDMS product (*RQ5, RQ5.2*).

How the faster feedback cycle from of a non-working installation done on top of a new build can be provided when automating installation as part of CI operations? (RQ5.1)

Instead of using a dedicated machine to run the *CI* test installation for a specific customer installation scope the remote installation platform can utilize environment resources dynamically. Using dynamic resourcing can speed up starting of a new installation for the same installation scope while some other environment is still running a previous installation (*RQ5.1*). Having a dynamic rather than dedicated environment resources is useful when the build times can be shorter than the installation times. Moreover, the used pre-validations of conditions leading to a certain failure can provide the failure feedback significantly quicker than same conditions in the previous installation procedure (*RQ5.1*). In addition, the modelled installation process workflow for the automation purpose provides insight on further improvements of the installation process to make it more efficient and quicker (*RQ5*). One of the agreed improvement targets is to make the process to increase parallel execution of non-dependent tasks.

Research topic conclusion

The remote installation platform was designed to be used with a very specific, complex EDMS platform software installations. Since most of the target software specific installation characteristics are isolated into a separated installer tool, the most concepts of the design can be utilized in some other similar remote installation or installation automation platforms as well.

The design was proved in practise. It provides the much needed efficiency and cost-effectiveness for the often needed task that is part of the software development and delivery process. However, the design is still missing some key functionality regarding security. Regardless of the lacking security measures, the proposed design already provides value in a closed, isolated environment where security is not as essential as it is in the open internet.

There are other generic multi-agent platforms (e.g. *Akka.net and Boris.net*) and remote configuration management systems (e.g. *Salt Stack* and *Vagrant*) that handles installations. The multi-agent platforms generally focuses on solving the communication patterns and delivery guarantees, where the message formats and other functionality is left to the developer. The remote installation platform design proposed in this thesis selects the communication approach per case instead of relying on one solution to fill all of the needs. Both approaches have their pros and cons from the implementation cost versus a value perspective. The available generic remote configuration management solutions seemed not to be sufficient to manage the required installation processes with all the specialities, tasks and rules that are involved in the target EDMS platform installations. The potential of the remote installation platform and the lacking of proper available solutions was considered to justify the investment.

5.2 Future research

The proposed design provides a solution to solve the problem of automated remote installations and to have a centralized way of managing the installation. While the focus

was on having the two capabilities, some of the subjects were out-scoped from this research and left for the future research. This section discusses briefly about the potential topics for future research.

The most critical topic for being able to adapt the proposed solution for more open and unsecure environments is the end-to-end security. One of the future research topics is to take a security angle and conduct a research on how to manage credentials required for the installation in a way that the customer does not need to share them with the software vendor. In addition, any transmission that contains sensitive data should be protected with modern security measures.

The scalability subject of the remote installation platform was only surfaced in this research. Before the system needs to grow larger, the scalability capabilities, bottle neck components and the effects of adding specific resources should be researched in order to keep the system operational and responsive with a larger size. If one CFA is not performance wise sufficient for the platform in the future, a research should aim for distributing the CFA role and the backend resources into multiple environments, scaling the resources horizontally rather than vertically in order to aim to the long term scalability capabilities.

In the current design, the new environment must be prepared with the prerequisite software and configurations in order to install the EDMS platform software into it. In the increasing trend of virtualized hardware resources and moving toward cloud based environments, a promising future research topic is to automate the environment management as on-demand based that integrates to some of the widely adapted cloud services that provide hardware platforms and infrastructures as a service. Utilising the cloud services could decrease the need for self-maintained hardware in the CI system and testing purposes that covers the most of the remote installation platform usage cases.

Another future research topics to continue this research is optimizing and improving the current installation process workflow. As mentioned earlier in this thesis, the installation process workflow runs mainly in sequential execution that does not utilize the available CPU cores very efficiently in the target environment. The current trend in the energy business in Europe is to move the data balance settlement operations from individual energy companies towards centralized national operators, i.e. into data hubs. From the target EDMS platform software perspective, that means larger installations with similar domains for multiple energy companies. Based on the dependency analysis done as part of this research, such installations have huge optimization potential in the form of parallelization in the installation process. Therefore that topic is the most prominent optimization focus area identified by this research.

6 Bibliography

BAKER, F.J., 2009. *Core Protocols in the Internet Protocol Suite.* Internet Engineering Task Force.

BELLIFEMINE, F., POGGI, A. and RIMASSA, G., 2001. Developing multi-agent systems with a FIPA-compliant agent framework. *Software-Practice and Experience,* **31**(2), pp. 103-128.

