TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatika instituut
Tarkvaratehnika õppetool

Tarkvara modulaarse laiendamise võimalus CMS-i näitel

Bakalaurusetöö

Üliõpilane:	Stanislav Babuškin
Üliõpilaskood:	103716IAPB
Juhendaja:	Kaarel Allik

Tallinn
2015

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

....

(*kuupäev*) (*allkiri*)

Annotatsioon

Käesoleva töö eesmärk on välja töötada tarkvara arhitektuur, mis võimaldab täiendada süsteemi funktsionaalsust, kasutades dünaamiliselt lisatud moodulaarsid laiendusi. Jargnevalt kirjeldatakse vaadeldava arhitektuuri disainiga seotud probleeme ja nende lahendusi keskse haldussüsteemi näitel. Kasutades kirjeldatud lahendust, on võimalik uuendada ja muuta tarkvarakomponentide funktsionaalsust ilma masina taaskäivituse ja programmi peatamiseta. Näidisprogrammi arenduskeeleks on cSharp. Süsteem koosneb kahest osast: keskserverist (Server) ja mitmest agendist (Agent). Keskserver saab kasutaja käest ülesanded, koostab klientprogrammile mõistetava käsustiku ja saadab selle klientarvutitele, kus töötab agenditarkvara. Süsteemi funktsionaalsus jaotatakse kirjeldatud laienduste vahel ära. Iga laienduse moodul paigutatakse süsteemi erinevasse osasse ning seejärel algatavad moodulid omavahelise suhtluse. Süsteemi tuuma (core) peamine ülesanne on laienduste haldamine ja sõnumite edastamine läbi arvutivõrgu.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 43 leheküljel, 5 peatükki, 25 joonist.

Software Modular Extensibility On A Central Management System Example

Abstract

The purpose of this thesis is to design software architecture for Central Management System that allows to extend functionality of the system using dynamically added modules. In additional to describe the problems associated with the architecture and their solutions. Solution should be able to update and change functionality of agent extensions without reinstalling agent program. Application will be written in CSharp language and consist of two components Central Server and Agent. Central server receive tasks from user, converts to understandable instruction set and sends it to the client agent. The functionality of the system is stored in extensions. Each of the extension is placed on system both components and communicate with each other. The system core main functionality is extension management and providing communication between modules of extension.

This thesis is in English and contains 43 pages of text, 5 chapters, 25 figures.

Glossary of terms and abbreviations

CMS	Central Management System
MEF	Managed Extensibility Framework
xml	Extensible Markup Language
Base64	Binary-to-text encoding schema
IIS	Internet Information Services
WCF	Windows Communication Foundation
CBSE	Component Based Software Engineering
MS SCCM	Microsoft System Center Configuration Manager
OCSI NG	Open Computers and Software Inventory Next Generation
NTFS	New Technology File System
OSI	Open Systems Interconnection

Table of Contents

Table of Figures

1 Introduction

For user computer is a set of applications. One of the main tasks for user is application management – installing, deleting and updating a software elements. Administration is not a critical issue for a one workstation, but at organization level many stations need to be controlled - what becomes a bigger issue. In most companies the users right must be limited to prevent property damage or for security reasons. Each computer can be configured separately but management of computers configuration is still an issue. A Solution that allows to centralize management of machines is called Central Management System.

There are few popular implementations of CMS systems like Microsoft System Center Configuration Manager (MS SCCM) or Open Computers and Software Inventory Next Generation (OCS Inventory NG).

In perfect static world CMS resolution will be enough, but environment is developing and things change all the time so do software also. As a consequence an application meant to keep external components up to date need to be updated too. Even more in case where part of workstations are inaccessible most the time.

In this work will be considered a software architecture that allows to dynamically add, replace and remove functional components. Some problems will be highlighted on code level with snippets. This method should make possible to change system modules at the run time, without rebooting workstations and restarting application components.

The method itself construction software based on extensive modules or plugins is not new it is called Component Based Software Engineering, the major difference of project application architecture is that application exclude all functionality except plugin management core. In some cases additional functions can be added. Because of prototype components cores located on a different workstations, the communication functionality need to be added to cores.

In scope of this work will be considered an application server and client core developing and communication channel creation through the network using web services. Creating a few extensions with different functionality, work of each extension will be demonstrated and service layers on agent and server side. Work out message traveling method through the system from client side extension to the server side extension and vice versa. As an important issue the stability of sample CMS system, error handling and system stability will be highlighted in a separate chapter.

Application requirements:

- Application must be separated into the individual parts being able to communicate with each other.
- Ability to update application components according to needs and technologies.
- Each system component should be able to work independent and accumulate data in case of connection lost.
- System should be stable to errors.
- Core should be able only to manage plugins.
- Application cores should be able interact with each other and transfer messages.

Because prototype system contains of many projects and classes then none of its parts will be presented entirely including class descriptive diagrams. In each chapter will be highlighted only part directly connected to topic.

Purpose of this thesis is to workout enterprise application architecture solution that allows dynamically add functionality to System without reinstalling or rebooting its components and application as a system. System security is out of project scope. The work does not include graphical interface creation and optimization, only applying principles of current architecture will be highlighted.

This work is divided into 5 chapters, each one which contains subtopics.

The first chapter will give a short overview of used technologies and platforms and domain knowledge of sample system. The second chapter is about system components, application architecture and its layers description. Putting together system components, communication within application and between system components will be described in the third chapter. The fourth chapter is sample of usage methods and solutions explained in previous chapters. Last chapter is a short overview of system stability, diagnostics and testing.

2 Theoretical base

At some point of working as a developer in company producing more than one product I came to realization of software updating problem. Due to company size we didn’t had a special helping software for delivering a builds to client. The small prototype were done in small period of time and demonstrated to customer. One of requirement was to make a tool as simple as it possible from the inside to avoid bug fixes and updates. Despite a fact that client declined I continued working on a project in educational purposes. One issue stuck in my head – how to update software that is meant to update software without going to maintenance or stopping the production facilities. The forward work is my thoughts and an attempt to solve facing issue.

2.1 Environment

A Central Management System prototype will be implemented on Windows and Windows Server[1] operating system and a .NET platform[2] for a software development. As a programming language will be used one of the .NET supported languages – CSharp. For creating an application is used development environment provided by Microsoft - Visual Studio. Any data storage that application use will be created using SQL Server Management Studio and stored in SQL Server[3]. The application itself works on Internet Information Services[4].

Files stored outside of database represents an unstructured data. The using of outside unstructured data can cause management complexities in other case when data is stored directly to database the performance of server can be lowered due to bid files streaming. Solution provided by SQL Server called FILESTREAM is integration of NTFS[5] system to database engine. This solution is taking advantages of both previses cases, can provide additional security to database server and stored data. Extension distribution files will be stored in SQL Server using file stream feature.

A .NET platform supply good support for component based software engineering by providing ability to dynamically adding modules to running applications. One of platform frameworks which makes it easier is Managed Extensibility Framework[6].All project modules are built using MEF.

All these components are provided by Microsoft. On one hand it is easier to create a more stable system because components are meant to work together and proper error handling and debugging in development process. In other hand using special features of each component makes harder a multiplatform implementation. The implementation of system features might differ from original due to platforms architecture.

Initially a .NET were created for a Windows, but not so long ago Microsoft announced about launching its .NET distribution for Linux and Mac OS. Which makes possible of using initial project implementation on different platforms with only slight changes.

2.2 Terms

Further in this work will be used many terms related to system or application break down to its components. Usage of those term is different perspectives and architectonical scales might be confusing, therefore there is a need in rigorous definition of each of them. At first the common definition will be given and then applied to project architecture.

Most definitions are from book Component Based Software Engineering: Putting the Pieces Together[7] by Bill Councill and George T. Heineman. Table of used terms:

Terms	
Software element	A sequence of abstract program statements that describe computations to be performed by a machine.
Software component	Software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
Component Model	A model of specific interaction and composition standards. A component model defines specific interaction and composition.
Component Infrastructure	Set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.
Interaction	An action between two or more software elements.
Composition	The combination of two or more software components yielding a new component behavior at a different level of abstraction.

Figure 1 Common term definitions

As is shown on a Figure 2 System consist of handled workstation, in software perspective the root object in a break down is an Application which consist of individual self-sustained software elements that are working on a separate workstations and interact with each other – Agent or Central Server.

Next level of breakdown of elements are Software Components – On project prototype sample those are cores and functional modules of extensions. Because of Extension is basically a three modules that meant to be installed on a different instances of application, placed on same level of modules hierarchy located on opposite edges of application and communicate with each other. Two interactive modules of same level is called mirror modules.

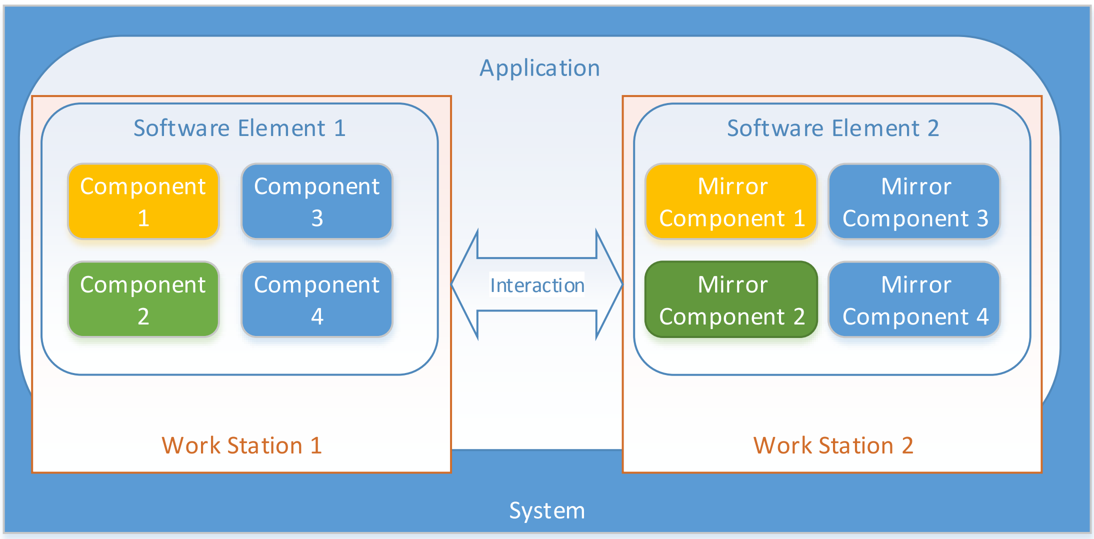


Figure 2 System Composition Model. Elements related to hardware is marked by shape sharp edges, software elements are marked by round edges.