CHOUDHARY, S.R. and JALOTE, P., 2014. Cross-platform testing and maintenance of web and mobile applications. New York NY: ACM, pp. 642-645.

DANEK, J., 2016. Finding optimal compatible set of software components using integer linear programming. Springer Berlin Heidelberg, pp. 457-468.

ERICKSON, J., 2008. Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality. *Journal of Database Management,* **19**(3), pp. 42-54.

FRENZEL, L., 2013-last update, What's The Difference Between The OSI Seven-Layer Network Model And TCP/IP? [Homepage of Electronic design], [Online]. Available: http://electronicdesign.com/what-s-difference-between/what-s-difference-between-osi-seven-layer-network-model-and-tcpip [04/13, 2016].

HARMENING, J. and WRIGHT, J., 2013. Virtual Private Networks. *Computer and Information Security, 2nd ed., JR Vacca, Ed., Waltham, Elsevier Inc,* pp. 855-867.

HAU, T., EBERT, N., HOCHSTEIN, A. and BRENNER, W., 2008. Where to start with SOA: criteria for selecting SOA projects, *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual* 2008, IEEE, pp. 314-314.

INTERNET ENGINEERING TASK FORCE, 2011. *RFC 6455. The WebSocket Protocol. Protocol specification.* Internet Engineering Task Force.

INTERNET ENGINEERING TASK FORCE, 2005. *RFC 4026. Provider Provisioned Virtual Private Network (VPN) Terminology.* Internet Engineering Task Force.

INTERNET ENGINEERING TASK FORCE, 1981. *RFC 793. Transmission Control Protocol. Protocol specification.* Internet Engineering Task Force.

JENNINGS, N.R., 1999. On agent-based software engineering. Artifical Intelligence. Southampton SO17 1BJ, UK: Department of Electronics and Computer Science, University of Southampton, pp. 277.

JEZEK, K. and AMBROZ, J., 2015. Detecting incompatibilities concealed in duplicated software libraries, *2015 41st Euromicro Conference on Software Engineering and Advanced Applications* 2015, IEEE, pp. 233-240.

KHUE, N.T.M., 2012. Developing an Intelligent Multi-Agent System based on JADE to solve problems automatically, *Systems and Informatics (ICSAI), 2012 International Conference on* 2012, IEEE, pp. 684-690.

KUSEK, M., VELÁSQUEZ, J.D., HOWLETT, R.J., JAIN, L.C., KANADE, T. and RÍOS, S.A., 2009. Functionality and performance issues in an agent-based software deployment framework. Springer Berlin Heidelberg, pp. 46-53.

MAO, H., ZHU, L. and QIN, H., 2012. A comparative research on SSL VPN and IPSec VPN, *2012 8th International Conference on Wireless Communications, Networking and Mobile Computing* 2012.

MICROSOFT, 2016-last update, The Component Object Model. COM Fundamentals. [Homepage of Microsoft], [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms694363(v=vs.85).aspx.

MISHRA, A. and SRIVASTAVA, V., 2014. Multi agent paradigm used to complexity measure for perfective software maintenance, *Computer Science and Engineering (APWC on CSE), 2014 Asia-Pacific World Congress on* 2014, IEEE, pp. 1-9.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2009. *Guidelines on Firewalls and Firewall Policy - Recommendations of the National Institute of Standards and Technology.* Special Publication 800-41. Revision 1. edn.

NWANA, H.S., 1996. Software agents: An overview. *Knowledge Engineering Review,* **11**(3), pp. 205-244.

PANZICA LA MANNA, V. and CRNKOVIC, I., 2011. Dynamic software update for component-based distributed systems. New York, NY: ACM, pp. 1-8.

POSLAD, S., 2007. *Specifying protocols for multi-agent systems interaction.* UNITED STATES: Association for Computing Machinery.

RANGANATH, V.-. and CRNKOVIC, I., 2014. Compatibility testing using patterns-based trace comparison. New York NY: ACM, pp. 469-478.

RAO, A.S. and GEORGEFF, M.P., 1995. BDI Agents: From Theory to Practice. *Proceedings of the First International Conference on Multiagent Systems.*

REGLI, W.C., 2014. Development and specification of a reference architecture for agent-based systems. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans,* **44**(2), pp. 146-161.

SITNIKOV, D., RABASA, A. and BREBBIA, C.A., 2013. A method for building desktop software automated update systems. SOUTHAMPTON: WIT Press, pp. 105-111.