3 System architecture

In this chapter the main topic is application architecture, main elements and used components description. As a beginning the highest-level breakdown of a system into its parts, software components breaking apart into the layers and solution applied to each layer. The design and functionality of system elements. The main effort is to design such architecture where dynamic extension and interaction between newly added modules is possible.

Architecture will be designed to support dynamically added components with business logic to application elements. Static component will manage only extension components.

3.1 Application architecture design.

In general the project have a Client-Server architecture. There is two software elements: Central Management System Central Server or just a server and an agent machines - client which communicate with server through a network.

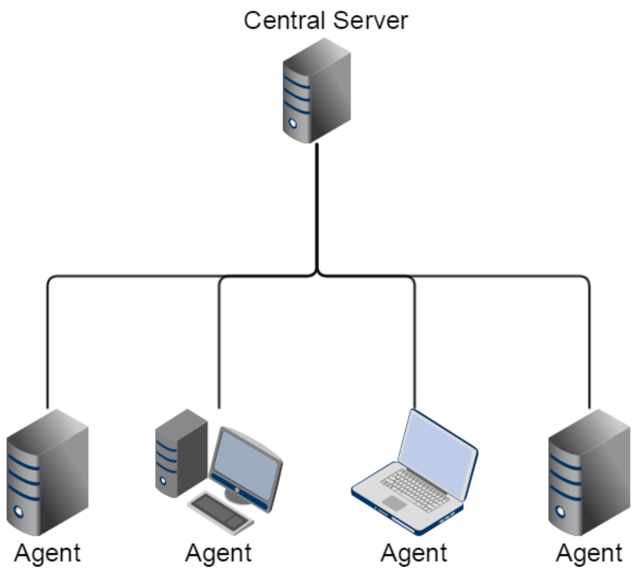


Figure 3 System Architecture

Application have a hierarchical structure in other words there is one primary element – Central Management System Central Server which forward will have a codename – Octopus, and a secondary elements installed on a managed workstations Central Management System Agent forward will be referred using a „Tentacle“.

A main task for an octopus will be handling messages from tentacles and a receiving tasks from a user. For a tentacle major job will be a processing octopus messages, execute instructions from messages and sending a report.

3.1.1 Common Design

Each application element has a similar structure: The static component which has limited functionality – the core and a dynamic component which have a common interface and different purpose - functional modules.

- Core – Unchangeable component with limited functionality. For avoiding changes in core all business logic must be placed in child modules. Core is a part of a service layer it provides or consumes services. Minimum required functionality for this part is:
    - Child components management
    - Communication with opposite side
    - Error handling

- Functional Module – Using common interfaces for communicating with the core or child modules it implements business logic according to its needs. Can be dynamically added or removed from parent module. A module functionality can be reduced to message forwarding and child modules management. In this case module will have a proxy role. Another role that module can have is an executive. This kind of module usually don’t have any child components and only executes instructions in receiving messages.

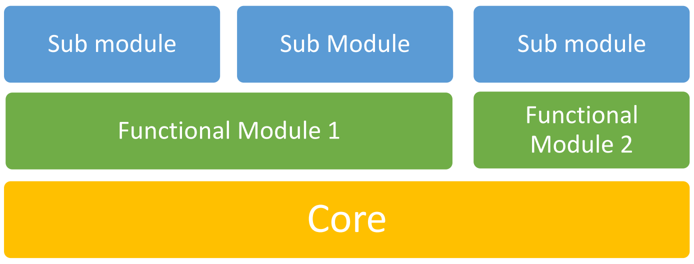


Figure 4 Software element common architecture

3.1.2 System components

The reason why CMS like system was chosen by sample prototype is because of its complexity. To demonstrate potential and adaptability of fully component based enterprise applications. Before making decision how to build each of components there is a need to take a closer look to each of them.

Sample CMS consist of two main components. Each component have his own purpose. In next chapter will be given a short overview of each of them, the running environment and tasks.

Central Server

Central Server is a workstation with Octopus web service application hosted on Internet Information Services Server. Central server is a core of all system. All workflow are controlled by the application control part. Octopus is presented by two parts: the processing element and visual management tool.

The processing element core component itself is basically a service layer[8] which has a set of available operations listed in service endpoint to communicate with tentacles. The endpoint is implemented using Windows Communication Foundation[9] library presented in .NET platform.

User interface have a tools set to give instruction to central server and can be implemented in two ways. One is a Web application hosted at the same server. In this case extension components are located on same workstation but been used with different software element. The second way is to create a desktop application. In this way components with user controls need to be transferred to the same workstation.

In scope of this work one of main tasks for central server is runtime component distribution. All extension components are stored on central server storage for distribution. It can be stored on file system or database. Central server tasks can vary depending on the implementation.

Client

Agent is a software element running on background of handled workstation as a windows service that reacts on changes in environment, making reports for a central server and listens to its instructions.

Due to Windows OS platform peculiarity the agent will be divided in a two parts: the mandatory and optional part. For each end user there is a separate session. A services can run on a local system or in user session, the key difference between them is a services working in user session stops with user logout. For this reason a mandatory part – agent core should be placed in local system to be able to control user sessions. In other hand the local system service is unaware of user session environment running processes and events. The optional part goes to user session in case of need of controlling it.

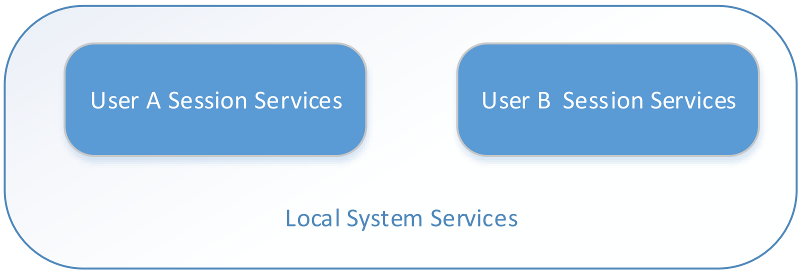


Figure 5 Windows User Sessions

The optional part is implemented and fully controlled by functional modules. In fact that not all functional modules need a presence in a user session this part is optional.

Agent core tasks are:

- Interaction with central server.
- Components management.

Client have a two working states: normal and accumulative state. In normal state the connection with central server is working and messages are not delayed. In case of connection lost the messages are stored locally to be sent later. This functionality is missing on client core and can be placed there because of core limited functionality. By following architecture design the additional functionality must be placed to separate extension. A new module will implement messages storing functionality and need to be placed as a root component.

3.2 Extension

Extension is an application component which can be dynamically added or removed by application core. This component contains application custom functionality. The component itself represents a set of class libraries or modules. Modules are created using Managed Extensibility Framework - the library that allows to create extensible applications by developing user extensions which can be plugged into core.

In development process an extension is composed from three parts or plugins. Each part represents a code library which is handled by cores on the workstations. Extension have a list of available operations or methods listed in extension common interface. Important point that need to be concerned while creating an extension is that each logical operation of extension need to be divided into the three parts and each of this parts located on a different system components.

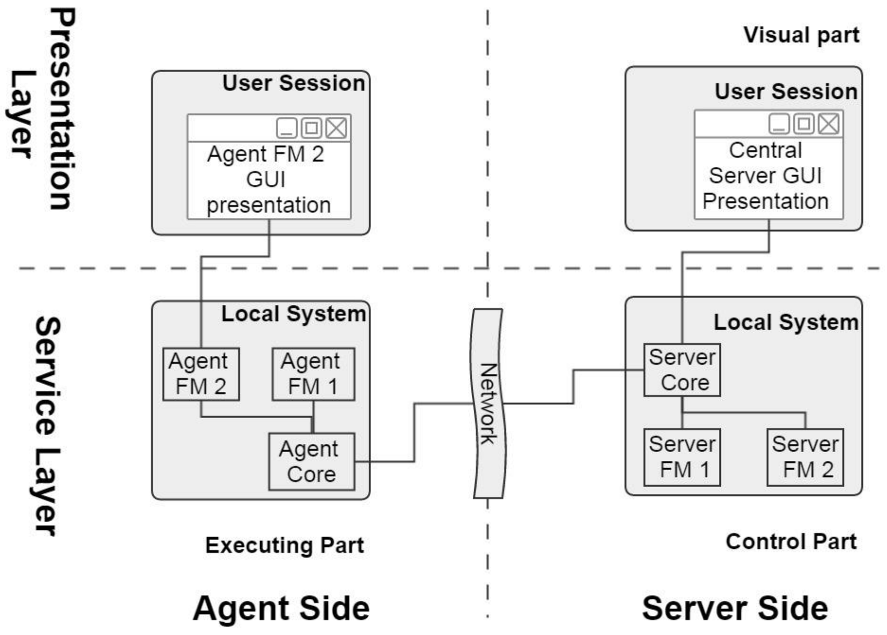


Figure 6 Extension distribution

Control part is a class library where the biggest part of business logic is stored. A result of work in this part is a saved state of modules, task management and a string instruction that will be sent to agent. In addition can have a functionality of:

- Operation result handling.
- Database interaction.
- Task distribution among the agents.
- Additional business logic can be also placed here.

Executing part have an executive part of operation. Executive module is located on an agent workstation and docked to its core. This module contains the implementation of work part of method. It might have a querying nature, or configuration setting change.

User extension located on administrator work station and docked into the user interface core is a visual part of extension. The working process is quite similar to agent - it uses same communication endpoint and works on user computer. It has a two fundamental differences:

- The core of visual part works as a desktop application or a web application.
- Composes instructions for central server.

In this part operation is implemented to compose an understandable string command list for a control part. In other words a list of methods and parameters that need to be called on control part. Visual part have a custom control which is presented to end user as a tool for creating tasks.

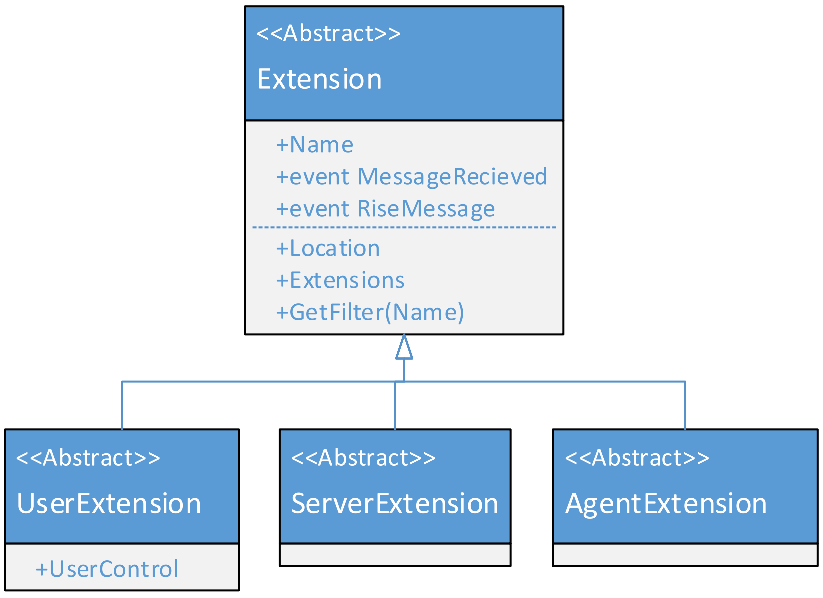


Figure 7 Extension Class Diagram

All part of extension have a similar composition. There is a Loader class which implements one of module extension: Server Extension, Agent Extension and a User Extension abstract classes. Loader class is used as a proxy for messages instructions. It reads command from task and sends to operation processor class which in addition to module operation set have a command decoder method called *MethodEntry*. Decoded method is used for selecting a right method from operations set, setting parameters and calling it. When task is processed a loader class notify about task completion by creating a response message.

On a system edge all extension components have a hierarchical structure, to module it means that it can have a child modules stored in *Extension* property. Each of them have a predefined *Name* property used for addressing messages. Parent module can access name by using getter method *GetFilter*. *Location* is a physical path to extension directory.

All modules can work as an individual software element: it has an entry method - *MessageRecieved* (similar to program “main” method), parameters can be passed in message and a result is returned by rising result message.

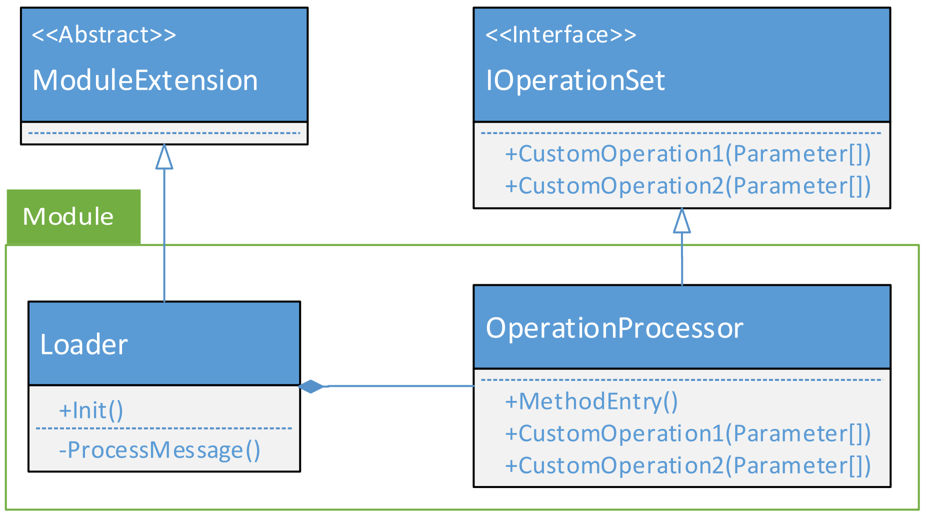


Figure 8 Module Class Diagram

In scale of operation the visual part and control part have an accessory facilities. A main task for these two parts is to deliver a task from user to agent.

Each module contains a class what implements operation set interface. For this reason Control Part and the Executing Part have same method sets, the difference between them that in Control part each method construct scripts for executing part mirror method is placed where all business logic for certain operations. By the signature of control part it is a mirror functional module to control part.

Visual part of extension is a separate software component or library. In addition to certain extension method it has an additional one for returning a user control.

3.3 Extension Management

As long as project has a component based architecture the component management issue is important. Extension management is a loading functionality from modules to main program and handling code libraries and within a software element.

3.3.1 Initial loading

Application software element consist of two parts: a static part and a dynamic part. The relatability of first part will be discussed in a diagnostic chapter. This subtopic is about initial loading mechanism of dynamic software components.

At the beginning of work program have only working core and file system which need to loaded to execution library. The components are loaded recursively, which means that each component are loading only child components and triggers child component initiation. The first place where initialization starts is core:

```
    public class ExtensionManager{
        [ImportMany]
        public IEnumerable<AgentExtension> Extensions { get; set; }

        public ExtensionManager(String ExtensionsDirectory){
            InitExtensions ExtensionsDirectory);
        }

        public void InitExtensions(String extensionDir){
            var extensionDirs = Directory.GetDirectories(extensionDir);
            var catalog = new AggregateCatalog();
            foreach (var dir in extensionDirs)

                catalog.Catalogs.Add(new DirectoryCatalog(dir));
            var container = new CompositionContainer(catalog);
            container.ComposeParts(this);
        }
    }
```

This method set is quite similar to extension interface loading mechanism. The *AggregateCatalog*, *DirectoryCatalog* and *CompositionContainer* classes re part of .NET MEF library. A *DirectoryCatalog* is a folder with extension module which has an implementation of *AgentExtension* implementation class. The implementation object – module is loaded to extension list. An *AggregateCatalog* is a helper class which is needed when modules are distributed to different directories.

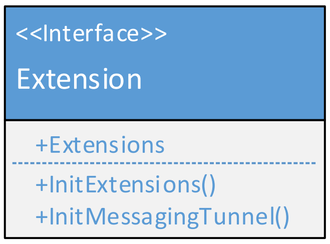


Figure 9 Extension Class Diagram. Initialization mechanism.

All extensions have an initialization mechanism with default implementation similar to core. In addition it has a message channel initialization method to concatenate communication events:

```
    public abstract class Extension{

        [ImportMany]
        public virtual IEnumerable<Extension> Extensions { get; set; }

        ...

        public virtual void InitMessagingTunnel(){
            foreach (var extension in Extensions)
                extension.OnMessage += (sender, args) =>
                    SendMessageToParent(args.Message.WrapMessage(this));
        }

    }
```

3.3.2 Runtime management

Previous chapter was about extension loading but initial structure was constant. How to manage extension in runtime is a topic for this chapter. The main idea of this work in scope of component based software engineering is how add, delete or update components without stopping program workflow.

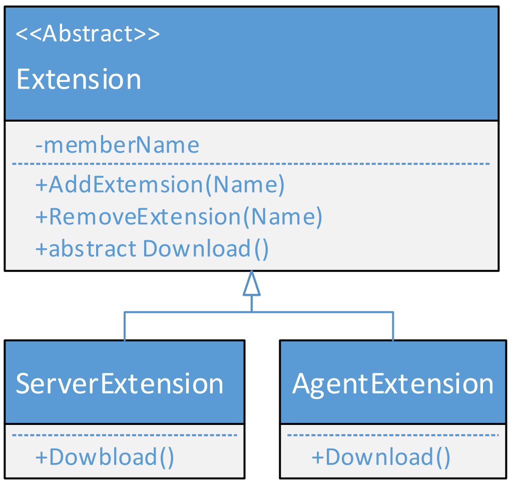


Figure 10 Extension Class Diagram. Components Management mechanism

A method Download has default implementation on module level extension interfaces. Operation divided according to system roles. General purpose of the method is transmitting physical files. The implementation of this method may vary and not will be demonstrated in this project.

To make changes in composition possible there is a need in two operations implementation: the physical file system management and a component model management.

Once the main object where extension modules is being composed the forward elements addition or deletion is going through the same way as on initialization

3.4 Message

In this chapter we will look at message as an object and its structure and message properties purposes. The message states while traveling through the system.

3.4.1 Message Structure

The structure of message is the same for each layer of application. The message is an object that is being constructed in functional module and meant to the mirror functional module on the other side. Message has next fields:

Action – An action that need to be done with message. Possible values are forward/apply. Value forward means that current functional module is not a destination module and message need to be sent to parent or child module, depends on message direction. Apply value means that module reached end point and instructions need to be applied.

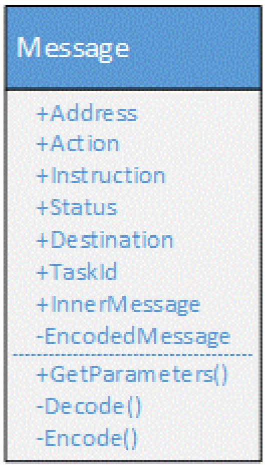


Figure 11 Message Class Diagram

- Address – In case if action is equals to forward then we need to know the next module name where to send.
- Destination – Workstation name. Used At core level. Duplicated at all message levels.
- Task ID – When message is constructed and need to be confirmed the functional module store message to response task queue and awaiting response message with same id.
- Instruction – a set of commands or script for mirror functional module.
- Status – When message is processed by mirror functional module it change status and send back acknowledge message with script result.
- Encoded Message And Inner Message – Encapsulated message for the next layer.

Encoded message is a text representation in base64 format message used at the network layer. As a simple example representation message in string. Can be used for security purposes and coded with secret key.

3.4.2 Message Serialization

On the way through the system message transforms from object to xml state.

For convert messages between two states is used .NET *System.Runtime.Serialization* class library. Using special attributes from library it is easy to define xml structure right in CSharp class. Thereby by using serialized data objects in services there is no need in creating envelope xml messages to send over the network.

```
[DataContract]
public class Message{
        [DataMember]
        public String Address;
        [DataMember]
        public String Destination;
     ...
}
```

4 Communication System

Modules of system can have various amount of functionality and operations. In case if two modules were designed to work together directly there is no need in complex proxy system of method calls through limited communication interfaces. This chapter contains a description of solution to communication problem – the way how two modules located on a different system edges can cooperate using standard message interface.