SLACK TECHNOLOGIES, I., 2016-last update, Incoming Webhooks. Send data into Slack in real-time. [Homepage of Slack Technologies, Inc], [Online]. Available: https://api.slack.com/incoming-webhooks.

THE INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 1994. *Information processing systems - Open Systems Interconnection - Basic Reference Model.* Second edition edn. Switzerland: ISO/IEC.

THE POSTGRESQL GLOBAL DEVELOPMENT GROUP, 2016-last update, About PostgreSQL. Available: https://www.postgresql.org/about/.

VAN DER AALST, WIL MP, TER HOFSTEDE, A.H., KIEPUSZEWSKI, B. and BARROS, A.P., 2003. Workflow patterns. *Distributed and parallel databases,* **14**(1), pp. 5-51.

VAN DER AALST, W.M.P., 2003. Workflow patterns. *Distributed and Parallel Databases,* **14**(1), pp. 5-51.

WIELAND, A., 2012. Dealing with supply chain risks: Linking risk management practices and strategies to performance. *International Journal of Physical Distribution & Logistics Management,* **42**(10), pp. 887-905.

WOOLDRIDGE, M. and JENNINGS, N.R., 1995. Intelligent agents: theory and practice. *Knowledge Engineering Review,* **10**(2), pp. 115-152.

ZHANG, H.L., LEUNG, C.H. and RAIKUNDALIA, G.K., 2005. AOCD: A multi-agent based open architecture for decision support systems, *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)* 2005, IEEE, pp. 295-300.

ZHANG, H.L., 2006a. Classification of intelligent agent network topologies and a new topological description language for agent networks. *IFIP International Federation for Information Processing,* **228**, pp. 21-31.

ZHANG, H.L., 2006b. Matrix-agent framework: A virtual platform for multi-agents. *Journal of Systems Science and Systems Engineering,* **15**(4), pp. 436-456.

Appendix 1. List of installation plan tasks and sub-tasks.

Parent Task	Sub-Task	Type	Description
Restoration	Check input data for application files restoration	Validation	Check that all required input data for application files restoration was provided.
	Check input data for database restoration	Validation	Check that all required input data for database restoration was provided.
	Stop application daemons	Before-action	Stop relevant existing daemon services before application restoration operation.
	Stop database daemons	Before-action	Stop relevant existing daemon services before database restoration operation.
	Restore application files	Action	Restore application files to one or many locations from backups.
	Restore database	Action	Restore database from backup.
	Start application daemons	After-action	Start stopped application daemon services.
	Start database daemons	After-action	Start stopped database daemon services.
Installation package creation	Check network connectivity	Validation	Check that there is working connection with package source.
	Check input data	Validation	Check that all required input data was provided.
	Check free space	Validation	Check is there enough free space to create package.
	Resolve installation version range	Before-action	Resolve installation version range (current and target versions).
	Create package skeleton	Action	Create package directory structure, place installer software into it and download manifest files.
	Download installation files	Action	Download installation files from various data sources defined by package content manifests.
	Verify downloaded package	After-action	Verify downloaded package's validity and CRC checksum.

	Compress package	After-action	Compress package with file compressor.
File installation	Check input data	Validation	Check that all required input data for file backup was provided.
	Check free space	Validation	Check is there enough free space to complete file installation and backup.
	Stop application daemons	Before-action	Stop relevant existing daemon services before file installation operation.
	Take backups	Action	Take backups from application binaries and shared files.
	Install files	Action	Install application binaries and shared files to all target locations.
	COM-server registrations	Action	Register all application COM-servers from application binary directory.
	Install new daemons	Action	Install new daemon services.
	Start application daemons	After-action	Start stopped and installed application daemon services.
Database backup	Check input data	Validation	Check that all required input data for database backup was provided.
	Check database existence	Validation	Check that database files exists in the provided location.
	Check free space	Validation	Check that there is enough free space to create database backup copy.
	Stop database daemons	Before-action	Stop relevant existing daemon services before database restoration operation.
	Take backup	Action	Take database backup that can be restored if needed.
	Start database daemons	After-action	Start stopped database daemon services.
	Check installation package validity	Validation	Check that the installation package is valid.

Database installation	Check database Connection	Validation	Check that the database connection with specified connection details works.
	Check tools' operability	Validation	Check required command line tools' availability and operability.
	Check application instance state	Validation	Check current state of application instance and common database credentials for instance handling.
	Check application domain states	Validation	Check current state of application domains and domains' database credentials.
	Check external schemas for domains	Validation	Check current states for application external schemas and their database credentials.
	Run Database updates	Action	Run database schema creations, updates and migrations.
	Install new external schemas	Action	Install new external schemas for application domains.
	Import configurations	Action	Import data from input files as database objects.
After-installation operations	Delete old file backups	Action	Delete old backups made in file installation task based on expiration rules.
	Delete old database backups	Action	Delete old backups made in file database task based on expiration rules.

Appendix 2. Example installation plan in serialized format


```
{
    "Enabled": true,
    "Identifier": "Plan:INSTPLAN",
    "Parameters": {},
    "Name": "ACME Environment installation",
    "Uuid": "4a1d06a5-a4c9-4967-dd44-9d1242c61159",
    "Tasks": [
        {
          "Identifier": "Task:FILEINST",
          "Enabled": true,
          "SubTasks": [<details omitted>]
        },
        {
            "Enabled": true,
            "Identifier": "Task:DBINST",
            "SubTasks": [
                                {
                    "Enabled": false,
                    "Identifier": "Subtask:DBINST/ACTION/DOMAIN/INSTALL",
                    "Parameters": {
                        "DomainNameList": "DOM1|DOM2",
                        "DatabaseRootDir": "d:\\Oracle\\Database\\EDMS",
                        "ContinueOnSqlErrors": "True",
                        "CredList": "<input omitted>"
                    }
                },
                {
                  "Enabled": true,
                  "Identifier": "Subtask:DBINST/ACTION/CONFIGURATIONS/IMPORT",
                  "Parameters": {
                    "DomainNameList": "DOM1|DOM2",
                    "DatabaseRootDir": "d:\\Oracle\\Database\\EDMS",
                    "ContinueOnSqlErrors": "True",
                    "ContinueOnRegAppErrors": "True",
                    "CredList": "<input omitted>"
                  }
                },
                {
                    "Enabled": true,
                    "Identifier": "Subtask:DBINST/ACTION/EXTSCHEMA/INSTALL",
                    "Parameters": {
                        "DomainName": "DOM1",
                        "ExternalSchemaNameList": "<ext.schma1>|<ext.schma2>",
                        "SkipInstallationIfSchemaExists": "True",
                        "ContinueOnSqlErrors": "True",
                        "ContinueOnRegAppErrors": "True",
                        "CredList": "<input omitted>"
                    }
                },
                {
                    "Enabled": true,
                    "Identifier": "Subtask:DBINST/ACTION/EXTSCHEMA/INSTALL",
                    "Parameters": {
                        "DomainName": "DOM2",
                        "ExternalSchemaNameList": "<ext.schma1>|<ext.schma2>",
                        "SkipInstallationIfSchemaExists": "True",
                        "ContinueOnSqlErrors": "True",
                        "ContinueOnRegAppErrors": "True"
                    }
                }
            ]
        }
    ]
}
```

Appendix 3. Example installation subscription in serialized format


```
{
  "InstallationPlanName": "ACME Environment installation",
  "InstallationPlanUuid": "4a1d06a5-a4c9-4967-dd44-9d1242c61159"
  "DataSource": "EDMS",
  "MaxVersion": "2.15.3.99",
  "CustomerPackageName": "ACME System",
  "BranchName": "V2.15 R3",
  "UpdateOverBaseBranchVersion": false,
  "ReportRecipients": ["mr.head@company.com"],
  "SlackReportRecipientWebHooks":
["https://hooks.slack.com/services/tsttst/tsst/test"],
  "InstallationScopes": ["Default", "ProdEnv"],
  "Enabled": true
}
```

Appendix 4. List of all requirements and the solutions presented by the thesis.