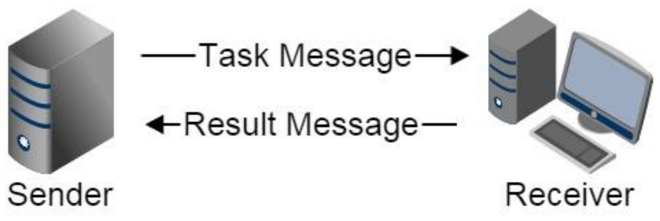


Figure 12 Communication model.

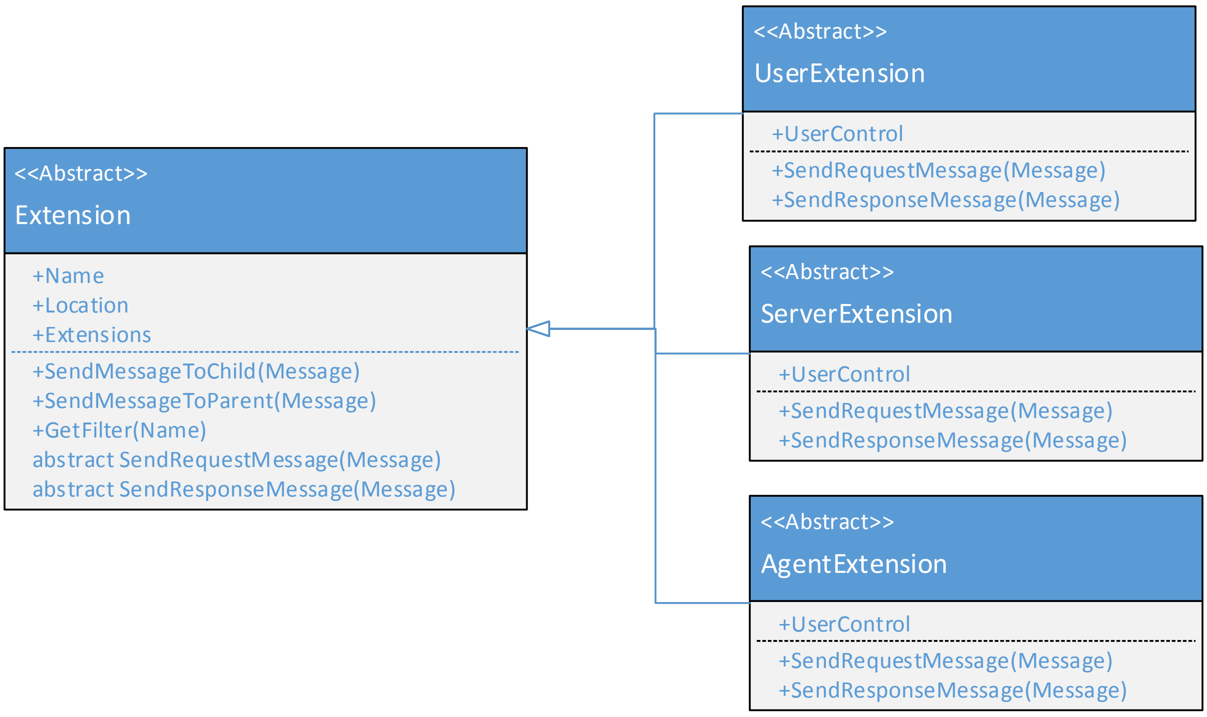
In communication model system have two components: Sender and receiver. They are using a message as a communication object. The Central Server and Agent both can have a Sender and Reviver roles. The Sender role generates task and send message to receiver. At receiver side task is being processed and a result is sent back to sender.

4.1 Messaging Forwarding

In applications built as a final application all operations can be called using standard method call procedures and communication channels have a specific structure however in system designed to be able add and remove senders and receivers the major question is how to find destination module in dynamic environment.

Message is moving through application layers or modules using functional module base functionality implemented in abstract class.

The operation *SendRequestMessage* and *SendResponseMessage* have a functional default implementation on a module extension because of different forwarding methods. The difference will be described in next topic.



<<Abstract>>
Extension
+Name
+Location
+Extensions
+SendMessageToChild(Message)
+SendMessageToParent(Message)
+GetFilter(Name)
abstract SendRequestMessage(Message)
abstract SendResponseMessage(Message)
<<Abstract>>
UserExtension
+UserControl
+SendRequestMessage(Message)
+SendResponseMessage(Message)
<<Abstract>>
ServerExtension
+UserControl
+SendRequestMessage(Message)
+SendResponseMessage(Message)
<<Abstract>>
AgentExtension
+UserControl
+SendRequestMessage(Message)
+SendResponseMessage(Message)

Figure 13 Extension Class Diagram. Forwarding mechanism methods.

4.1.1 Logical Implementation

All layers of system component (functional modules and cores) are working on similar logic implemented in extension interface (abstract class). When incoming message is detected the first thing is to check Action. If action equals "Apply" the message is meant to this certain module and need to be processed. Otherwise message will be send to next level of functional modules.

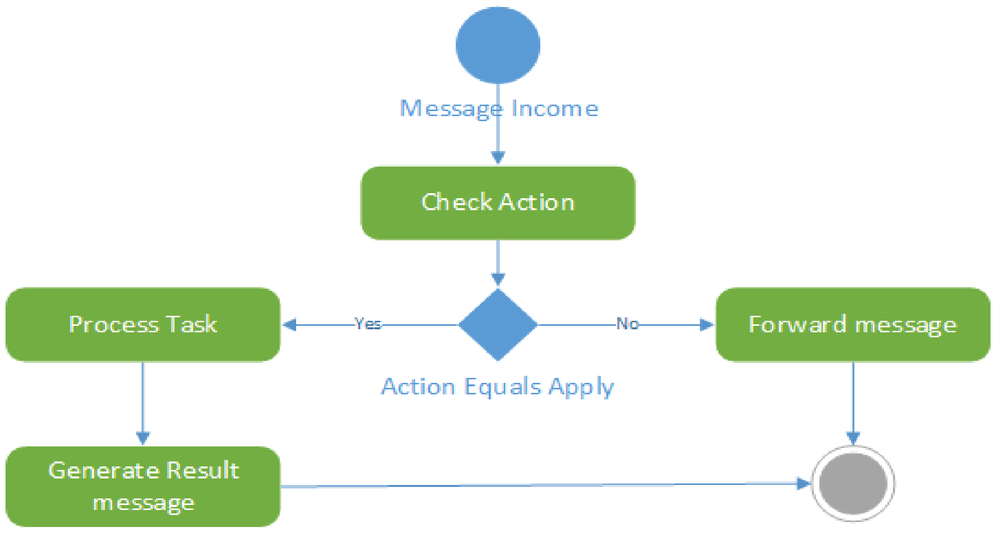
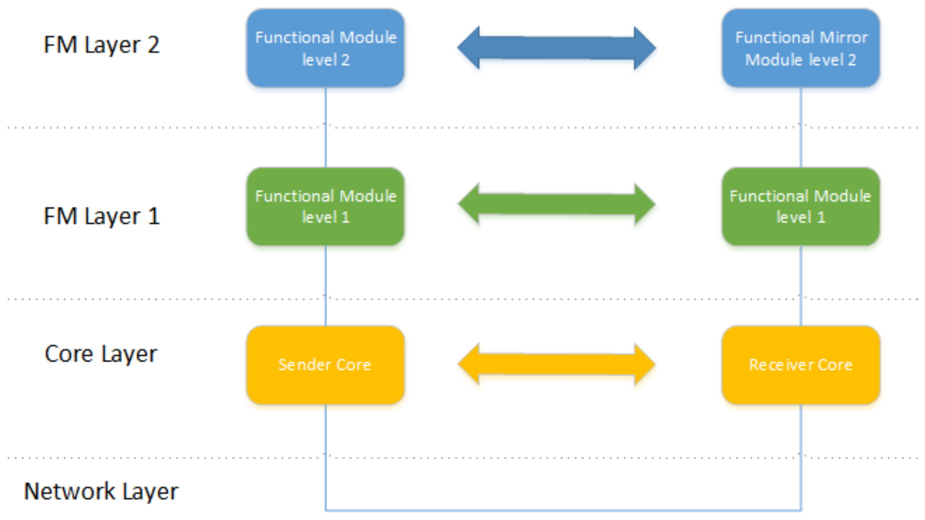


Figure 14 Message Activity Diagram of each Functional Module

In this way using limited messaging interface message from newly added module can reach mirror module on other side. Using hierarchical modules structure it is easy to find a destination module. On each level of modules message is being wrapped and destination is set to mirror module at the same level on different side.

Once message is reached a network level and traveled to receiver side it starts moving from core to destination module. Within each level a module unwraps message and search next module by inner message destination address. The procedure repeats until message reached destination module.



FM Layer 2
Functional Module level 2
Functional Mirror Module level 2
FM Layer 1
Functional Module level 1
Functional Module level 1
Core Layer
Sender Core
Receiver Core
Network Layer

Figure 15 Communication Model

Communication model is similar to ISO OSI[10] Communication model where layers exchange messages using lower level services.

There is a special case when no child component is found. This is not an error. The complete hierarchy of extension modules exists only on central server. On agent side child modules are loaded on first call. To module who didn't foun a destination child module it means that the child module need to be downloaded and installed before call.

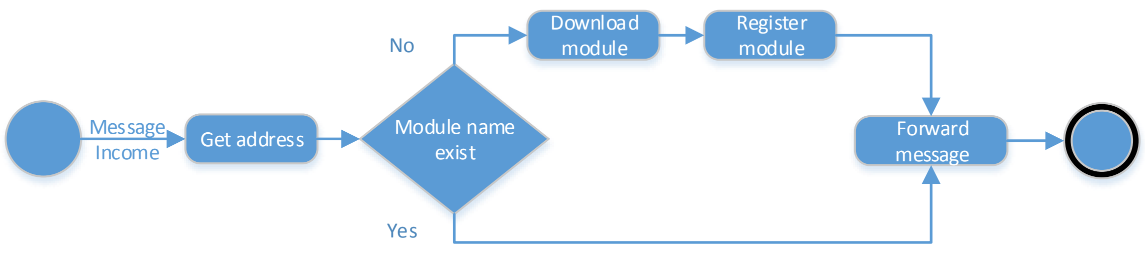


Figure 16 Message forwarding use case diagram. Missing child module

4.1.2 Code Implementation

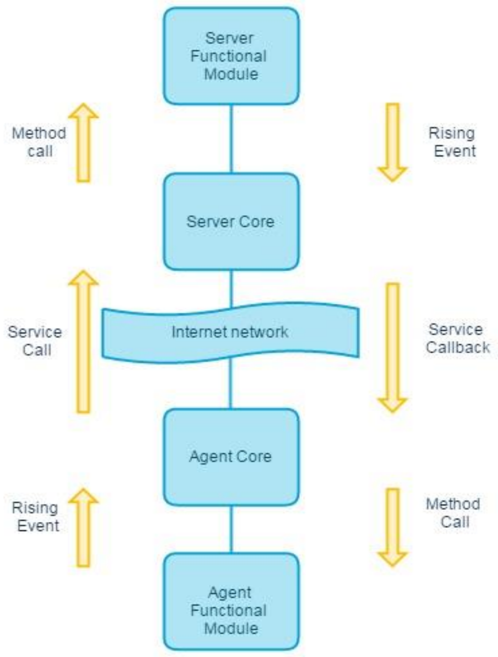


Figure 17 Communication Architecture

In logical implementation there is no difference between message moving to core direction and from core, but on code implementation they are two different methods of forwarding message.

To core

Message moving to core using event bubbling method. When message moves to the core direction through the layers it uses an events rising. Each functional module has an event rising mechanism that triggers when module have something to send. Functional module has list of sub modules and event handling mechanism. When one of sub modules raise an event the parent module get message from event using event handler. If message are meant for this certain module it process message data, otherwise it rises event with message inside and send message to parent modules.

```
public abstract class Extension{
    ...

    public virtual void SendMessageToParent(Message message){
        if (OnMessage != null)
            OnMessage(this, new MessageEventArgs { Message = message });
    }

}
```

To Functional Module

On the way from the core to the functional modules system uses method calls. Each functional module reads message destination address and check list of sub modules, when the right sub module is found then the message is sent to selected module.

```
public abstract class Extension{
    ...
    public virtual void SendMessageToChild(Message message){
        if (message.Action == MessageAction.Forward){
            var extension = Extensions.FirstOrDefault(x =>
                 x.GetFilter(message.InnerMessage.Address));
            if (extension != null)
                extension.SendMessageToChild(message.InnerMessage);
        }
        else if (message.Action == MessageAction.Apply){
            NotifyMessageIncome(message);
        }
    }
    ...

}
```

As a result we need to have both methods implementation on each module type. But depending on module type the call will be crossed. On Server type:

```
public abstract class ServerExtension : Extension{
    ...

    public void SendRequestMessage(Message message){
        this.SendMessageToParent(message);
    }

    public void SendResponseMessage(Message message) {
        this.SendMessageToChild(message);
    }

}
```

On Agent type:

```
public abstract  class AgentExtension : Extension{

    public override void SendRequestMessage(Message message){
        this.SendMessageToChild(message);
    }

    public override void SendResponseMessage(Message message){
       this.SendMessageToParent(message);
    }
}
```

Through the network

At the moment of crossing network the way how it is been transferred depends not on role in communication model but on system elements. At the way transferring message from agent to server is simple service call. In opposite case the mechanism is a bit complicated.

For this purpose a call-back technology is used to create duplex communication channel. For creating two side service there is a need in two components. The service interface:

```
    [ServiceContract (CallbackContract = typeof (IMessageCallback))]
    public interface IOctopusService
    {
        [Operation Contract (IsOneWay = true)]
        void Subscribe(String Name);

        [OperationContract (IsOneWay = true)]
        void Unsubscribe();

        [OperationContract(IsOneWay = true)]
        void SendMessage(Message message);
    }
```

And a call-back interface:

```
    interface IMessageCallback : ICommunicationObject
    {
        [OperationContract (IsOneWay = true)]
        void Notify(Message message);
    }
```

While travelling through the system message have two states: XML Format and DataObject type. DataObject is used in code. Functional module creates message as an object and sends to parent functional module. Upon message reaches core level it converts into the xml format and travels through the network. At the other side, when core receive a message it converts it back to an object and sending to child functional module.

Message travels from sender to receiver. At sender side each layer wraps message and put his own header address. After reaching a receiver side on each level unwrap message and send inner message to next level module according destination address. After message was processed at destination module, s status message travels back to sender. The way how it is moving through the design levels is reversed. On agent side message is being capsulated and unwrapped on a server side.

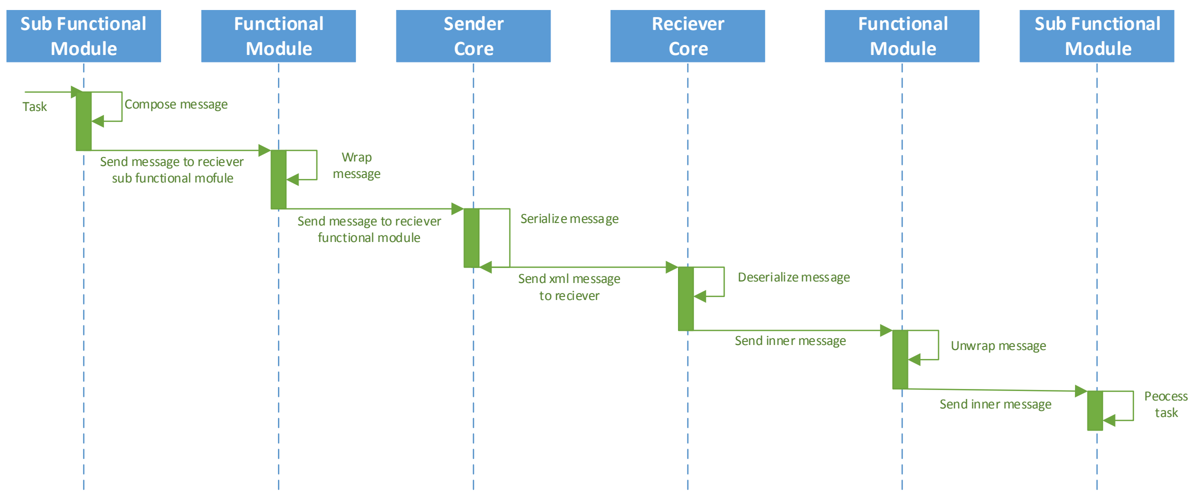


Figure 18 Message movement through layers sequence diagram

When user gives an instruction for a system the task appears. Task can be initialized by scheduled event or an error handling logic also. The functional module who got a task construct a set of commands – instruction for an executive module which is located on another system component. When script is done the functional module creates a message and sets a destination module name. On the way through the sender modules the message is wrapped on each of them and a certain level name is specified. When message reaches to destination point and acknowledge message need to be sent then sender and receiver changes roles and message travels back at the same way.

4.2 Message Queuing

By reason of unavailability to make direct calls and processing return values due to dynamic application structure the communication within project is asynchronous, what means when functional module generated message is sent to a mirror functional module it does not go to awaiting state. For this reason a special solution need to be implemented to make possible getting operation result to initial module. For sending result a mirror module uses a message. In a logical representation a message can be divided in two groups.

Task Message is a message which is sent for processing to receiver. In cases when task requires a feedback to show to end user or just store processing result to database there is a need in task *Result Message*. Those messages are moving from receiver to sender with instruction execution result.

To handle response messages, extension have a special mechanism. Each generated message have unique task Id, only result message have same id as a task message. *TaskID* is used at the moment when message arrives to functional module. If task queue contains received message task id it means that arrived a result message it and need to be handled in other way.

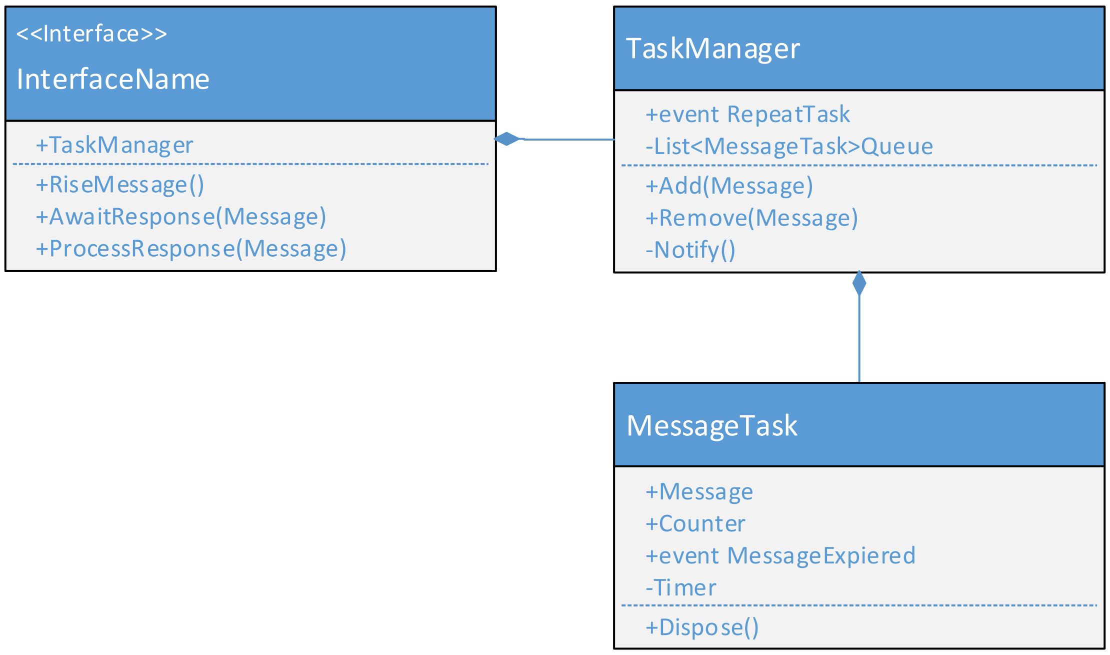


Figure 19 Extension Class Diagram. Message queueing mechanism methods.

Not in all cases module operation need a result message, but in cases when it do it uses a message queueing mechanism. When extension rises a message it being encapsulated to *MessageTask* class and stored to *TaskManager*. It has a timer to notify manager about message expiration:

```
    public class MessageTask{
        ...
        public MessageTask(Message msg){
            ...
            timer.Elapsed += TaskExpiered;
            ...
        }

        private void TaskExpiered(object sender, ElapsedEventArgs e){
            if (MessageExpiered != null){
                MessageExpiered(this, new TaskEventArgs { Task = this });
            }
        }
        ...
    }
```

TaskManager uses *MessageTask* counter property to calculate resend attempts. When it reaches maximum allowed number the status of message will be changed to error and forwarded to awaiting module:

```
    public class TaskManager{
        public MessageHandler RepeatTask;
        private List<MessageTask> Queue;

        public void Add(Message message)        {
            var task = new MessageTask(message);
            task.MessageExpiered += Notify;
            Queue.Add(task);
        }

        ...

        private void Notify(object sender, TaskEventArgs e){
            if (e.Task.Counter > 3){
                var message = e.Task.Message;
                message.Status = "Unreached";
                if (RepeatTask != null) {
                    RepeatTask(this, new MessageEventArgs{Message = e.Task.Message});
                Queue.Remove(e.Task);
                e.Task.Dispose();
            }
            e.Task.Counter++;
        }
    }
```

In case when result message arrives successfully a task manager removes message from queue by task id. Method that requires a result need to be able to handle it.

4.3 Message Instruction

As long as direct method call from server side to agent functional module is impossible due to dynamic components common interface there is need to compose small textual commands with parameters – instructions to make possible tasks understanding for mirror functional module. A set of instructions or script is generated by server functional module when user makes changes in configuration or gives tasks to agent through the central server using extension visual part. A script need to be sent from server side functional module to agent side functional module using system local standard message and be processed on agent side.

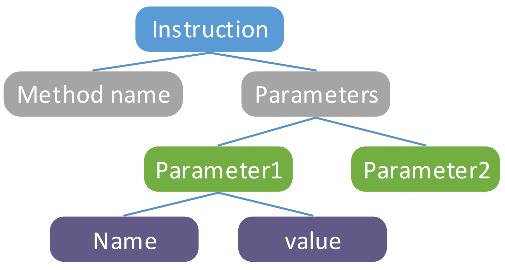


Figure 20 Instruction semantic structure

Instruction is a string which consist of two parts: called method name and parameters. The amount of parameters is limited by custom operation signature. Each parameter have a leading name and its value.

By purpose instructions are divided into task instruction which is sent to receiver with task message and result which are sent back from receiver. Each operation who is generating a message and awaiting result should recognize the result message addressed to the same operation. To decode instructions to method call an extension interface have a proxy entry method.

To have a similar method set all modules of one Extension inherit base interface. Each module implements operations according to it needs.

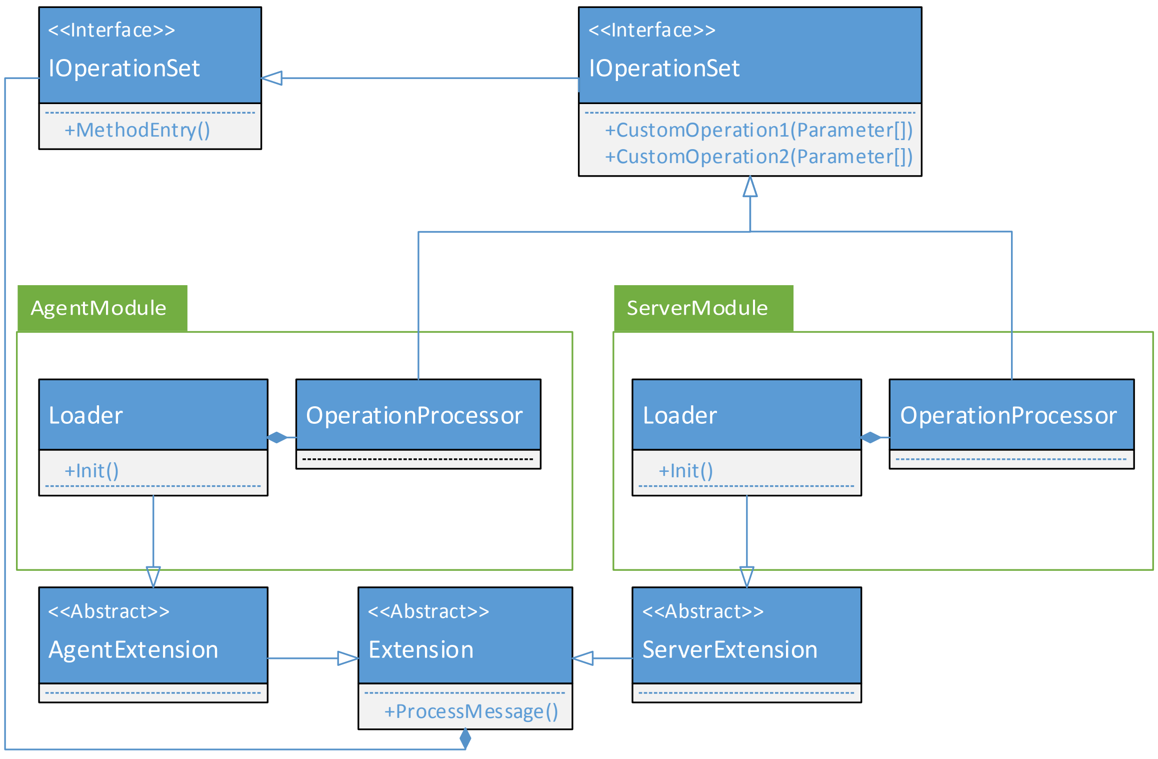


Figure 21 Extension Class Diagram. Common operations set.

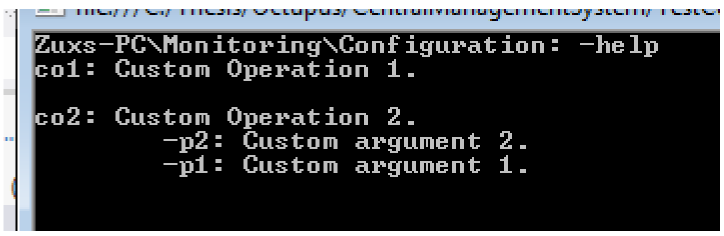
In case if extension interface have a few operations it is easy to define methods short names and parameters set. But in opposite situation, when we have more than dozen of operations, the management of names can became an issue. A .NET platform have a Reflection classes[11] which allows to use custom attributes on classes, methods and properties.

Using reflection class and custom attribute classes *MethodNameAttribute* and *MethodParamAttribute* is possible to add method and parameters short names at the moment of definition custom extension:

```
public abstract class CustomExtension : OperationSet
    {
        [MethodNameAttribute(Name="co1",Description = "Custom Operation1")]
        public abstract string  CustomOperation1();

        [MethodNameAttribute(Name = "co2", Description = "Custom operation 2.")]
        [MethodParamAttribute(Name = "p1", Description = "Custom argument1.")]
        [MethodParamAttribute(Name = "p2", Description = "Custom argument2.")]
        public abstract string CustomOperation2(String param1,String param2);
   }
```

Attributes can be used within custom method implementation for string instruction creation and parsing. The method *GetHelp* is used for showing list of available operations and its descriptions:

```
public abstract class OperationSet{
   public string GetHelp(){
      string info = String.Empty;

      foreach (var operation in this.GetType().GetMethods()){
         try{
            var method = (MethodNameAttribute)
            operation.GetCustomAttributes(typeof(MethodNameAttribute), true)[0];
            info += method.Name + ": " + method.Description + Environment.NewLine;

            var parammeters = (MethodParamAttribute[])
            operation.GetCustomAttributes(typeof(MethodParamAttribute), true);
            foreach (var param in parammeters)
               info += "\t-" + param.Name + ": " + param.Description +
                                                       Environment.NewLine;
            info += Environment.NewLine;
          }catch(Exception) {
                  continue;
          }
        }
        info += Environment.NewLine;
        return info;
  }
}
```



```
Zuxs-PC\Monitoring\Configuration: -help
co1: Custom Operation 1.

co2: Custom Operation 2.
        -p2: Custom argument 2.
        -p1: Custom argument 1.
```

Figure 22 Help tool output example

Custom attribute classes:

```
[AttributeUsage(AttributeTargets.Method)]
    public class MethodNameAttribute : Attribute
    {
        public String Description { get; set; }
        public String Name { get; set; }
    }

    [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
    public class MethodParamAttribute : Attribute
    {
        public String Description { get; set; }
        public String Name { get; set; }
    }
```

4.4 Message Addressing

On the way to central server there is no need to determine destination point – it is only one. But in opposite case, when message is traveling to agent it is important to which one. When central server receives a task from user or other control part as a parameter is sent a destination agent name which will be set to message object at the moment of creation.

When message reaches network layer it don't use *Address* – the module name. There is many agents with same modules. A *Destination* agent name is stored in separate property and used only at service call back:

```
public class OctopusService : IOctopusService{
    private static readonly Dictionary<String, IMessageCallback> Agents =
       new Dictionary<String, IMessageCallback>();

    public static void ForwardMessage(Message message)
    {
        try{
            foreach (var agent in Agents){
                if (agent.Key.Equals(message.Destination)){
                    if (agent.Value.State == CommunicationState.Opened)
                        agent.Value.Notify(message.InnerMessage);
                }
            }
        }
       ...
    }
}
```

4.5 Network Communication

On the way from sender to receiver the message need to be sent over the network in both ways. To network layer it mean that it must have not only service endpoint to listen client requests and send responses but a mailing functionality also – when server send request to clients without client call. A web service which combines those two functionalities is called Call-back services.

For creation a network layer is used Windows Communication Foundation library from .NET framework. It also supports call-beck technology. This chapter highlights a network layer implementation.

In order to make possible communication between newly added modules through the network the cores need to have:

- Service Endpoint
- Subscription mechanism
- Call-Back mechanism

On Figure 10 is demonstrated prototype network layer classes hierarchy. On a service there are service interface *IOctopusService* which is used as a service metadata for clients and service implementation *OctopusServiceClient* class. As long as service have duplex communication channel it need to have a call-back communication interface - *IOctopusServiceCallback*.

Subscription mechanism is quite simple and consist of three components subscribe, unsubscribe methods and list of subscribed clients – Agents. In scale of network layer each client is a saved callback channel.

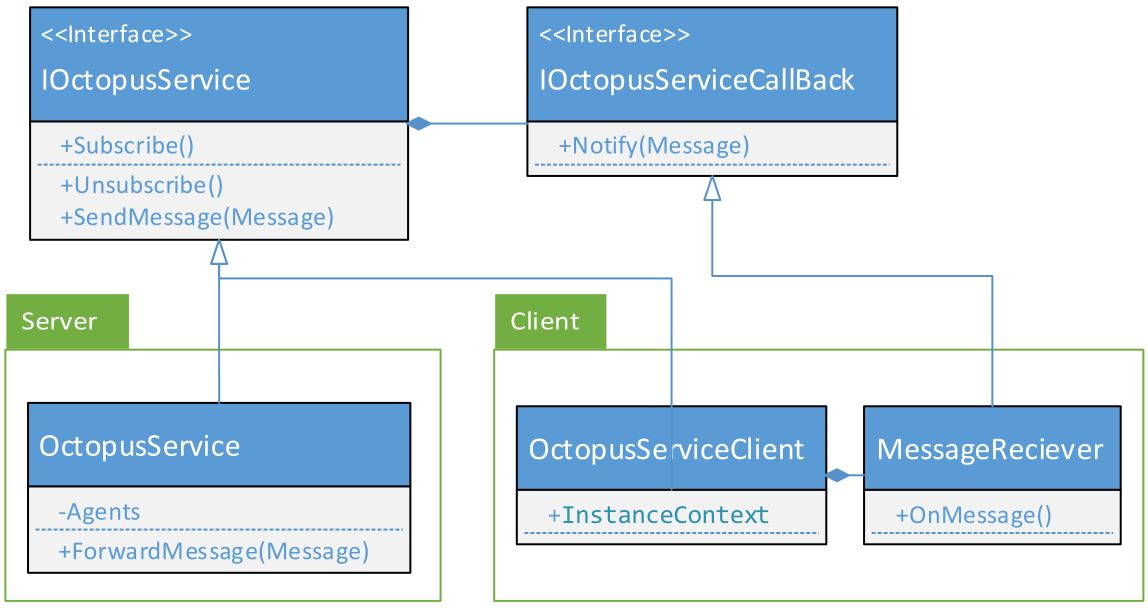


Figure 23 Network Layer Classes Class Diagram

Based on prototype modular application architecture the system cores do not need more operations than sending messages through the network and forwarding to functional modules. In network layer the way how message do not depends on roles from communication model it depends on system component.

In case when agent want to send a message it makes a simple service call using service client object:

```
    public OctopusServiceClient Octopus;
        public void SendToServer(Message message) {
            Octopus.SendMessage(message);
        }
    }
```

The request is being processed on a server side. In different case when server sending message to agent it uses *IOctopusServiceCallback* interface to send message to call-back callback:

```
    public class OctopusService : IoctopusService{
        private static readonly Dictionary<String, IMessageCallback> Agents =
           new Dictionary<String, IMessageCallback>();

        public static void ForwardMessage(Message message){
              foreach (var agent in Agents){
                     if (agent.Key.Equals(message.Address))
                         if (agent.Value.State == CommunicationState.Opened)
                             agent.Value.Notify(message.InnerMessage);
        }
```

Agent call-back channel is registered on mailing subscription operation. At the moment when agent core receives a message a *MessageReciever* rises an OnMessage event to forward message to functional modules.

```
    public class MessageReceiver : IOctopusServiceCallback
    {
        public event MessageHandler OnMessage;
        public delegate void MessageHandler(object sender, MessageEventArgs e);

        public void Notify(Message message)
        {
            if (OnMessage != null)
                OnMessage(this, new MessageEventArgs { Message = message });
        }
    }
```

Agent cannot be detected automatically without additional development. At this stage of development the only way to find agent if a subscription call to predefined service endpoint.

5 Putting all together

In this part the prototype project will be considered a bit closer with module examples using principles described before. The sample contains two extensions: The Monitoring extension and configuration. The configuration extension is a sub module of a monitoring. Which means that all messages meant to configuration module will be going through monitoring module.

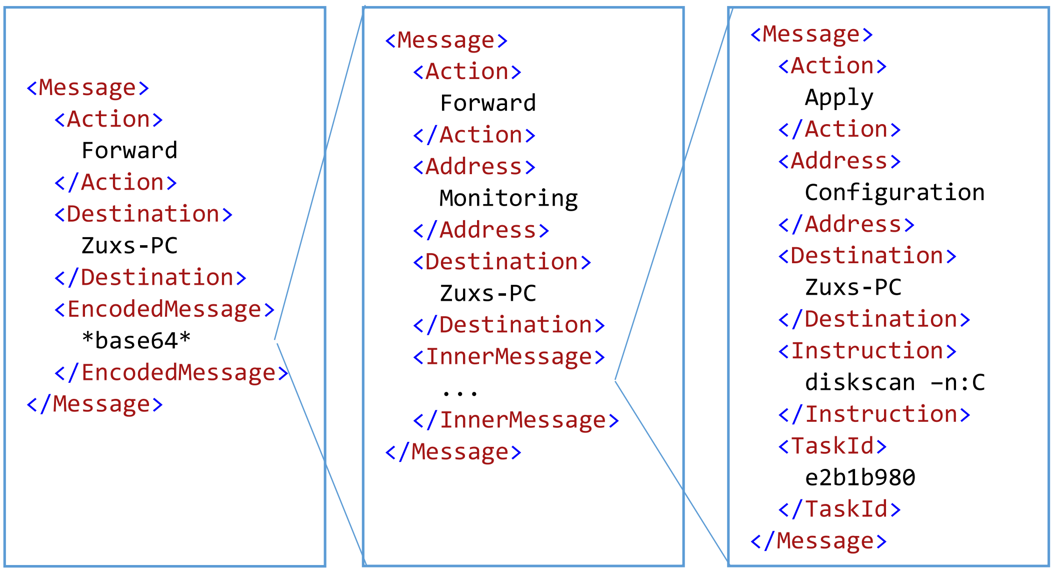
Thought the layers of modules message is traveling using methods described in chapter 4. In sample case there is 3 levels on each side: the core as a root of hierarchy, it has one child module Management module which has a sub functional module Configuration module. When configuration module decides to get ZUXS-PC workstation storage free space it composes an instruction “diskscan –n:C” and creates a message shown on figure 24. The initial message is shown on a figure c).

On the way to core the message is being wrapped on an each layer - at monitoring level figure b) and at core level part c).Message is transferred to core by rising events.

On figure 24 is shown an xml representation of message unwrapping on an agent levels. The message *a)* is an xml arrived from the network. From the filled properties it has only Action – Forward, Destination – current workstation name, and message b) encoded in base64 format. To send message to next level the core decodes message to data object and send to child module using its name as an address.

In this case the Monitoring module is used only as a middle module and don’t do any additional tasks. The b) message has only filled properties needed to forward message. Inner message is being transferred to next child module.

And a last part of message – part c) has action set to “Apply”. The module reads instruction, finds a right method and process it.



```
<Message>
  <Action>
    Forward
  </Action>
  <Destination>
    Zuxs-PC
  </Destination>
  <EncodedMessage>
    *base64*
  </EncodedMessage>
</Message>
```

a) Agent Core level

```
<Message>
  <Action>
    Forward
  </Action>
  <Address>
    Monitoring
  </Address>
  <Destination>
    Zuxs-PC
  </Destination>
  <InnerMessage>
    ...
  </InnerMessage>
</Message>
```

b) Monitoring level

```
<Message>
  <Action>
    Apply
  </Action>
  <Address>
    Configuration
  </Address>
  <Destination>
    Zuxs-PC
  </Destination>
  <Instruction>
    diskscan –n:C
  </Instruction>
  <TaskId>
    e2b1b980
  </TaskId>
</Message>
```

c) Configuration level

Figure 24 Message XML Representation on a different module levels

According to task semantic structure mentioned in chapter 4.2 it is easy to find a right method and give parameters. The example of entry method custom implementation

```
public class Executor : IConfigurationExtension{
        ...
        public string Execute(String instruction)
        {
            var method = instruction.GetMethodName();
            switch (method){
                case "diskscan":
                        var name = instruction.GetParamValue("n");
                        return AnalyzeStorage(name);
                    break;
                case "liststorages":
                    return GetStorageList();
                    break;
            }
            throw new UnknownOperationException(method);
       }
```

Two helper methods to parse instruction:

```
public static class InstructionParser{
       ...
        public static string GetMethodName( this String instriction){
            return instriction.Split(' ')[0];
        }

        public static string GetParamValue(this String instriction, String paramName)
        {
            foreach (var param in instriction.Split(' ')){
                if (param.Contains(':')){
                    var parts = param.Split(':');
                    var name = parts[0].Trim('-');
                    if (name == paramName)
                        return parts[1];
                }
            }

            throw new ParamNotFoundException(paramName);
        }
    }
```

6 Diagnostics

One of the reasons of current application architecture is to avoid errors which appears while static components is being updated. For this reason a static component must have minimum functionality to avoid unnecessary updates. The main issue that is need to be concern is a stability of a system and possibility to diagnose it.

To diagnose a module a man functionality need to check possibility of adding, removing and updating child modules. The easiest way to implement diagnostic it to use some simple child module which can be added to whatever component. The diagnostic method is going through main core tasks checked on each module. Next steps need to be done:

Step 1 – Download dummy module to local drive.

Step 2 – Apply child module to extensions library.

Step 3 – Call test method. The result need to be asserted with predefined value.

Step 4 – Unregister library and delete from drive.

Step 5 – Steps from 1 to 3 is being repeated for next version of dummy module.

Step 6 – Call test Method. If result is different from previous call then update was successful.

Step 7 – Clean test data.

Steps listed above is used for core, but possible that custom modules might need some additional testing. Extension interface has an entry method for custom tests *RunTests*. Which is used after module is initialized and added to library and before start of usage:

```
    [Export(typeof(AgentExtension))]
    public class Loader :  AgentExtension{

        public override string RunTests(){
            String errors = String.Empty;
            errors += TestHardRdiveAccess();
            errors += customTest2();
            return errors;
        }

        private string TestHardRdiveAccess(){
            String result = String.Empty;
            try{
                var ds = Directory.GetAccessControl(Location);
            }
            catch (UnauthorizedAccessException e){
                result += e.Message;
            }
            return Result;
        }

        private string customTest2() {
            ...
        }
    }
```

Tests entry method can be used as an initial check or can be periodically. System critical component need to be checked periodically for solving errors and prevent crashes. In this case custom test mechanism is implemented in simple way as a one possible way.

6.1 System stability

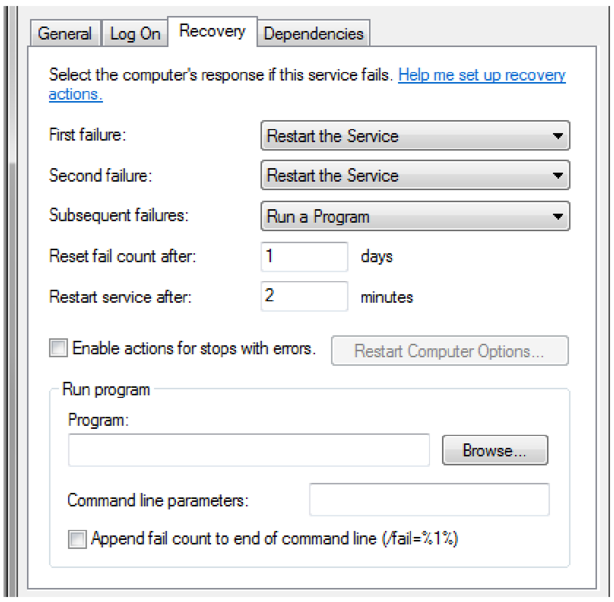
For a dynamic system, where all modules are added and deleted dynamically, an important issue is stability. First step to get a stable application is well tested code. To predict all errors is almost impossible and that is why errors appears in every code. The second step of achieving stable system is proper error handling.

There are a few case of system behavior in error occasion. The handled exception can be solved locally in module or in central server. An unhandled error usually cause an application crash. If central server is crashed then it need to be recovered automatically or manually. At time when central server is unavailable agents are working in data accumulative mode. There might be situations when agent is crashed. A special recovery mechanism is provided by run environment – Windows.

Useful windows feature that can be used in project is services recovery. In case if service crashes few times in a row it is possible to run a second application which might be remote access session for problematic agent.

Another useful tool for diagnostics is a detailed logging. A useful data can be collected in runtime and sent to central server for storage for later analyze by user. A popular framework used in .NET platform is NLog[12]. It is simple in used and have two features is leveling of logging and good multiples log files handling system.

As a result of combining these two tool is an auto reporting system. A possible crash recovery scenario:



Select the computer's response if this service fails. Help me set up recovery actions.

First failure: Restart the Service
Second failure: Restart the Service
Subsequent failures: Run a Program
Reset fail count after: 1 days
Restart service after: 2 minutes

☐ Enable actions for stops with errors.

Run program
Program:
Command line parameters:
☐ Append fail count to end of command line (/fail=%1%)

Figure 25 Service Recovery

- First failure: Change logging configuration to trace level and a separate streaming file. After configurations were changed start service.
- Second failure: Submit log file to central server. Restart service again on same log configuration preferences.
- Third failure: Submit fresh logs. And Register workstation in technical support for a maintenance.

Having two logs files right before crash will be make easier to find an issue.

Conclusion

The goal of this work was to work out a fully component based application architecture and prove operability of prototype project by solving problems related to architecture. For example a biggest issue was a communication between dynamic modules, second challenge related to project requirements was a runtime components updating and etc.

The application type were chooses as an example of applying Component Based Architecture and didn't had strict requirements to functionality. Since the project was designed as a prototype it do not fulfil all requirement to real life project. The reason for this is that only real life environment can dictate requirements which can be different from project to project, but it demonstrates well the flexibility of architecture and possibilities.

The central server were hosted on IIS and listened agent client located on another test machine. They exchanges messages in XML format trough the network and component within system were able to communicate with each other without having static addressing environment.

Agent is possible to find a server with predefined service endpoint or work in offline mode with data accumulating for sending later. It was also able to download mirror modules from Central Server and register them without restarting a program. The custom module were able to receive tasks, process and send a response task to server side. All middle modules were able to forward messages in both side with choosing right address.

As an example were chosen CMS system but the architecture can be applied to every kind of enterprise application. It is not reasonable to use the solution to small program because of complexity of design – amount of work spent on core development should not be biggest part in development process. Simpleness and universality from outside means complexness from inside. The decision of choosing this architecture must be fully motivated by conditions. For example if:

- The project is a long life. As it was mentioned before the updating of small component is easier than updating the whole system.
- The all functional requirements to project or system is not predefined.

The worked out architecture design and solutions makes possible of using modular extensibilities containing system functionality. The approach of software creation can be applied not only to applications what are working on an OS platform but OS itself can be designed in this way. As a simple and a bid sci-fi example it could be a cluster of robots controlled by central server. The OS composition of each node might be constructed according to machine objectives or even of physical configuration.

Resümee

Käesoleva töö põhieesmärk oli välja töötada laienduste haldussüsteemi rakenduse baasarhitektuur, luua selle põhjal töötav prototüüp ja tõestada selle toimimist. Töös lahendati mitmeid vaatlusaluse arhitektuuriga seonduvaid probleeme. Üks suurematest probleemidest oli suhtlemine dünaamiliste moodulite vahel. Teine väljakutse oli seotud töö eesmärkidega, nimelt püstitati nõue, et laiendusi peaks saama hallata programmi töötamise ajal.

Näidisrakenduse tüübiks valiti hajutatud süsteem, millel ei olnud rangelt defineeritud funktsionaalsuse nõudeid. Kuna projekti näol oli tegu prototüübiga, ei olnud reaalse alussüsteemi kasutamine tarvilik. Reaalses ettevõttes loob konkreetne töökeskkond spetsiifilised nõuded, mis võivad valdkonniti erineda, samas demonstreerib antud töö käsitletava arhitektuuri paindlikkust ja võimalusi.

Keskserveri tarkvaraks valiti IIS server, mis kuulas klientide päringuid. Klient ja server suhtlesid omavahel kasutades XML-vormingus sõnumeid. Väljatöötatud aadressisüsteem võimaldas süsteemikomponentidel omavahel dünaamilises keskkonnas suhelda.

Agent oli võimeline leidma etteantud veebiaadressiga serveri ja töötama võrguühenduseta režiimis. Viimasel juhul kogus agent andmeid, et need hiljem keskserverile saata. Juhul kui saabus sõnum, kus aadressiväljal oli kirjeldatud agendi jaoks tundmatut moodulit, oli see võimeline keskserverilt vastavat infot pärima ja mooduli süsteemi paigaldama. Serveri moodulid võimaldasid saata ülesandeid kliendile ja klient suutis neid ülesandeid täita ning vastuse tagasi saata. Alammoodulite süsteem suutis edastada sõnumeid õigele aadressile.

Näidisprojektiks oli valitud CMS süsteem, kuid loodud arhitektuuri saab rakendada igat liiki rakendustarkvarale. Väikesemahuliste programmide jaoks ei ole väljatöötatud lahendust mõistlik kasutada, sest tuuma arenduse ajakulu ei tohiks olla märkimisväärselt suur võrreldes kogu tarkvara arenduseks kulunud ajaga. Raamistiku väljaspoolne lihtsus ja universaalsus toob endaga kaasa seespoolse keerukuse. Enne selle arhitektuuri kasutamist tuleb kindel olla et:

- Projekt on pikaajaline. Nagu juba eespool mainitud, süsteemis väikese osa uuendamine on lihtsam, kui kogu süsteemi ajakohastamine.
- Kõiki projekti või süsteemi funktsionaalseid nõudeid ei ole antud töös kirjeldatud.

Käesolevas töös loodi tarkvaraarhitektuur ja sellel põhinev prototüüplahendus, mis võimaldasid rakendada olemasolevale arvutussüsteemile modulaarset laiendatavust. Antud juhul on tarkvaraarendusel võimalik luua rakendustarkvara mingile kasutatavale operatsioonisüsteemile, kuid samuti võib operatsioonisüsteemi enda vastavalt kavandada.

Siinkohal näeb töö autor oma visioonis veel seni veel *sci-fi* valdkonda kuuluvat näidet, kus robotite klastrit juhib keskne server ning operatsioonisüsteemi kooseisu iga lüli on loodud vastavalt konkreetse masina vajadustele või isegi selle füüsilise konfiguratsiooni järgi.

References

[1] Windows Server [2015] [WWW] https://technet.microsoft.com/en-us/library/bb625087.aspx

[2] .NET platform [2015] [WWW] http://en.wikipedia.org/wiki/.NET_Framework (25.04.2015)
[3] SQL Server [2015] [WWW] https://msdn.microsoft.com/ru-ru/library/bb545450.aspx(25.04.2015)

[4] IIS Web Server [2015] [WWW] http://www.iis.net/learn/get-started/introduction-to-iis/iis-web-server-overview (25.04.2015)

[5] NTFS [2015][WWW] http://blog.sqlauthority.com/2012/07/06/sql-server-ntfs-file-system-performance-for-sql-server/ (25.04.2015)

[6] MEF [2015] [WWW] https://msdn.microsoft.com/en-us/library/dd460648%28v=vs.110%29.aspx (1.05.2015)

[7] Chapter 1. Definition of a Software Component and its Elements. George T. Heineman, William T. Councill (2001). Component-Based Software Engineering: Putting the Pieces Together. London: Addison-Wesley Professional. p8.

[8] Service layer. Martin Fowler (2002). *Patterns of Enterprise Application Architecture*. London: Addison-Wesley Professional. p133.

[9] WCF [2015] [WWW] https://msdn.microsoft.com/en-us/library/ms731082%28v=vs.110%29.aspx (25.04.2015)

[10] ISO OSI Communication model [2015][WWW] http://en.wikipedia.org/wiki/OSI_model (1.05.2015)

[11] Reflection Classes [2015][WWW] https://msdn.microsoft.com/en-us/library/f7ykdhsy%28v=vs.110%29.aspx (1.05.2015)

[12] Open Source logger NLog [2015][WWW] http://nlog-project.org/(1.05.2015)