#	Requirement	Solution
1	Remote installation platform should have means to describe intentions and transform them into executable actions.	The solution presents *installation plan* and *installation subscription* data structures that contains installation process and its input that a DIA can use to execute an installation.
2	The intentions should be transferrable through the network passing all its traffic regulator mechanisms, e.g. firewalls.	The data structures defining an installation can be serialized and deserialized and transferred through HTTP compatible protocols that has same connection requirements as any web-technology based applications for the end-to-end connection.
3	The remote installation platform should have minimal integration or requirements for allowing its traffic.	Only standard two-way TCP traffic is used in communication between two agents. When used as *mobile* agent the DIA that is placed to a restricted environment does not require connection to any other agents. In other cases the DIA requires connection only to a single CFA.
4	The intentions should be executable by in the target environment.	The DIA that is placed to a target environment and operates as a proxy for the installation process requires administrative privileges to the target systems that allows it to alter the environment and install new application components.
5	The actuator that executes an intention should be able to provide feedback to the party that creates the intention about the execution.	The DIA running an installation provides constant feedback information that is visible in the management interface. The status information is passed also to the CFA.
6	The messaging protocol should include acknowledgements and detection for non-delivered messages.	The acknowledgement and delivery status is achieved by using a standard TCP protocol as well as JSON response messaging format in all the communication between agents that provides the information.
7	The system's own software's upgrade process should take minimal time and cause minimal disruption to the service it provides.	The proposition suggest to limit single update process to take less than 30 seconds for any agent part of the platform that is in the boundary of retrying any failed connection attempt.
8	A remote installation system should have a capability to update its software without causing failures on ongoing operations.	The proposition has a signalling procedure that allows upgrading the software only when the agent is in idle state and not executing any installation tasks.

9	A software update procedure should be fault tolerant and able to restore operability on unexpected errors.	The updater component takes backup of any working software before trying updating and has a restoration process in case a new version is not starting without errors. In addition the agent keeps track of any failed installations and does not try to re-install any versions that have been previously detected as non-working.
10	Selected 3rd party software components should be sufficient for the purpose as well as compatible with each other and the target software.	All the 3rd party libraries and components were assessed carefully and selected only from mature and live projects. They were tested for compatibility before including as part of the distribution.
11	The list of dependencies and their compatible versions expected to be in the target system should be known and available for the user.	As a side product for the software implementation, a reference document with all the prerequisites were created that was reviewed and tested in an actual fresh installation by a partner company.
12	It should be possible to pre-define an installation process in order to automate executing it.	(Same as solution for Req. 1)
13	It should be possible to execute installation with input that applies the installation to a specific target system.	The *installation plan* data structure contains all the input not available in the target environment that is required by the installation process. Several hundred installations were executed by using the implemented remote installation platform where some of them were successfully executed and verified and others failed on errors in the installation package, not in the remote installation platform itself. The installations included an *offline environment* in real customer's premises with actual installation procedure for real use.
14	Installation input data format should support serialization and deserialization.	All the installation data structures were implemented to be transformed to and from the serializable JSON format without losing any data in the transformation process. All the input parameters designed to have a string-encoded format that was used in serialization.
15	Updating the network platform software should not cause any nodes to become incompatible and not being able to become compatible.	The transferrable data format serialization is designed to be backward compatible in a way that an agent running an old version can deserialize data formats with new parameters without extra effort. The agents have automated update procedure component that handles keeping the agent software up-to-date. New version check is triggered periodically and whenever agent suspects that it needs to update

		itself. In addition, the CFA can force all agents to update themselves when they become idle if needed.
16	A multi-agent platform should have at least three roles: Agent Management System, Agent Communication Channel and Directory Facilitator.	The CFA is designed to manage all these roles and provide the services to the other agents in the platform.
17	A multi-agent platform should enable agents to be able to communicate, discover and invoke services from each other as well as from some non-agent software services when the agent cannot complete a task individually.	The required functionality is implemented by exposing set of REST APIs to be used by other agents. The CFA provides agents all the information and capabilities they don't possess by themselves to complete their tasks.
18	A multi-agent platform architecture should divide the agent design into independent components that have clear responsibilities to manage the functional requirements set for the system and are easy to modify or replace.	All agent implementations are designed based on a clear component models where the components manage one or many functional requirements.
19	An agent network should have a messaging protocol implementation that supports passing commands and information from one party to another.	All the messaging between agents are built on top of TCP protocol by utilizing RESTful APIs and HTTP protocol. The messages are JSON-formatted that are compatible with all the agents.
20	The agent's software should be updatable in order to correct software faults and adapt to the changed requirements.	The CFA distributes updates to agents per requests from an agent software repository where new software versions are placed.
21	An agent should have ability to update its own software automatically when the agent software distributor has a new version available.	Agents run periodic task to check if new version is available and handles downloading and installing it as well as recovering from any non-working software update attempt.
22	An agent should have the ability to recover from problems like system restart, offline time and temporary network connectivity issues.	DIA*s* are installed to the environment as daemons that starts automatically when system starts. Any installation that was interrupted by a system shutdown or restart is considered as installation failed for environmental reasons that will require an administrator actions. Any connection failure is followed by number of retries (five times with small delays) before accepting that there is no connectivity currently available to the target agent. The DIA*s* keeps track of sent reports and provides ability for the CFA to query the information later.
23	An agent node should be self-maintainable and hardly ever require manual maintenance done by human administrator.	Agent instances should require maintenance only if the software is corrupted from some unknown reason. The installation process may require help if something unexpected problems occurs. The target

		environments where agents are running may require human actions if some resources (i.e. disk space) are getting low or need replacing.
24	An agent should not require more resources that are realistically available in any target environment it is installed into.	The DIA comes with a set of prerequisites including hardware resources. It does not require significant resources apart from some disk space but more available resources may speed up the operations. The CFA should be placed to a more (but not unreasonable) powerful server to avoid causing any bottle necks for the platform operability.
25	An agent implementation should consider the balance between available network bandwidth and computing resources.	The implementation uses 7z compression for large data sets sent over a network to save bandwidth. The solution requires using additional computing resources and/or more time on the target site but it speeds up the overall platform performance and speed since network resources are in general scarcer than basic computing resources.
26	Sensitive data transmission should be avoided through open network or the data should be protected by appropriate measures.	From customers' point of view, password data is the only sensitive information identified to be sent through network in the proposed solution. A suggestion for future research is made to improve the protection for password and credentials management throughout the entire platform.
27	An agent should have a way to verify authenticity of the other party it is communicating with.	The proposed solution uses unique id's for each agents and a single CFA with known, predefined address. The solution should be improved with HTTPS and other more secure authentication methods, but the security related subjects were left for future research.
28	To improve efficiency the input data format should prefer machine-readability whenever it is not visible for human users.	The internal data formats use relational and serialized formats and the data representation is only transformed to a more human readable formats in the user interface components and not before.
29	When the input data is transformed from one form to another, no information should be lost in the process.	All the data conversions are designed to be lossless.
30	The input data format should be able to contain an installation process control flow for all the relevant installation cases.	The *installation plan* format can contain all the actions in task and sub-task level that can be part of the installation process. It defines the installation process control flow that is used in an installation.

31	Agents running in one customer's environment cannot be directly connected to other agents running in other customers' environments.	The network topology selected to the design implements a model where DIA*s* that are placed into a customer environment connects to the platform only through the CFA that operates in the software vendor's premises.
32	Having a strict installation window the agent network should have a level of redundancy of multiple agents being capable of completing the same goal.	In the proposed design a single target environment can have only one installation operation ongoing at a time and there should be always agent free to execute the installation to that environment whenever the environment is free from any other installations. The solution does not count the less than 30 second window when an agent may be self-updating itself and therefore unable to execute an installation.
33	The solution should use 3rd party software components when there are good ones available, but the decision of using each component should be evaluated and though carefully.	As part of the implementation, many 3rd party libraries were included after careful consideration to manage e.g. data sources, connectivity through standard protocols and user interface functionality that supports variety of web browsers. There were some general agent platform frameworks available but utilizing them were considered to be too risky from the operability and long term manageability perspective and therefore the platform itself was self-implemented.
34	Selected 3rd party components should be compatible to the maintenance strategy thought for the platform and not cause severe problems or offline time.	The selected 3rd party components were tested to work in the target operating systems and with the target web browsers. During the implementation the agent software was updated several dozen times and the update operation was always evaluated to take less than 30 seconds that was decided to be the acceptable upper boundary.

Appendix 5. Executed performance test scenario setup with Apache JMeter application.

Test description:
The test's purpose is to test performance of the data provider API that is used by all radiators that shows ongoing progress status of installation operations that are part of Continuous Integration system. The provided data is collected from several back end systems and is expected to become one of the firstly detected bottle necks from the remote installation platform.

The performance test is executed with popular performance testing software Apache Jmeter with three slave computers that generates load (HTTP protocol based REST api queries) simulating ~10 000 concurrent users that queries the data once. The maximum load is achieved within 60 seconds ramp up period resulting 10 000 queries within one minute period.

The test measures succeeded and failed connection attempts. The connection and query response must be met within the test parameter timeout values (20 and 30 seconds). The expected data format is XML that is asserted after each request.

Test parameters:

Parameter	Value
Number of hosts (JMeter slaves) included in test exec.	3
Simulated concurrent users (per host)	3334
Full users rampup period, seconds	30
Path execution loop count (per user)	1
Max. samples per user per minute	60
Connection timeout, milliseconds	20 000
Response timeout, milliseconds	30 000

Apache JMeter test plan: