



Norwegian University of
Science and Technology

Prototyping an OPC UA Server for Use in an Information Management System

Kristian Mo

Master of Science in Engineering Cybernetics

Submission date: June 2011

Supervisor: Sverre Hendseth, ITK

Problem Description

As oil fields produce less in the later part of their life cycle oil companies must look into methods for making their operations more efficient. This will increase profits and extend the lifetime of their oil fields. Integrated Operations (often called IO) are one of the methods for increasing efficiency. IO implies that data must be shared between different levels in the company. ISO 15926 “Industrial automation systems and integration—Integration of life-cycle data for process plants including oil and gas production facilities” is an ISO standard for data integration, sharing and exchange.

OPC UA is one of the possible technologies for sharing data organized according to ISO 15926. Siemens in Trondheim is currently participating in a project where they are modeling the Snorre A and Snorre B oil platforms according to ISO 15926. This master will try to implement a prototype OPC UA server for Siemens Oil and Gas in Trondheim. Siemens save their historical and current data using a tag.field data model; this should be represented in the OPC UA server. The thesis will investigate if ISO 15926 models can be implemented using the OPC UA Part 3 - Information Model, and investigate if connecting the tag.field model to the ISO model is doable.

The server should offer the following communication methods (security and signing is not prioritized for the prototype); OPC UA TCP and SOAP/Web services.

Research questions

- RQ 1
 - Build an OPC UA server framework based on the SDKs and the communication stack to perform the basic setup of services and security. In other words make communication, read/write of simulated values, encryption/encoding and certificates work.
- RQ 2
 - Evaluate different ways of building the server Information Model based on the Siemens tag.field information model.
- RQ 3

- Try to build a server Information Model based on an ISO 15926 model.
- RQ 4
 - Test the server for its Profile compliance using the Compliance Testing Tools(CTT) from the OPC Foundation.

The server should have these general properties:

- Modular
- Small amount of configuration before start up

Assignment given: 6, January 2011

Supervisor: Sverre Hendseth

Master's Thesis

Prototyping an OPC UA Server for Use
In an Information Management System

Kristian Mo

Spring 2011
Department of Engineering Cybernetics
Faculty of Information Technology,
Mathematics and Electrical Engineering
Norwegian University of Science and Technology

Abstract

When oil & gas fields produce less in the late part of their life cycle, oil & gas companies must run their operations more efficient. One of the methods for running their operations efficiently is called Integrated Operations (IO). Rapid introduction of IO can possibly save 250 billion NOK from 2005-2015. Stage one of IO involved creating onshore expert centers which advises the operators offshore. Stage two will involve integrating suppliers and the oil company itself using common data models. ISO 15926 “Industrial automation systems and integration—Integration of life-cycle data for process plants including oil and gas production facilities” is an ISO standard which may facilitate this data integration. Siemens is currently participating in a project where they are modeling the Snorre A and B oil platforms according to the ISO 15926 standard.

OPC (Openness, Productivity and Connectivity) specifications are communication specifications for process plants, and are the dominant specifications within the automation industry. OPC UA is the newest OPC specification and was created to use new IT technologies, be platform independent, and handle complex process models. OPC UA should offer the technology and platform for implementing ISO 15926 models and be one of the technologies used in stage two of IO.

The purpose of this thesis has been to determine if OPC UA can be used for presenting ISO 15926 models, with the consequence that OPC UA technology may be part of an implementation for IO. To achieve this goal an OPC UA server prototype was created for Siemens. The prototype server retrieves current and historical data from Siemens’s PIMAQ software system. When the server was connected to PIMAQ an ISO 15926 model of Snorre B was created in the server and the data in PIMAQ was connected to this model.

The prototype server that has been created supports reading of current and

historical values from PIMAQ. Reading of current values passes the tests with some warnings for minor non-compliance with the OPC UA standard. However the rest of the server is not compliant and as a consequence not yet ready for industrial deployment.

The OPC UA server prototype has been able to present the ISO 15926 model of Snorre B to clients. Only parts of the ISO 15926 has been modeled in the server because of time constraints, but enough has been modeled and implemented to show that an OPC UA server can represent ISO 15926 models. This also implies that OPC UA may be used by a company in stage two of implementing Integrated Operations.

Contents

1	Introduction	1
1.1	The Context for the Thesis	1
1.1.1	OPC	2
1.1.2	Information Management Systems (IMS)	2
1.1.3	PIMAQ	2
1.1.3.1	DynamicProcessAPI	3
1.1.4	Purpose of the Thesis	3
1.1.5	Server Properties Specified by Siemens	4
1.1.6	Acknowledgments	4
1.1.7	Audience	4
1.2	Terms Used in the Thesis	4
2	Background	9
2.1	Integrated Operations (IO)	9
2.1.1	Perspectives on Accidents	10
2.1.2	Risks and Rewards by Implementing Integrated Operations	12
2.2	ISO 15926	14
2.2.1	Why ISO 15926 Exists	14

2.2.2	Upper Ontology	16
2.2.3	A General Overview of the ISO 15926 Specification	16
2.2.4	How ISO 15926 Will Work	18
2.2.5	Advantages and Disadvantages	19
2.3	OPC UA SDKs	21
2.3.1	Licenses	21
2.3.1.1	OPC Redistributables License	21
2.3.1.2	MIT License	22
2.3.1.3	Reciprocal Community License ("RCL")	22
2.3.1.4	Reciprocal Community Binary License ("RCBL")	22
2.3.1.5	Commercial Source Code License ("CSCL")	22
2.3.2	DLLs	22
2.3.3	Programming Languages	23
3	Server and Information Model Design	25
3.1	The Design of the Framework for the UA Server	25
3.2	The Design of the Prototype UA Server	26
3.2.1	The PIMAQ Structure	28
3.3	Designing Information Models	30
3.3.1	Creating an Information Model	31
3.3.2	Description of Tag.Field Model	31
3.3.2.1	Advantages of the Tag.Field Model	33
3.3.2.2	Different Methods for Creating the Model	33
3.3.3	Description of the ISO 15926 Model	36
3.3.3.1	Different Methods for Creating the Model	37

4	Implementation	43
4.1	Choosing the Programming Language	44
4.2	Choosing Solutions to Base the Framework and Prototype on . . .	44
4.3	The Implemented Classes for the Prototype Server	45
4.3.1	UaServer	45
4.3.2	FrameworkNodeManager	46
4.3.3	Boiler.Classes.cs	48
4.4	Implementation Challenges for the Prototype	48
4.4.1	For versus Foreach Loops	48
4.4.2	Loading and Using C++ Dlls in C#, or Using Unmanaged Code in Managed Code	48
4.4.3	Differences Between DynamicProcessAPI and OPC UA	49
4.4.3.1	Data is Saved as a String	50
4.4.3.2	Reading Forwards	50
4.4.3.3	Dead Band Resolution	51
4.4.3.4	Bounding Values	51
4.4.3.5	Fields vs Properties	51
4.4.3.6	Time	52
4.4.4	Implementation Challenges Which Arise From Behavior in the OPC UA Client SDK	52
4.4.4.1	Epoch Requests	52
4.4.4.2	Year 1 Request	53
4.5	The Implemented Classes for the Prototype Server	53
4.5.1	DynamicProcessAPIwrapper	53
4.5.1.1	ReadData()	53
4.5.1.2	ReadField()	54
4.5.2	DynamicProcessAPIinterface	54

4.5.2.1	Initialize()	54
4.5.2.2	Disconnect()	55
4.5.3	DynamicProcessAPINodeManager	55
4.5.4	DescendingOrder	55
4.5.5	TagFieldModelNodemanager	55
4.5.5.1	New()	56
4.5.5.2	LoadPredefinedNodes()	56
4.5.5.3	CreateAddressSpace()	56
4.5.5.4	Read()	57
4.5.5.5	HistoryRead()	58
4.5.5.6	HistoryReadRawData()	59
4.5.5.7	ReadHistoricalData()	60
4.5.5.8	ReadHistoricaldataFromCacheOrSource()	60
4.5.5.9	CreateHistoryData()	63
4.5.5.10	ReadDataavaluesFilter()	63
4.5.6	ISOModelNodeManager	65
4.5.6.1	CreateAddressSpace()	65
4.5.6.2	ReadModelFilter()	66
4.5.7	Conversion functions	66
4.5.7.1	CreateField()	66
4.5.7.2	CreateFieldOfType()	66
4.5.7.3	DataTypeConversion()	66
4.5.7.4	QualityConversion()	66
4.6	Filters and Configuration Files	67
4.6.1	App.Config	68
4.6.2	UaFramework.Config.xml	68

<i>CONTENTS</i>	xi
4.6.3 InstallConfig.xml	68
4.6.4 DynamicProcessAPIinterfaceConfig.xml	68
4.6.5 ISOModelFilter.xml	71
4.6.6 DatavaluesFilter.xml	71
5 Testing	73
5.1 Read and Write Compliance Test	74
5.2 Server Compliance Test	75
6 Discussion	79
6.1 Unifying OPC UA and DynamicProcessAPI	79
6.2 Inheritance	81
6.3 Modularity	81
6.4 HistoryRead and Cache	81
7 Conclusion	83
7.1 Future Work	84
Bibliography	85
A OPC UA project	89
B Server Documentation for Siemens	91

Chapter 1

Introduction

1.1 The Context for the Thesis

As the highest peak has been reached for extracting oil in the North Sea [1], oil companies have been looking into methods for making their operations more efficient and cheaper, in order to increase profit and extend the lifetime of their oil fields. Integrated Operations (often called IO) are one of these methods [2] and IO implies that data must be shared between different levels in the company [3].

ISO 15926 “Industrial automation systems and integration—Integration of life-cycle data for process plants including oil and gas production facilities” is an ISO standard for data integration, sharing and exchange [4], and will be used by the Norwegian oil industry in their second generation integrated operations [5]. OPC UA could be one of the technologies for sharing data organized according to ISO 15926. Siemens in Trondheim is currently modeling oil platforms according to ISO 15926 and is therefore interested in investigating if presenting these models is possible in an OPC UA server, and if OPC UA could be a part of a future product for the IO market. As OPC UA was the target of an earlier project for Siemens in the autumn of 2010, it is not investigated in depth in this thesis. The earlier report is attached and can be viewed if the reader would like to gain more knowledge about OPC and OPC UA.

1.1.1 OPC

OPC specifications are made by the OPC Foundation and used for exchanging data in process plants. OPC stood for OLE in Process Control and currently it stands for Openness, Productivity and Connectivity. OPC is tied to the Windows platform via the COM/DCOM technologies. An OPC server retrieves data from sources like sensors, PLCs and databases and makes the data available for clients. The presentation of the data is called the servers Address Space. An OPC client connects to an OPC server and browses the Address Space and retrieves data from the server for further processing.

The first OPC specification was created in 1996 and has been fragmenting as new technologies have been developed and currently consists of a large amount of specifications. To remedy this the OPC Foundation created OPC Unified Architecture(UA) in 2006 which was designed to be able to incorporate new technologies without having to create new specifications, and to be cross platform i.e. no longer tied to only Windows. OPC UA also offers a better platform for modeling complex processes in its Address Space. More on OPC and OPC UA can be had from [6] or <http://www.opcfoundation.org/>.

1.1.2 Information Management Systems (IMS)

An IMS is a software system that "...gathers live and historical data from sources and presents these for further processing for different users. Processing can mean trending or parameter/model estimation based on the measured data. Some typical requirements on an IMS are access to live and historical data, high throughput, high availability, trending and analysis tools, and meta data support." [6].

1.1.3 PIMAQ

Siemens Oil & Gas in Trondheim delivers among their products the Process Information Management and Acquisition Information Management System or the PIMAQ IMS [7]. PIMAQ is used on oil & gas platforms and other process plants. PIMAQ gathers live data from the signals in the control system and saves these signals in a historical database. This data is used by different applications in PIMAQ for further processing like trending, daily reports and

parameter/model estimation. A typical demand on a PIMAQ is high throughput as there may be large amounts of data that must be moved and processed [6] so efficiency of work is important.

1.1.3.1 DynamicProcessAPI

DynamicProcessAPI is a class that is part of PIMAQ. It provides functions for connecting to data providers, reading/writing of values and disconnecting from data providers. It is the class used for data requests.

1.1.4 Purpose of the Thesis

The purpose of the thesis was to build a working OPC UA server prototype using the OPC UA Standard Development Kit(SDK) and the OPC UA communication stack. Once the server framework was done there was an attempt to model process data according to ISO 15926 in the server and connect this model to a historical data source. To understand the context of the thesis, I needed more knowledge of IO and ISO 15926. To achieve these overall goals I defined, in collaboration with Siemens, these project goals:

- Study Integrated Operations and ISO 15926 more in depth.
- Evaluate the SDKs and decide on the programming language with which to start development.
- As the prototype may be part of a commercial solution the different licenses and their usage must be investigated.
- Build an OPC UA server framework which allows a client to connect to the server and browse a custom Information Model.
- Connect to PIMAQ and present current and historical data in the Address Space of the server.
- Implement the methods necessary for an OPC UA client to be able to read current and historical data from PIMAQ.
- Try to model an ISO 15926 model in the OPC UA server Address Space and connect this model to data in PIMAQ.
- Evaluate the server for compliance using the OPC UA Compliance Test Tools.

1.1.5 Server Properties Specified by Siemens

Siemens specified some properties for the server before and during implementation. These were:

- Modularity, it should be easy to extend/modify the server.
- There should be as little configuration of the server as possible before booting.
- Ideally it should be possible to add/modify the existing Information Model in the server without having to recompile the server, just reboot it.

1.1.6 Acknowledgments

I would like to thank my fellow graduating class mates of MITK 2009 for the fun we have had and their support during these two years at NTNU. From Siemens I would especially like to thank Espen Breivik for his guidance, support and feedback, and the other people at Siemens OGTI in Trondheim for their help. Finally I would like to thank Professor Sverre Hendseth at NTNU for his guidance.

1.1.7 Audience

It is assumed the reader is familiar with the principles of object oriented programming and data structures like lists and trees.

1.2 Terms Used in the Thesis

Some of the terms used in the thesis are from the OPC UA specifications. These are written as they are in the specifications with big first letters as in Information Model, Nodes or Address Space. Software classes and functions are also written with big first letters as in the function “Read”.

Information Model

An Information Model is the data itself and how this data is built up. Or as can be deduced from the name it is a model of the information and the structure of the information.

Nodes and References

Objects in an OPC UA server is built up using Nodes. Nodes are the building blocks of objects. To relate Nodes to other Nodes they are connected using References.

Model Compiler

The Model Compiler is a tool that is part of the OPC UA SDK. It compiles Information Models defined in an xml file into different formats including a C# class definition file and a binary file with predefined Nodes and References.

Address Space

The Address Space is the collection of Nodes and References that an OPC UA server presents to clients.

NodeManager

When developers create their own Information Model they will also create custom Nodes and references that are not part of the OPC UA specification. To be able to handle/manage requests for these Nodes and References a NodeManager is created.

Namespace

This is a container for objects, classes and variables. In this thesis it is used in both the C# code to identify different data models and to identify Information Models.

Uniform Resource Locator (URI)

An URI is a string of characters identifying an abstract or physical resource on the Internet. The well known URL is an example of an URI, so `www.Google.com` is the URI identifying the resource Google.

Factories

Factories are objects in a programming language which has the job of creating other objects. A factory is usually used when creating the object directly is not an option, for instance there might be an identifier in the object that needs to be unique. One could pass into the object a unique identifier but if there are many it will be easier to create a factory which would create objects with unique identifiers automatically.

Wrappers

There are different uses of wrappers but in this thesis it is used as a function/-class which calls/uses another function in a different language. The purpose in this thesis is to call C++ functions using C# by creating a wrapper.

Tags

In relation to automation a tag is a unique identifier for an object or part of an object, generally associated with a process value. The identifier is usually descriptive so the location and function of the object is easily identifiable. As an example; to create a tag for an oil pressure transmitter on a generator, the tag could be called: `Generator1.Oil.PT`.

The Semantic Web

Semantics is the meaning of something. The Internet as it is now is a large collection of web pages. The information on web pages are exposed through keywords. Search engines use these keywords, usage patterns and clever algorithms to find the most relevant web pages but ultimately it is up to a human

to look at the web page and decide if it is relevant. The semantic web is technologies and methods which allows a machine to understand the meaning of a web page[8].

Ontology

Ontology comes from philosophy and is the science of describing things/objects and how they are related. Ontologies "...describe a shared and common understanding of a domain that can be communicated between people and heterogeneous software tools. We construct an ontology by defining classes of things, their taxonomy, the possible relations between things and axioms for those relations." [9]. This means that "...loosely speaking, collections of classes of objects such as entity-relationship models from the database community or object-oriented class definitions can be considered as ontologies." [9].

UTC

Coordinated Universal Time or UTC is time given from 00 to 23 in the Greenwich meridian. Local time is then given as UTC +- an offset. UTC is not adjusted for summer time so Norway has UTC+1 during winter and UTC+2 during summer.

Dictionaries, Key Value Pairs

A Dictionary is a data structure which maps one object to another object. In other words a Dictionary is a hash table which can only point to one entry per key. For instance the definition:

```
Dictionary<string, int>
```

creates a dictionary which can be used to map strings to integers, like a name to a social security number.

A Key Value Pair is a data structure consisting of key and a value. This makes it easy to collect two values into one object for later access.

Chapter 2

Background

2.1 Integrated Operations (IO)

This background chapter about IO is not directly relevant for the implementation of the server, but IO was investigated to understand the context of the thesis. The information in chapter 2.1 is taken from [10, 11, 2, 12] unless otherwise specified.

Integrated Operations is also called Smart Field, Field of the Future and E-Field by different companies. On the Norwegian Continental Shelf (NCS) Integrated Operations is used. As oil and gas fields produce less in the late part of their life cycle oil companies must run the day-to-day operations of the fields more efficiently, IO is part of this. The estimated gain of rapid introduction of IO on the NCS can be around 250 billion NOK from 2005-2015 according to [13]. Also "...OLF has estimated that the implementation of integrated operations on the NCS can increase oil recovery by 3-4%, accelerate production by 5-10% and lower operational costs by 20-30%." [14].

The Norwegian Oil & Gas Association defines IO as "...real time data onshore from offshore fields and new integrated work processes." [11]. IO consists of new work processes and methods for doing oil and gas exploration and production with the main goal of creating better planning and decision processes. The ideal result is that exploration will be more effective, better reservoir utilization,

increased production, less downtime, less interruptions, more efficient operation and project implementation and reduced HSE risk[15].

This is possible because of new IT technology. New technology allows real time communication and data sharing between offshore and onshore which allows personnel onshore to have access to the same information as those offshore. The consequences of this is that onshore personnel can advise/support those offshore, equipment and processes can be remotely controlled, and more personnel can be moved onshore. This results in a change of work methods. By reducing cost it can also enable oil & gas extraction from fields which were not economically feasible. More upsides is that IO should result in cooperation and closeness between groups which earlier had no contact, especially within the same organization, and that sharing of HSE information between companies may increase the overall understanding of risk and HSE within the petroleum business [16].

Traditionally onshore and offshore was regarded as separate units. The day to day operations were handled offshore with limited support from onshore. Most decisions were taken by operators without or with limited engineer support.

Generation 1 of IO is now underway and involves creating onshore centers with 24/7 communication links to several offshore control rooms. This allows companies to have experts on land advising several platforms, instead of having to send one expert to each platform. Having onshore experts allows better optimization of production and provides extra support during problems.

Generation 2 is the next step and involves integrating suppliers and having expert centers around the world to allow 24/7 support without having night shifts. Suppliers will also take over part of the job of advising operators offshore as they are more familiar with their own equipment.

2.1.1 Perspectives on Accidents

While these perspectives/views on accidents are not directly relevant to the server implementation, they are interesting and educational. There are different perspectives on risk management and major accidents described in [15], below is a summary of these:

- Energy barrier perspective

- This involves preventing accidents by focusing on dangerous amounts of energy (like pressure, weight and heat) and creating barriers to separate these from humans, equipment and the environment. Safety strategies involves reducing the amount of energy, create barriers or handling the dangerous situation (evacuation).
- Information processing perspective
 - This perspective identifies the root cause of major accidents, as not the technical system itself, but as a result of lack of information flow and/or misinterpretation of events. This hindered identification of problems. In other words the accident is a result of organizational problems. To combat these problems an organization must be able to utilize information, observations and ideas from everyone in the organization without regard of a person's/group's status.
- Decision perspective
 - Risk management in this perspective is handling three different goals and trying to keep the company within three different boundaries. If a boundary is exceeded there will be an accident. The management wants better efficiency and to avoid their boundary of financial bankruptcy. Workers would like a comfortable workload and their boundary is an uncomfortable workload. The safety management wants to reduce risk and they have an unacceptable risk boundary. Choices will affect one or several of these goals. Lack of awareness of this will result in accidents.
- Normal-accident perspective
 - This perspective claims that the barrier perspective will not detect some accidents as it does not take into account the interaction of active and latent errors in complex systems. In a tightly coupled system disturbances and unintended side effects can spread and escalate. This makes it impossible to attain complete knowledge of the system to prevent accidents.
- High Reliability Organization (HRO) perspective

- HRO is partially intended as an answer to the Normal-accident perspective and tries to explain how an organization who has Normal-accidents conditions does avoid accidents. The central elements here are organizational redundancy and the ability to change operation modes during crisis. Organizational redundancy implies that tasks and competence is overlapping within the organization and there is a culture for exchanging information and overseeing each other. Some of the theory comes from studies of the military, which may go into more adaptable and flexible operation modes if there are deviations from the norm.
- Resilience engineering perspective
 - This is a new perspective based on theory from cybernetics and biological systems. It has two premises; Systems are nonlinear and dynamical. The properties of the system vary naturally which is a premise for learning and development but also a source of unwanted deviations and events. Based on this, errors are not the collapse or faults of normal system operation but unintended and coinciding effects of the adaptive processes which are needed for a system to work in a complex environment. Resilience engineering then hopes to provide models and tools to understand, monitor and handle system variations.

2.1.2 Risks and Rewards by Implementing Integrated Operations

The offshore industry has some of the highest security requirements in the world and there has been research into how the introduction of IO will affect security. The SINTEF report [15] uses the accident perspectives to create questions relevant to each perspective. As none of the perspectives tells the absolute truth the hope is that by using all six perspectives it will give a varied and good insight into the risk factors for major accidents with regards to the implementation of IO. To answer these questions five scenarios are created which are analyzed to find the changes that IO implies. These changes are then collected to find the answer to the original questions. As one may expect there are both positive and negative effects to all changes and such no clear answers to the questions, however these isolated changes seem to have a positive effect:

- Better planning/support.
- Increased use of IT.
- Better possibilities for simulation and training.

The negative effects are more generic and are heavily connected to:

- Complex structures, like integrated suppliers with limited responsibility.
- Lack of understanding of the implications from the new decision context like different situation understanding, lack of hands-on knowledge, group based decisions can blur the responsibility for executing the decision and efficiency/cost reduction dominates at the cost of safety.
- Lack of focus on work methods compared to work efficiency.

[16] describes three different situations offshore and the generic elements which may contribute to a major accident:

- Though decisions can be taken remotely it may not be the best option as efficiency and cost is a driving factor in most contracts. The result may be that decision makers may not have the necessary prerequisites to make their decision, i.e. they may know simple components intimately but may lack the overall systems knowledge.
- An operator may feel that something is wrong but he could simultaneously feel that it is not important enough to share.
- By creating expert centers companies will be more vulnerable to personnel changes which could result in loss of key competence.
- Any uncertainties with regards to responsibility and decision boundaries between operator, suppliers and sub suppliers.
- Electronic communication can hide underlying information.
- There could be a lack of personnel to handle emergencies as the current personnel situation is already minimal/cost effective.
- The dependency on the IT structure increases, and only the possibility of vulnerability to hacking, viruses and denial of service attacks can introduce uncertainty whether the deviation is as a result of this or equipment/operator fault.

2.2 ISO 15926

The information in this chapter was not used directly in the implementation, as a finished ISO 15926 model was used, but ISO 15925 was investigated as part of understanding the context of the thesis. ISO 15926 is large specification involving many concepts and can easily compromise enough work for a master's thesis by itself so the information in this chapter tries to give a general overview, describe why the specification exists, how it is implemented and the advantages/disadvantages of the specification. Unless otherwise specified the information in chapter 2.2 is from [17, 4, 18, 19, 14].

2.2.1 Why ISO 15926 Exists

ISO 15926 is a specification in eleven parts for digital interoperability for process plants and facilitates exchanging complex plant information. Digital interoperability is "...the ability of different types of computers, networks, operating systems, and applications to work together effectively, without prior communication, in order to exchange information in a useful and meaningful manner"[20].

"...In the US capital facilities industry in 2002 alone, NIST estimated the annual cost of poor interoperability –the cost of finding and verifying (& the cost & risk of not finding) correct information for operational decision support –at USD15.8 billion–over and above wider health, safety and environmental risks." [21]. As seen the lack of digital interoperability has costs [9, 21] and this is the background for the ISO 15926 specification.

On the NCS there has been a trend for the last years that there are more small oil and gas fields, and more specialized service companies. This makes coordination and collaboration more important[14].

The hope is that ISO15026 can reduce the cost of retyping and reformatting data when moving it from one system to another. As an example when designing, specifying and purchasing an instrument the following operations could be done:

1. After the design the information is entered into a spreadsheet or database.
2. The officer in charge of purchases assembles the information and sends to the suppliers.
3. Each supplier will enter the information into proprietary software and create an offer based on this.

4. During design some properties of the instrument may be supplier specific and is not taken account of. After a supplier is chosen the supplier data will have to be manually entered into the CAD program.
5. After receiving documentation from the supplier important values will have to be entered into an asset management system.

The information above is entered and converted into many different systems. Not only does this take time but it also introduces the element of human error in each operation. What is needed and what ISO 15926 should provide is a way for each participant's software to communicate complex information to another participant without having to know the receivers data structure/format. Now doing the same operations above with ISO 15926:

1. After design the information is entered into a spreadsheet or a database.

With ISO 15926 tools extracting the information automatically, the next operations may be:

1. The purchase officer will use a public interface for the company called a façade to create purchase request and expose it via the company's website. The URL can then be sent to the suppliers via email.
2. Suppliers can connect to the façade and retrieve the information for each instrument. The suppliers can now choose to enter the data manually or, since ISO 15926 data is rich enough, they can use a software program to find the relevant instruments and let their engineers review the bid before sending it.
3. After a supplier is chosen the design engineer can use his CAD system to connect to the supplier's façade and retrieve the data automatically.
4. The asset management system can connect to the supplier's façade and retrieve the interesting values.

As seen the process should now be faster and less prone to human errors. If everyone uses the same standard, then information can be exchanged without having to know of each other's data format, information will not have to be re-typed and data has higher fidelity as there is less human involvement. Everyone can still use their own proprietary format or storage format and information will be exchanged via a façade.

2.2.2 Upper Ontology

The knowledge in chapter, 2.2.2, is taken from [9]. ISO 15926 is an upper level ontology. Ontologies can be developed either from bottom-up or top-down. Bottom-up development means that one starts by defining the most relevant concepts for the area where the ontology is to be used. This results in ontologies which are hard to modify and integrate with other ontologies. The top-down design starts by defining high level concepts, which most likely are relevant for many areas. Engineers using top-down design will know where the ontology will eventually be used. This knowledge may influence the high level design in a negative way as the engineers may create high level concepts which corresponds to or leads faster to the final use. Upper ontologies avoid this problem by defining the top/high level classes first and engineers can then use these classes to create the more specific classes for their need.

2.2.3 A General Overview of the ISO 15926 Specification

A short summary of the eleven parts of the ISO 15926 specification taken from the “How Does ISO 15926 Work?” part of [17] is below:

- 1 Overview and Fundamental Principles
 - The title is fairly self explanatory.
- 2 Data Model
 - This is the foundation of the specification and creates the grammar for ISO 15926 and “..is akin to an Upper Ontology”[19]. It consists of the rules and constraints for using ISO 15926. Part 2 requires a fair amount of work to understand, but fortunately most organizations will only have to deal with part 7.
- 3 Reference Data for Geometry and Topology
 - Part 3 is still under development and will eventually used to represent 3D CAD objects.
- 4 Reference Data Classes

- Similar to a dictionary or thesaurus, it defines types of entities and their relationships. An entity can be for instance an “Inanimate physical object” as seen in figure 3.9 on page 39. Its parent may be “physical object”. How an entity in part 4 is related to part 2 can be seen in figure 2.3 on page 20.
- 5 Registration Procedure
 - “..Procedures for registration and maintenance of reference data.” from “How Does ISO 15926 Work?” part of [17].
- 6 Reference Data Additions
 - The requirements when creating additions to ISO 15926.
- 7 Templates
 - A template is “...a pattern for stating facts.”[20] or it can be thought of as a small ontology. An example of a pattern can be “The ambient temperature during operation of a 3051CG pressure transmitter should be within -40 and 85 degrees Celsius”[20]. Figure 2.2 on page 20 shows how this can be represented in an ISO 15926 model. Templates can be the building blocks of what one needs to represent. For example if one needs to create a template of an engine, the template can be created by combining templates of cylinders, generators, pressure/temperature transmitters and so on. This is the most important specification for most users as it can be said to encapsulate part 2.
- 8 RDF/OWL Implementation Specification
 - This part shows how to implement part 7 using Resource Description Format and Web Ontology language (OWL). RDF and OWL are technologies that have grown from the concept of the semantic web. “...XML allows the creation of structured data but the data has no meaning. The meaning is given by RDF. RDF asserts that objects have properties which have values, this called a triplet. Formally objects are called subjects, properties are called predicates and values are called objects. An example is “Alice is the sister of Bob” or “Shakespeare is the author of Hamlet”. In these instances the subjects

are “Alice” and “Shakespeare” who have the predicates “sister of” and “author of” and the objects have the values “Bob” and “Hamlet”. By using RDF one can relate data to other data and convey meaning.”[6]. OWL is a language that uses RDF to create ontologies. For example an ontology describing the wines of the world.

- 9 Façade Specification
 - As mentioned earlier a Façade is the public interface of a company’s ISO 15926 repository. Other companies can use the Façade to retrieve information. A company can choose which parts to make public.
- 10 Abstract Test Methods
 - No information is available yet.
- 11 Simplified Industrial Usage including Gellish Implementation using Reference Data
 - “...Part 11 was proposed recently as an easier methodology to implement parts 7 & 8.” from “How Does ISO 15926 Work?” part of [17].

A comparison between the first nine parts of the ISO 15926 specification and natural language can be had from figure 2.1 on the next page.

2.2.4 How ISO 15926 Will Work

ISO 15926 will ultimately provide a set of public reference data, or a Reference Data Library (RDL). An RDL is implemented as a Reference Data System/-Work In Progress(RDS/WIP). An organization can then map their data and applications to the RDL. Before an organization sends ISO data it can validate the data against a RDS/WIP and then the receiver can also validate the incoming ISO data against the RDS/WIP. The RDL used in thesis is part of the Integrated Operations in the High North (IOHN) project and is created by Siemens in collaboration with other companies. The RDL uses part 4 of the ISO 15926 specification as a base and then extends those entities to model a oil platform. Ultimately the goal of the project is to create an RDL for the oil and gas industry that is good enough to be standardized into the ISO 15926 specification.

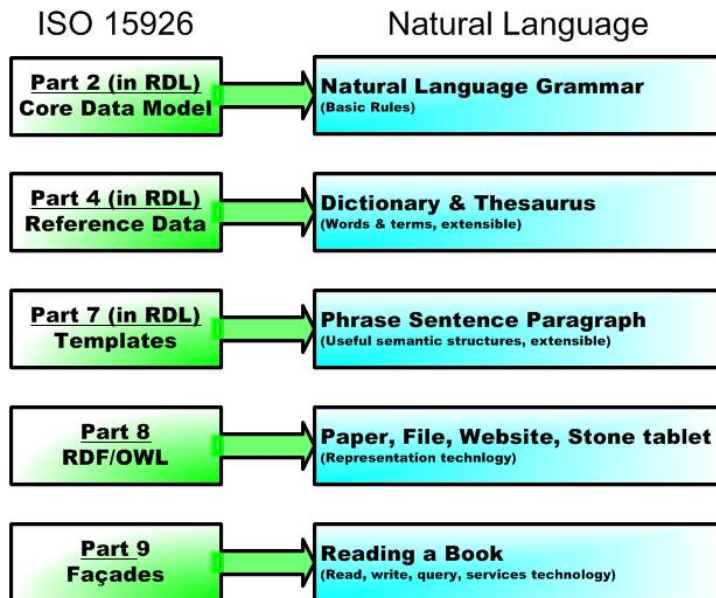


Figure 2.1: Comparison between language and the parts of the ISO 15926 specification, from [17]

2.2.5 Advantages and Disadvantages

The advantage as written earlier is that the specification should result in less time and money spent on retyping/interpreting data by allowing computers to do more of the work.

The article [22] discusses some issues with the ISO 15926 specification. While the author does not say whether the specification itself can or cannot fulfill what it was made to do, he does argue that there are problems with calling the specification an ontology. His main issues are:

- The specification is not very intelligible or easy to understand.
- It is not an open specification and as such there are costs and compromises when creating the specification.

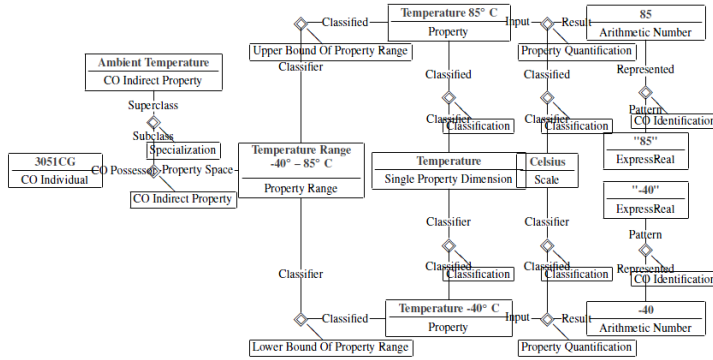


Figure 1: Range assignment, ISO 15926 model

Figure 2.2: ISO 15926 model of ambient temperature assignment, from [19]

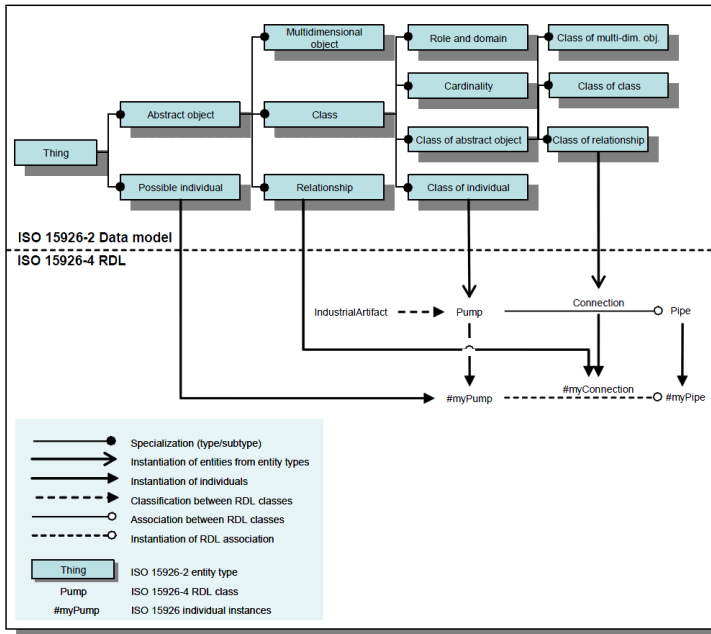


Figure 2.3: The relationship between ISO 15926 part 2 and 4 taken from [14]

- The author takes issue with the mathematical set theory that was used when creating the specification.
- It does not reuse available and good resources/ontologies.
- There is no clear way to distinguish between types and instances of types.

2.3 OPC UA SDKs

The OPC Foundation provides SDKs in Java, C# and C/C++ for server and client development. Understanding the SDKs is an important step before implementing. The source code and binaries for the SDKs are covered by different licenses. This must be kept in mind when implementing an application, as wrong use of licensed code is morally wrong and can be expensive.

2.3.1 Licenses

The information about licenses is taken from [23]. As the server prototype may be part of a commercial solution license usage must be investigated. The OPC Foundation uses a dual licensing model where users can choose between and Commercial license or a Reciprocal Community License. With the Commercial license the user can pay and then use the software without having to share their code with the community. The Reciprocal Community license allows users to use the software free of charge but the users must pay the community back by sharing their bug fixes and changes. To put it succinctly; developers pay in either cash or code.

2.3.1.1 OPC Redistributables License

This license applies to any prebuilt binaries and binaries compiled from distributed source code. It grants users the right to distribute and redistribute these binaries without royalty fees. Users can not redistribute without adding significant functionality.

2.3.1.2 MIT License

The MIT license is an open source license, originating from Massachusetts Institute of Technology, and allows users “..to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sub license, and/or sell copies of the Software.” subject to including the license in all copies or portions of the software. The author(s) is also not responsible for any damages occurred by using the software and the software is given without any warranty.

2.3.1.3 Reciprocal Community License ("RCL")

Based on the principle of fairness and the Open Source RPL license the RCL license closes some loopholes in the open source license which allowed companies to use open source code and gain financial benefit without having to share their improvements or bug fixes with the community. In this case the community is the OPC Corporate Members so it is not full open source. In essence this means that any bug fixes or changes to the original modules like the SDK, stack or wrapper must be shared with the community even if you deploy for internal testing. However any proprietary source code you create using the original modules is your own intellectual property and must not be shared. The RCL is meant to be commercially friendly.

2.3.1.4 Reciprocal Community Binary License ("RCBL")

The RCBL license is the same as the RCL only binaries can be shared, no source code.

2.3.1.5 Commercial Source Code License ("CSCL")

If the RCL or RCBL is not acceptable the OPC Foundation offers this license for a fee which removes any requirements to share modifications or enhancements. It does not add support or warranty.

2.3.2 DLLs

The OPC Foundation delivers four DLLs with the SDKs, these are:

- Opc.Ua.Client.dll
 - This DLL contains classes which are used to implement UA clients. This includes classes for managing and storing sessions, subscriptions and browsing the server model.
- Opc.Ua.Server.dll
 - Defines the classes which can be used to implement a UA server. Among these classes is the NodeManager which manages the custom Nodes developers uses to build their Information Model. As with the client DLL, the server DLL manages and stores session's data and subscriptions. The server is where a developer would access data and events in an external system, like a database, controllers and/or measurement devices.
- Opc.Ua.Configuration.dll
 - This DLL manages the configuration and security settings for a UA application. The classes here do the initial setup based on XML files and certificates.
- Opc.Ua.Core.dll
 - Defines the classes which implement all of the services and types in the UA specification.

2.3.3 Programming Languages

SDKs are offered in C/C++, C# and Java. As Siemens has no code in Java this SDK has not investigated. The C/C++ SDK consist of the communication stack and a simple server/client Visual Studio solution. The solution is for a server that does simple setup and start/stop of the server with no data presented to clients.

The C# SDK contains nine solutions showcasing different aspects of a server, a closer look of these are in section 4.2 on page 44.

Chapter 3

Server and Information Model Design

Implementation and design was done in two stages. Stage one, called the “framework server”, created/designed a framework for the server. In stage two, the “prototype server”, the framework server was extended to interact with PIMAQ and create/present Information Models based on PIMAQ data and/or an ISO 15926 model.

3.1 The Design of the Framework for the UA Server

There were no real design choices for the framework as the basic structure of an OPC UA server is already fixed. Most of the effort for the framework went into investigating all of the SDKs and how to configure the server. The server architecture in the C# SDK is as seen in figure 3.2 on page 27. All of these classes/interfaces are implemented in the OPC.UA.Dlls and/or in the SDK. The framework server structure is as seen in figure 3.1 on the following page.

One of the first goals of the thesis was to create a framework for the server which would configure the server with a certificate, define communication methods, define the security and present a Information Model in its Address Space. An

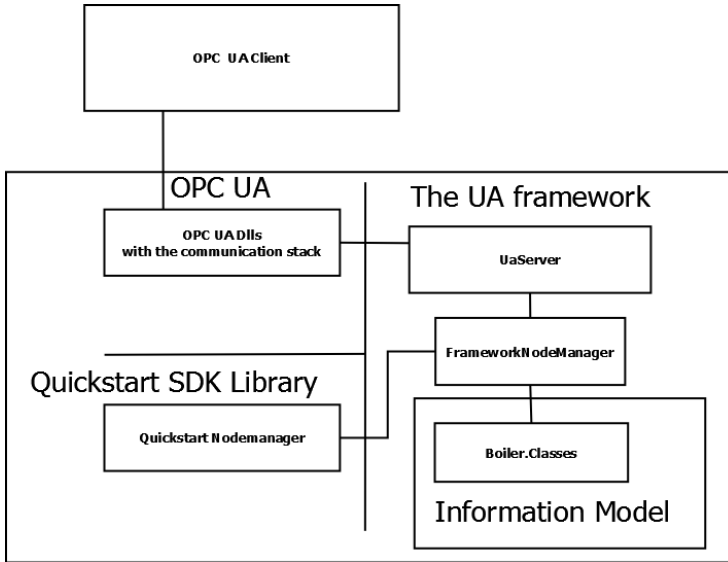


Figure 3.1: The structure of the OPC UA Framework server

implementation inherits from `StandardServer` to create a server class and inherits from `INodeManager` to create at least one `NodeManager` for use in the server class. A `NodeManager` has the job of managing the custom `Nodes` created in an `Information Model`.

The framework for the prototype server consist of several classes with different functions, these are described in section 4.3.1 to 4.3.2 . The framework server is simple with an `Information Model` of a boiler. Objects and `Nodes` were exposed to clients without reading and writing of data, but the model could be browsed. The server was tested with two clients, the UA SDK `DataAccess` client and a commercial client called OPC UA viewer [24].

3.2 The Design of the Prototype UA Server

The prototype server adds more classes to the framework server to connect and read from the data source(s) and expands the `NodeManager(s)` to handle read

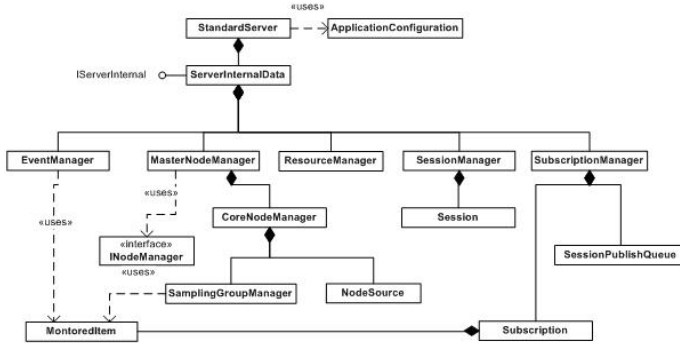


Figure 3.2: The classes in the server architecture according to [25].

and historical read requests. The full structure of the prototype server can be seen in figure 3.3 on page 29.

For the prototype there were not many design decisions to be made. This was because the PIMAQ structure is fixed and the NodeManager/UaServer structure is the same as in the framework server. The design decisions dealt then with the best method for the OPC UA classes to interact with PIMAQ.

The class DynamicProcessAPIwrapper was designed to mirror the functions in pInvokeInterface directly and convert data from DynamicProcessAPI into the data structures used in the server. It is a static class to make sure it is the only method for using DynamicProcessAPI.

The class DynamicProcessAPIinterface is intended to be a interface for classes that needs to interact with DynamicProcessAPI. DynamicProcessAPIinterface consists of simple or complex functions which utilize the functions in the wrapper to build functionality for other classes.

Modularity is important and this design should make it easy to extend or modify the server as both classes have clearly defined functionality.

The functionality for the server classes are as follows:

- The DynamicProcessAPIwrapper has the functions for communicating with PIMAQ.

- `DynamicProcessAPIinterface` uses the wrapper to build more complex functions for managing tags and initializing the `DynamicProcessAPI`.
- The `UaServer` has three `NodeManagers` which it calls in turn to fulfill requests from clients, it also initializes `DynamicProcessAPI` through the `DynamicProcessAPIinterface`.
- For data requests each `NodeManager` will use `DynamicProcessAPIinterface` to retrieve data if they need. Each `NodeManager` is inherited from the `Quickstart NodeManager` and consequently some function calls are handled by the functions in the `Quickstart NodeManager`.

3.2.1 The PIMAQ Structure

PIMAQ is a C++ application which consists of many modules. Each module is compiled as a DLL and then loaded by the program using this DLL. This makes PIMAQ modular and customizable. For the purpose of this thesis the modules used were:

- `SiLight.dll`
 - This is a support library which contains data structures like time/date structures and read/write mutexes/semaphores.
- `ProcessDataUtilities.dll`
 - This module contains among others the two main classes for connecting to data. `DynamicProcessAPI`, which is the API used to connect and read/write to the different data providers, and `pInvokeInterface` which is intended to be used by a wrapper class.
- `Staticprovider.dll`
 - This module is responsible for loading/providing csv files, these look like in listing 3.1 on page 30. CSV files are an easy method to test read/write of values in `DynamicProcessAPI` as they are local files requiring no network configuration or external database.
- `SiemensIp21ProviderRemote.dll`

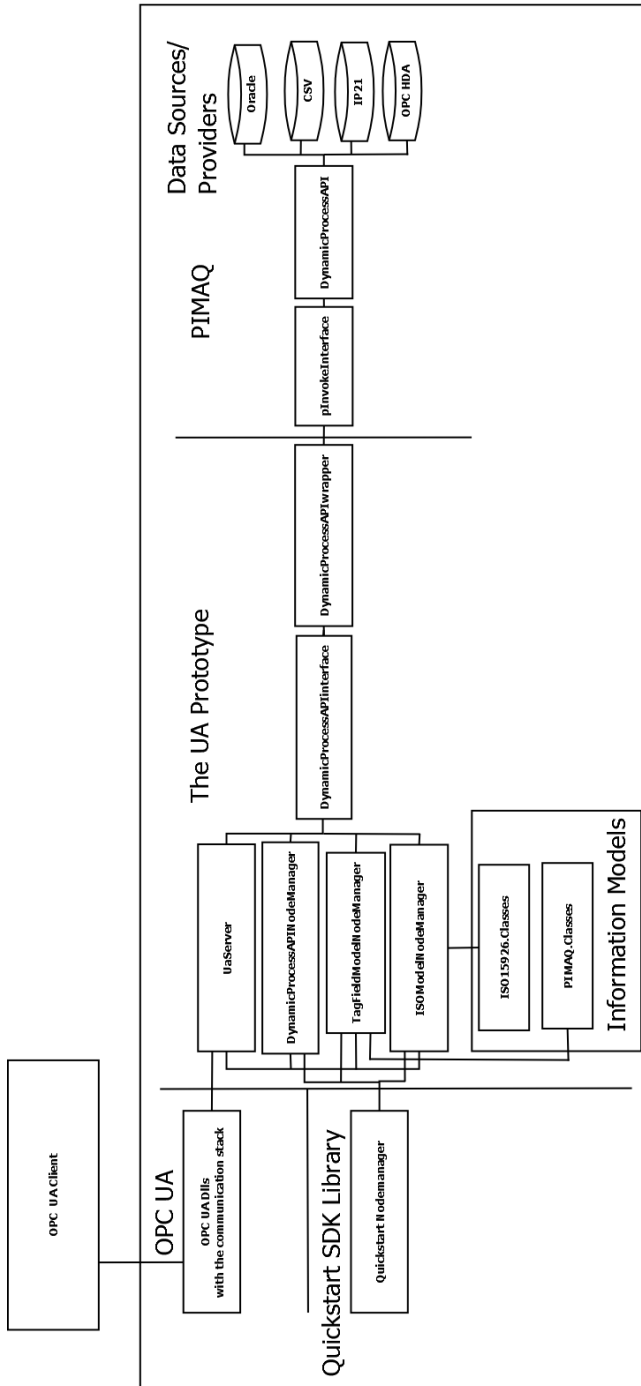


Figure 3.3: The structure of the OPC UA server prototype

- InfoPlus21 is a historical database made by Aspentech and this module connects to such a database for reading/writing. This provider connects to a remote machine on the network hence it is called remote.
- IP21FilterLib.dll
 - The API for the IP21 database from Aspentech is complex and not easy to use. Siemens has therefore created a wrapper class for the IP21 API, called IP21Filterlib for doing calls into IP21.
- Assorted DLLs made by Aspentech [26] for connecting to their InfoPlus21 database.

As seen in figure 3.3 on the previous page DynamicProcessAPI can connect to four different data sources, an Oracle database, csv files, an IP21 database and a OPC HDA server. The classes responsible for connecting to each data source are called providers and are dynamically loaded by DynamicProcessAPI on start up.

Listing 3.1: Example CSV file

```

1 #Tag, Field, Value, Timestamp
2 MLS-LIC-113, "FBTypeName", "CA"
3 MLS-LIC-113, "AutoMode", 1, 2011-01-07T00:00:00.000+02:00
4 MLS-LIC-113, "PropGain_Kp", 0.1, 2011-01-07T00:00:00.000+02:00
5 MLS-LIC-113, "IntegrTime_Ti", 0, 2011-01-07T00:00:00.000+02:00
6 MLS-LIC-113, "DerivativeTime_Td", 0, 2011-01-07T00:00:00.000+02:00
7 MLS-LIC-113, "IntSetpoint", 5000, 2011-01-07T00:00:00.000+02:00
8 MLS-LIC-113, "AnalogInputValue", 0, 2011-01-07T00:00:00.000+02:00
9 MLS-LIC-113, "ControlOutput", 0, 2011-01-07T00:00:00.000+02:00

```

3.3 Designing Information Models

To present data in the Address Space for clients, the data must be modeled. These models are called Information Models in UA. One of the strengths of UA is that it can model complex relationships and models, according to [6]. Two Information Models have been developed, the first to model/mimic the data structure in PIMAQ and the second to model an ISO 15926 model of the oil rig Snorre B.

The different Nodes in the OPC UA Information Model have normal definitions and type definitions. Type definitions are meant to be created for Nodes that are reused, for instance in a library. Complex objects have mostly been created as types, and then an instance of the type has been created and loaded into the server from a binary file.

3.3.1 Creating an Information Model

Information Models are created as XML files, and then compiled with the tool Model Compiler. The Model Compiler outputs a binary file; `NamespaceName.PredefinedNodes.uanodes`, which contains predefined Nodes and the `NamespaceName.Classes.cs` file which contains the C# class definitions for Nodes and References. Editing an XML file by hand can be hard and prone to errors. There is a solution to this, a free software tool called CAS UA Address Space Model Designer from [27]. This software allows an easier GUI creating and editing of Information Models, and then it compiles the models into the same files as from the Model Compiler. It will also refuse to compile on errors in the model. The full version includes options to verify the model. Figure 3.4 on the following page shows how the Information Model for the framework server looks in the CAS Model designer. As seen the model has two custom References to describe signal and fluid flow, type definitions for each part of the boiler (controllers, sensors, actuators, indicators, transmitters, pipes and the drum) and a predefined boiler1 which is the same as seen in figure 4.1 on page 47.

3.3.2 Description of Tag.Field Model

Data in `DynamicProcessAPI` are organized first under function block type, then tag name and then fields. Function block types are mostly from the NORSOK I-005 standard [28]. A control system for a process is mostly comprised of eleven different functions which was standardized in NORSOK I-005. Examples of the functions standardized are:

- MA, Monitoring of Analogue variables.
- MB, Monitoring of Binary variables.
- CA, Modulating control/PID controller.

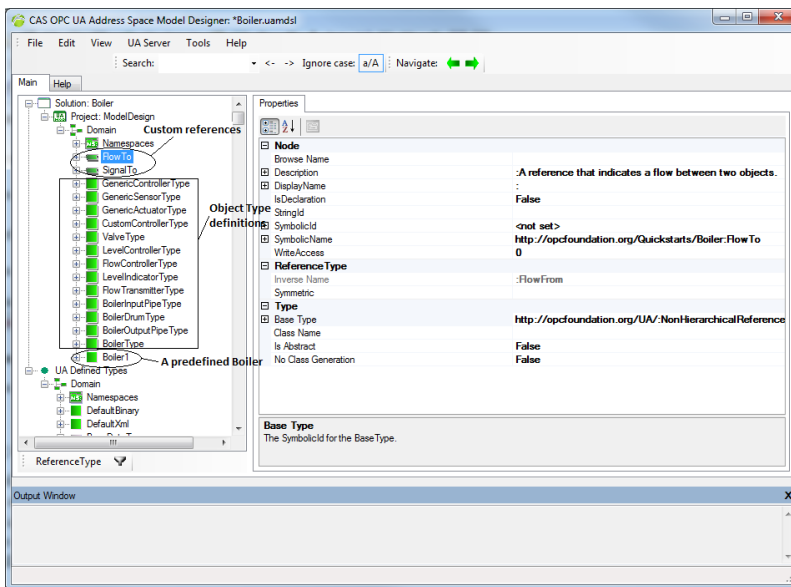


Figure 3.4: The Framework Information Model as seen in CAS UA Address Space Model designer

- CS, Step control of equipment.
- SBE, Binary control of electrical equipment.
- SBV, Binary control of pneumatic/hydraulic equipment.
- QA, The total amount of process values for a time interval.

These functions, or function templates as they are called in the standard, are associated with a tag in the system. The control system is then mirrored in the historian, normally an IP21 database. As seen in figure 3.5 on the following page each function template have several inputs and several outputs. These inputs and outputs are historized as fields under each tag along with other fields which comes from IP21 settings and other fields which describe the tag. Figure 3.6 on page 35 shows an MA function block from the control system from Snorre B modeled in CAS OPC UA Address Space designer. By comparing both figures referenced above one can see that the MA model mirrors most of the inputs and outputs for the function templates, with extra fields which describe both the MA block further and historian settings.

What all of the above means is that all tags in the system are organized under function blocks and that each tag have fields which both describe the measurements and are the measurements themselves. This means that the fields are uniquely tied to the function blocks and not to the tags themselves and that all function blocks of the same type have the same fields for a given process plant. These assumptions are used heavily when dynamically creating the Address Space for the server and are valid according to Siemens.

3.3.2.1 Advantages of the Tag.Field Model

Normally a control system will have tags for every option and measurement so around 50-60000 tags for an oil rig. The advantage with the Tag.Field model is that the amount of tags are reduced and then options and measurements are collected as fields/properties of the tag and then every single field is historized. The model also mirrors the control system.

3.3.2.2 Different Methods for Creating the Model

Two methods for creating the Tag.Field model have been explored. The first was a dynamical method where a FunctionBlockType Node, a TagType Node,

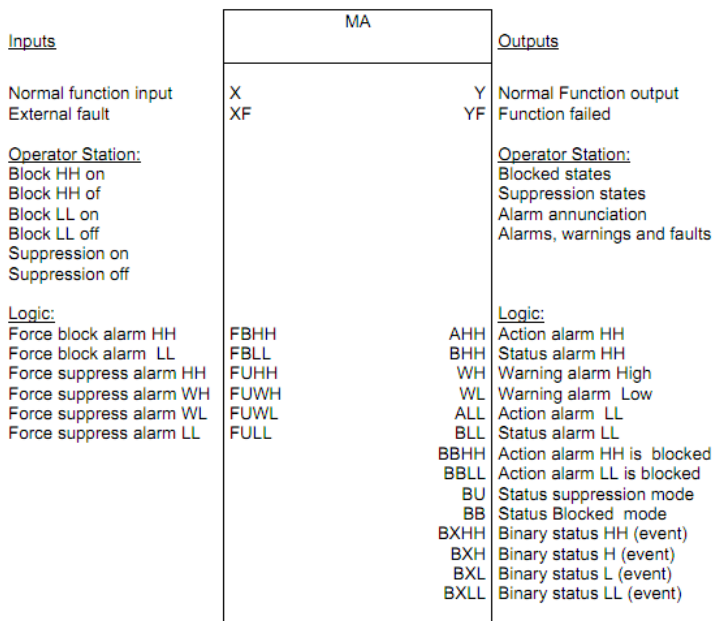


Figure 3.5: The MA function template from the NORSOK I-005 standard[28]

a `StringFieldType` Node, a `DoubleFieldType` Node, a `FloatFieldType` Node and a `StatusWordFieldType` Node was created in the Information Model as seen in figure 3.6 on the preceding page. Servers can then connect to PIMAQ, get the tags, iterate the tags, connect them to function blocks, and create fields based on the tag's function block type. The advantage here is that the model makes no assumptions apart from that all function blocks of the same type have the same fields and requires no configuration and will adapt to any data source dynamically. The disadvantage is that it does require time to create for the server.

The second method is to model the function blocks exactly as seen in the data source. An example of this is seen under `MATagFieldType` in figure 3.6 on the previous page. Of course this means every function block type must be modeled from the data source first. The advantage is that it should be faster to create the model for the server but it is not flexible as only one data source is modeled and hence more configuration is needed. More work is also required to use the second method.

Both methods have advantages and disadvantages so they were both tried and tested. In the end the dynamical approach was selected for the prototype as it was more in line with Siemens wishes for as little configuration as needed.

3.3.3 Description of the ISO 15926 Model

As described in the background chapter about ISO 15926 public RDLs should be created so that companies can share data according to the data models. Siemens is part of the IOHN project and have used the POSC Caesar RDL to create a ISO 15926 model for the Snorre B oil platform. It is a large model as seen in figure 3.7 on the facing page. The figure has no details but green color means a static element like flow lines, risers and joints, gray is active elements like valves and pressure elements, and purple are properties of elements like pipe inner diameter and pipe wall thickness. Because of time constraints the entire model has not been modeled for the UA server but oil well C6 (figure 3.8 on page 38) and C5 has been modeled to show the concept. The objects which the wells consists of are defined in the nomenclature in figure 3.9 on page 39. As seen Inanimate Physical Object is from ISO 15926 specification 4 and then has been further refined into different objects corresponding to platform equipment by the IOHN RDL project. To create an OPC UA Information Model the ISO

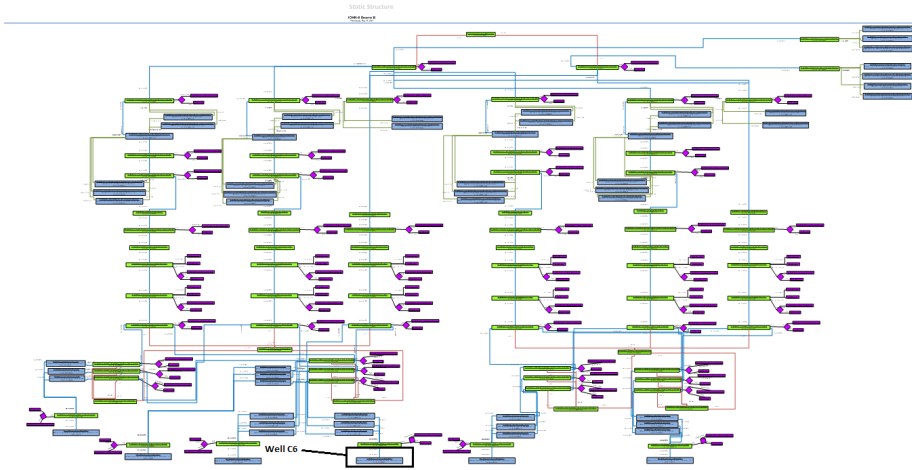


Figure 3.7: The ISO 15926 Model for the Snorre B platform[29]

objects become Nodes and the relations between objects are created as non hierarchical References.

OPC UA clients can browse a servers address space and filter away the References it is not interested in. Therefore the References for the ISO model is collected into a master Node object `ISONonHReferenceTypes`, so clients can view the ISO model by choosing to view `ISONonHReferenceTypes`.

The assumption for the ISO 15926 Information Model is that all Displaynames for Nodes are unique, which is OK as all objects in the original model have unique identifiers.

3.3.3.1 Different Methods for Creating the Model

First the basic Nodes and References of the model were created by converting RDL objects into Nodes and RDL relations into References using the nomenclature. This resulted in the objects above the WellType definition in figure 3.10 on page 41.

References can have an inverse name or be symmetric. An example is the `hasPart` Reference which is not symmetric and has the inverse name `isPartOf`

IOHN-6 Snorre B – Well C6

Wednesday, May 04, 2011

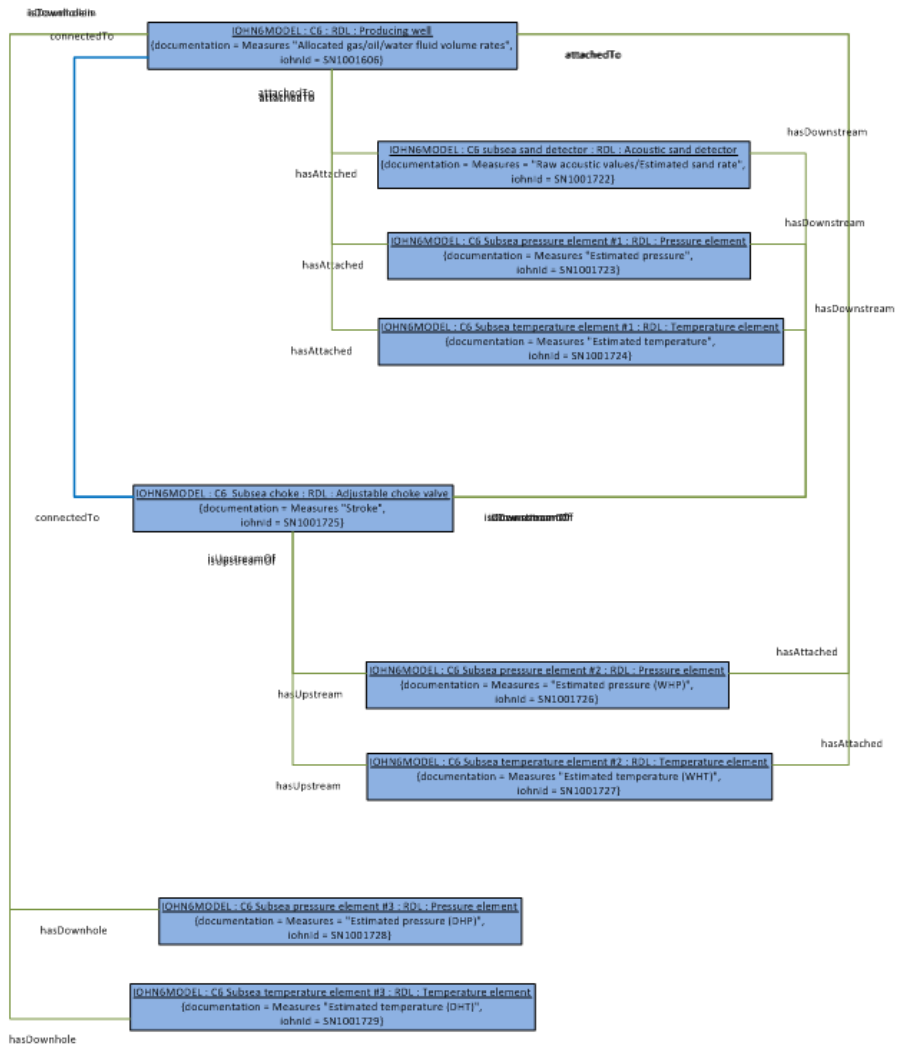


Figure 3.8: ISO 15926 Model for Well C6[29]

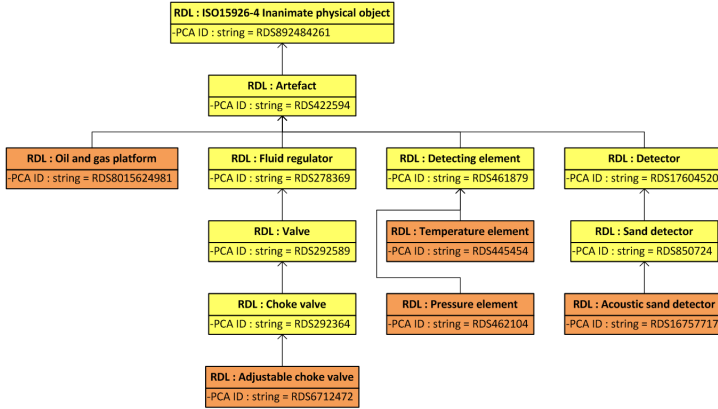


Figure 3.9: Nomenclature for the elements in well C6[29]

(Object A hasPart Object B, Object B isPartOf Object A). An example of a symmetric Reference is the connectedTo Reference (Object A is connectedTo Object B, Object B is connectedTo Object A). The CAS OPC UA Address Space designer does not support inverse names at the moment so it was hard coded into the model by editing the xml file.

Again two slightly different methods for creating the model were investigated, created and tested. The first consists of creating a PlatformType type definition and then add Nodes and References to mirror the ISO model as seen under PlatformType in figure 3.10 on page 41. Each Node references its type definition. In the end a Snorre B object is created which has its type definition from PlatformType. This method involves mirroring the entire ISO 15926 model in the Information Model. The advantage is that the model needs no processing it just have to be loaded by the server, the disadvantage is that it is static and harder to change.

The second method is similar to the first, but instead of creating a specific model the original model can be divided into general parts which are modeled by themselves as seen in WellType in figure 3.10 on page 41. The complete platform can then be created by putting together the general parts as seen under Snorre A in figure 3.10 on page 41. The advantage is that the model is more flexible and easier to change but with the disadvantage that the server must process the model to set the correct identifiers for the Nodes. This is

because the general model does not have unique identifiers.

The first method is the one that is used. Even though it needs more modeling and is less flexible, an oil platform is a fairly static structure so the extra effort is worth it. The second method, while more flexible, will add more configuration to the server to determine identifiers for Nodes.

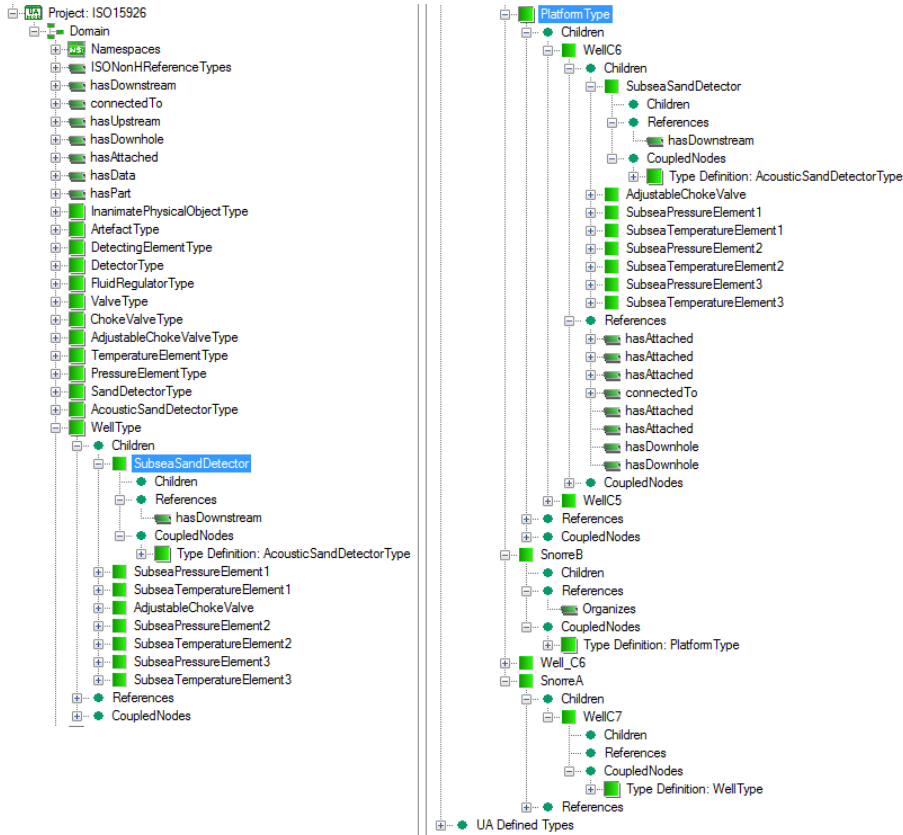


Figure 3.10: The complete ISO 15926 Information Model

Chapter 4

Implementation

The implementation part of the thesis has created two servers called the framework server and the prototype server.

The framework server has:

- Initial configuration of the server.
- An Information Model of a boiler.
- The possibility for clients to browse the Information Model.

The prototype server extends the framework server and adds:

- The possibility to retrieve current and historical data from PIMAQ.
- Two Information Models, the Tag.Field model and the ISO 15926 model.
- Information Models that can be browsed, and current and historical values can be read for the the Nodes in the model.
- Caching of historical request to minimize the load on PIMAQ.

Chapter 4 gives an overview of the two servers and the most important classes and functions for each. It also explains and discusses the problems encountered during the implementation and their solutions.

4.1 Choosing the Programming Language

Originally the plan was to create a server in C++ as most of PIMAQ is programmed in C++. But as the SDKs were investigated it was found that C++ is a harder starting point as there are no proper examples to work with. The only proper C++ SDKs are commercial and costs money. So in the end the choice fell on using the C# SDKs and creates a wrapper for communicating with PIMAQ. The knowledge gained during the C# implementation also works a starting point for a future C++ implementation as they both use the same functions in the OPC Dlls.

4.2 Choosing Solutions to Base the Framework and Prototype on

The SDK contains nine .NET server solutions in C# these are:

1. AlarmCondition Server, a server which simulates alarms in different areas of a process plant. The server allows filtering of alarms , enabling/disabling of alarms and to acknowledge alarms.
2. DataAccess Server, a server which has reading of current values, called DataAccess in OPC terms.
3. Empty Server, a server with an empty Address Space.
4. HistoricalAccess Server, a server which has historical data access, historical process values.
5. HistoricalEvents Server, a server which has historical events.
6. InformationModel Server, a server which builds and exposes an Information Model based on a precompiled binary file and also creates objects dynamically.
7. Methods Server, a server for calling methods in a server. Methods are the equivalent of functions.
8. SimpleEvents Server, a server which shows non historical events.

4.3. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER⁴⁵

9. UserAuthentication Server, a server which shows how to implement user authentication and other issues arising from authentication.

As this thesis is about building a prototype UA server that can connect to and retrieve current and historical data in PIMAQ and model this data according to different models, the most interesting solutions to base the server on was the DataAccess, HistoricalAccess and the InformationModel solutions. All three solutions were investigated more in depth and in the end the framework for the UA server prototype was based on the InformationModel server solution. The main reason for this choice was that there were no real differences between the three solutions for the purpose of creating the framework server. For the prototype server all three solutions were consulted.

4.3 The Implemented Classes for the Prototype Server

4.3.1 UaServer

UaServer is the class which implements the core of the server. It inherits from StandardServer in Opc.Ua.Server and overrides these functions:

- CreateMasterNodemanager()
 - This function creates the MasterNodeManager by populating a list with custom a NodeManager(s). To be able to handle Information Models with custom Nodes and References a developer must create a NodeManager to handle these. The intent is for developers to create a NodeManager for each model they build. When there is a Node request the MasterNodeManager calls each NodeManager in turn to see if they can fulfill the request. It is the job of each NodeManager to check if the Node belongs to it and perform the necessary operations.
- LoadServerProperties()
 - This function sets properties for the server which cannot be configured by the XML files, like manufacturer name, product URI, software version, software certificates and build number.

4.3.2 FrameworkNodeManager

The FrameworkNodeManager class creates and manages the custom boiler model. It overrides and implements some functions from the QuickstartNodeManager class which is part of a support library in the SDK. The QuickstartNodeManager class is covered by the RCBL license, 2.3.1.4 on page 22, and overrides the interfaces INodeManager2(from Opc.Ua.Server), INodeIdFactory(from Opc.Ua.Core) and IDisposable(from System). It is assumed that a production server will implement its own NodeManager(s) as there may be platform specific methods for increasing performance. But for the purpose of the framework it is sufficient as it can handle a custom Information Model.

- Constructor FrameworkNodeManager()
 - Defines the namespaces used in the FrameworkNodeManager, sets the NodeIdFactory and loads the configuration from FrameworkServerConfiguration. The namespace is divided into types and instances of those types.
- Dispose()
 - A function used for “...Use this method to close or release unmanaged resources such as files, streams, and handles held by an instance of the class that implements this interface. By convention, this method is used for all tasks associated with freeing resources held by an object, or preparing an object for reuse.” according to [30].
- New()
 - New overrides New from INodeIdFactory and lets developers create a custom NodeId scheme for keeping track of Nodes that are created. New is part of the NodeId factory. NodeIds are unique identifiers for the objects in the Address Space.
- LoadPredefinedNodes()
 - Loads a predefined Information Model from a binary file. The syntax for loading a file consists of the default namespace for the solution, any project folder names and the file name of the binary file. The resource name will then be DefaultNamespace.PathToFile.FileName.

4.3. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER47

- `CreateAddressSpace()`
 - This function creates the server address space by reading in the pre-defined Nodes and creates “Boiler #1” and then creates “Boiler #23” dynamically.
- `DeleteAddressSpace()`
 - Deletes the address space, freeing created Nodes.
- `GetManagerHandle()`
 - Returns a `NodeHandle` to a `Node` if this `NodeManager` controls the Node. A `NodeHandle` stores information about a `NodeId`.

The framework server’s Address Space is seen as in figure 4.1 when viewed from OPC UA Viewer.

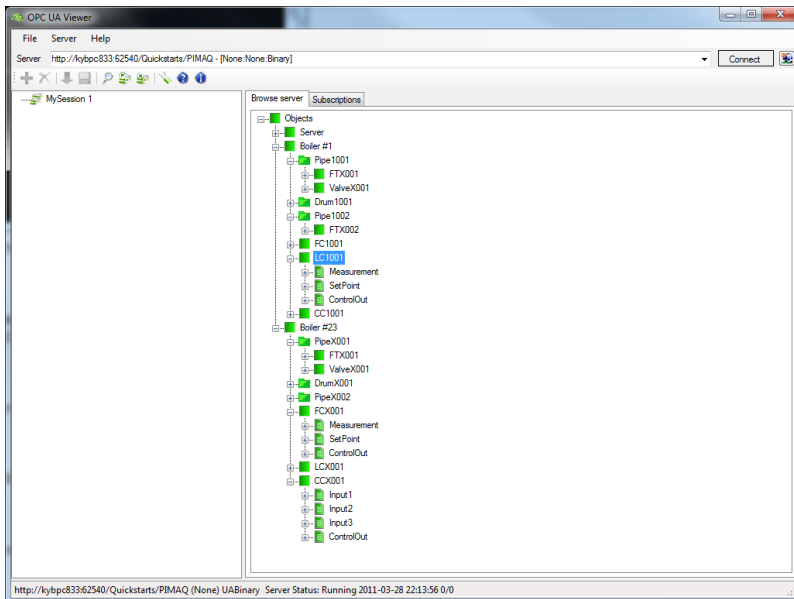


Figure 4.1: The Framework servers Address Space

4.3.3 Boiler.Classes.cs

This is a C# class which is the output of the Model Compiler, the Model Compiler is explained in 3.3.1. Boiler.Classes.cs contains the class definitions for the Nodes and References in the boiler Information Model.

4.4 Implementation Challenges for the Prototype

The implementation of the prototype server offered new challenges with regards to efficiency, caching of data and creating solutions to work around differences between OPC UA and DynamicProcessAPI.

4.4.1 For versus Foreach Loops

Foreach loops are used to iterate data structures like lists and dictionaries safely, i.e. there is no need to worry about indexing. The only requirement is that the size of the data structure does not change during the loop iteration. For loops can handle size changes. As seen in listing 4.1 a Foreach loop should result in more understandable code. Foreach loops may also have a slight performance disadvantage compared to for loops according to [31, 32]. It will not always be the case that the performance is worse but it could be. For this project mostly Foreach loops were used as they suit dictionary and list iteration better and resulted in better code.

Listing 4.1: Foreach vs for loop

```
1 foreach(Book in books)
2     Book.name = ...
3
4 for(int i = 0; i < books.length; i++)
5     books[i].name = ...
```

4.4.2 Loading and Using C++ Dlls in C#, or Using Unmanaged Code in Managed Code

The prototype retrieves data from PIMAQ and need a wrapper to use PIMAQ. C# is part of the .NET platform and is called managed code. It does not have pointers or memory management, but references and garbage collection.

References work more or less like pointers but they do not cause memory leaks if they are not deleted. C++ does have pointers and memory management and is called unmanaged code. As PIMAQ is mostly written in C++ a wrapper must be created to utilize the DynamicProcessAPI in PIMAQ. Using C++ dlls in C# is fairly straightforward:

1. Add System.Runtime.InteropServices to the declarations.
2. Create a class for containing the imported functions.
3. Import the individual functions from the dll by using DllImport before the function name.
4. Managed code has less data types than unmanaged code. For instance managed code has one string type, whereas unmanaged code has many different string types(a introduction can be had from [33]). So if there is a need to send a C# string or another data type into a C++ function then the data type must be marshaled as the correct data type.

Listing 4.2 shows an example of the technique used in DynamicProcessAPI-wrapper.

Listing 4.2: Example of importing a function and marshaling the parameters.

```

1 [DllImport("ProcessDataUtilities.dll", CharSet = CharSet.Unicode)]
2 static extern int pInvokeDisconnect([MarshalAs(UnmanagedType.LPStr)] string
   provider_name, bool last);
3
4 static static int Disconnect(string provider, bool last)
5 {
6     int return_code = pInvokeDisconnect(provider, last);
7     return return_code;
8 }

```

4.4.3 Differences Between DynamicProcessAPI and OPC UA

There are principal differences between OPC UA and DynamicProcessAPI which have been taken into consideration when implementing the server.

4.4.3.1 Data is Saved as a String

DynamicProcessAPI only exposes data as strings. This is a loss of precision as the different data may originally be strings, integer, double, float, guid or other data types. For example a process value could be a float, an engineering unit could be a string and a status word could be an 8 bit binary value. OPC UA offers a large selection of predefined data types, like the standard data types from programming but also image formats, date structures, and the possibility to create your own data type. The server implementation contains functions which show how to do type conversion. This works by assuming that when reading the value for a field in the Tag.Field model the data type is also returned. At the moment this is hard coded in the ReadField function in the DynamicProcessAPIwrapper and has been tested with double and string.

4.4.3.2 Reading Forwards

DynamicProcessAPI reads values forward in time from a start time to a end time, where the start time \leq end time and returns all values in the interval. OPC UA supports more dynamic reading options, these are:

- Forwards with start time, no end time and a maximum number of returned values.
- Backwards with no start time, with end time and a maximum number of values.
- Forwards with start \leq end time with or without a maximum number of values.
- Backwards read with end time \leq start time with or without a maximum number of values.
- For all requests if there is more data than the maximum number of values a continuation point is created so that the read can be continued from there.

To accommodate these reads there is preprocessing of the requests, and in the case of backwards requests post processing of the values.

4.4.3.3 Dead Band Resolution

DynamicProcessAPI uses dead band resolution for saving historical data. When a value changes enough it goes outside the dead band and it is historized. This gives a dynamical resolution which has many values saved for periods of fast change and fewer values for periods of slow change. The OPC UA read function has a parameter which is called maxAge. Maxage gives the maximum age for a value to be valid. As DynamicProcessAPI has dead band the last value is still valid even if it is over maxAge, and this parameter is ignored even though the value may have a timestamp older than maxAge.

4.4.3.4 Bounding Values

Bounding values can be explained as “...When making a request for historical data using the ReadHistory Service, required parameters include a start time and end time. These two parameters define the Time Domain of the ReadHistory request. This Time Domain includes all values between the StartTime and EndTime, and any value that falls exactly on the StartTime, but not any value that falls exactly on the EndTime. For example, assuming bounding values are not requested, if data is requested from 1:00 to 1:05, and then from 1:05 to 1:10, a value that exists at exactly 1:05 would be included in the second request, but not in the first..” from chapter 4.4 in [34]. The problem here is that DynamicProcessAPI returns the value either at start time or the value before start time. So if a client asked for data from 11.00 to 12.00 and the value valid at 11.00(because of dead band) was logged at 10.45 it will return the timestamp of 10.45, not 11.00. This is currently left as is but it may have to be changed when the OPC Foundation releases a compliance test for Historical Access.

4.4.3.5 Fields vs Properties

Values in OPC UA are represented as properties or data variables. Properties “...characterize what the Node represents, such as a device or a purchase order.” and data variables “...represent the content of an Object.” from chapter 4.4 in[35]. For example the engineering unit of a tag describes the tag and is a property. The measured value for a tag would be a data variable. DynamicProcessAPI uses the Tag.Field model where each field is considered a property of the tag. This is a loss of precision and a conversion function has been implemented. Siemens has a wish to have to configure the server as little as possible

so using the `DatavaluesFilter` is optional, and if it is left empty all fields are modeled as properties of the tag.

4.4.3.6 Time

OPC UA assumes all values are saved/stored/sampled in UTC time. `DynamicProcessAPI` uses epoch/Linux time given in local time with/without summer-time (depends on the computer settings). To compensate for this the difference between `DateTime.UtcNow` and `DateTime.Now` is found and subtracted from the start and end times before doing a call into `DynamicProcessAPI`. If there are time problems this should be checked.

4.4.4 Implementation Challenges Which Arise From Behavior in the OPC UA Client SDK

The general advice from the OPC Foundation on developing the server was to make no assumptions about what a client would do. During development and testing there was some issues concerning efficiency and stability which arose from client behavior. The steps taken to correct these issues are described below.

4.4.4.1 Epoch Requests

The client SDK(s) does an open data request from 1970.1.1 00.00.00 (or epoch time) to current time when opening the history read dialog. This is done to fill in the start time field. The commercial client that has been tested also does the same operation. Doing such an open request in `DynamicProcessAPI` is very far from ideal, and can crash the system. This is because there can easily be 3 million values or more for a field. Considering that the return value from `DynamicProcessAPI` contains dates, values, quality and quality status (around 50 characters for each set of data) the total number of characters can be around 150-200 million. According to Siemens such large requests does not happen during normal operation.

As there is no sure way to identify these requests directly, they are filtered out and filled with empty values. This problem is both an issue with client design and the fact that `DynamicProcessAPI` can not restrict the number of values returned in a request. However this is not a perfect solution as large requests

4.5. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER53

can still be done, so a full commercial implementation might have to restrict the time span for requests.

4.4.4.2 Year 1 Request

This is another request which comes in with start and EndTime at 0001.1.1 or midnight year 1. These requests were discovered during software tests. As DynamicProcessAPI uses epoch time there will never be data for these requests so they are filtered out and filled with empty data.

4.5 The Implemented Classes for the Prototype Server

Below is the intended functionality for each implemented class explained, along with important functions in the class. If more in depth knowledge is required then the source code or Documentation.chm can be consulted.

4.5.1 DynamicProcessAPIwrapper

This class imports functions from pInvokeInterface in ProcessDataUtilities.dll. These functions allow communication with DynamicProcessAPI. As can be deduced from the class name it is a wrapper; for using C++ dlls in the C# server.

4.5.1.1 ReadData()

ReadData reads in historical values for a given field under a tag. It works as seen in algorithm 4.1 on the next page. Using a dictionary is not ideal as the data structure does not support direct access of the first and last elements. There are solutions to this but they are as mentioned not ideal. Initially a List of KeyValuePairs was used to save the value and timestamp but later the quality was needed as it is important information. Adding the quality to the value string and then do string operations to retrieve both was considered and tested but it came with performance issues. Adding another list consisting of only quality was also considered but it adds complexity as one has to deal with two lists. In the end a dictionary sorted by DateTime worked as a good compromise.

Algorithm 4.1 The algorithm for reading historical values from DynamicProcessAPI

```

Convert start and end time in UTC to local epoch time
Read in values from DynamicProcessAPI into a result string
while result string contains ";"

    extract date, time and ms
    create a DateTime object using date, time and ms
    extract tag
    extract value
    extract quality
    extract quality status
    Save DateTime, value and quality in a sorted Dictionary

end (while)
return sorted Dictionary

```

4.5.1.2 ReadField()

ReadField reads in the last saved value for a field. This function also returns the data type for the value. Currently it is hard coded but if conversion needs to be tested/expanded this is the place to do changes.

4.5.2 DynamicProcessAPIinterface

This class is intended as the interface to be used for any class that needs to interact with DynamicProcessAPI. If more complex functionality needs to be implemented, by using the wrapper function, then this is the class intended to implement this functionality in. Currently it uses the functions in the wrapper directly except in Disconnect and Initialize.

4.5.2.1 Initialize()

Initialize reads in the providers and the connection string for each provider from DynamicProcessAPIinterfaceConfig.xml. Next it initializes each provider and retrieves a list of all the tags from the providers.

4.5.2.2 Disconnect()

Disconnect iterates through a Dictionary of providers and disconnects each one. When disconnecting from the last provider a bool parameter must be passed as true to avoid memory leaks in DynamicProcessAPI.

4.5.3 DynamicProcessAPINodeManager

When initial development of the prototype started this was the NodeManager that was created and used during most of the development. It uses about 1.5 seconds to create a tag in the Tag.Field model which is too slow for the final prototype but it sufficed for most of the development. The slowness comes from doing rather inefficient string operations which are used when collecting status word bits together. It is now obsolete and not used but it was kept for comparison.

4.5.4 DescendingOrder

When extending the ReadData function in the wrapper to include quality, a sorted dictionary was used. As OPC UA data can also be backwards in time and Dictionaries have no standard method for iterating backwards the class DescendingOrder was created. This class extends IComparable to create a reverse sorted dictionary which is used for backwards requests. This method was had from [36].

4.5.5 TagFieldModelNodemanager

This class mirrors the data structure in DynamicProcessAPI in the server's Address Space. It also handles all client requests which involve the Nodes in the Tag.Field model. It makes no assumptions about the data in DynamicProcessAPI, except that it follows the Tag.Field model and that the status words are called AlarmByte, StatusWord and StatusWord2. Bits for each status word are collected together, as seen in the example below and in figure 4.2 on page 57:

- AlarmByte
 - AlarmByteHH

- AlarmByteLL
- AlarmByteWH
- AlarmByteWL
- AlarmByteYF

Collecting status bits together reduces the amount of fields under each tag and improves the presentation.

4.5.5.1 New()

New is part of the NodeId factory. A NodeId is the unique identifier for a Node. There are 10 different methods for creating NodeIds. The one that is used for this NodeManager is the combination of a uint and the namespace for the NodeManager. The uint is a counter which may be used to keep track the number of Nodes the NodeManager handles. The namespace is the same as the namespace for the Tag.Field model with an added “/Instance”, this is to differentiate between predefined static Nodes and dynamically created Nodes. It is differentiated to avoid giving dynamically created Nodes the same NodeId as static Nodes.

4.5.5.2 LoadPredefinedNodes()

When compiling the Information Model created by either an xml editor or the CAS OPC UA Address Space Model designer a binary file is created, this contains all predefined Nodes. These Nodes can be loaded by the application on start up and used. LoadPredefinedNodes does this. When loading a binary resource the path to the file must consist of DefaultSolutionNamespace.PathToFile.FileName and the file must be included as an embedded resource. This is not documented and the OPC UA message board at [37] had to be consulted.

4.5.5.3 CreateAddressSpace()

This function adds the Tag.Field model to the server’s Address Space. The principle is similar as in DynamicProcessAPINodeManager but because of mapping Function block Types to status word bits before creating the Nodes it is a lot

4.5. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER57

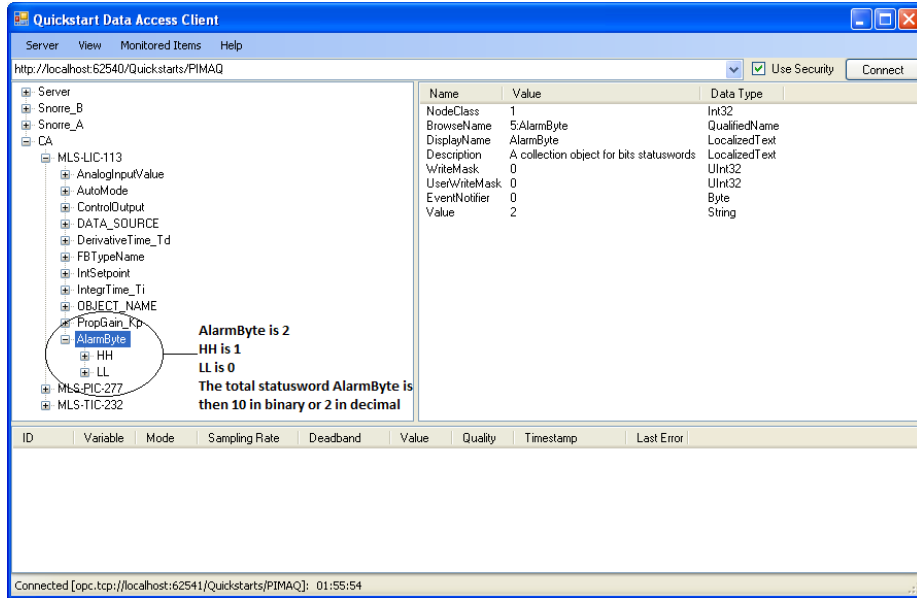


Figure 4.2: Example of collecting status bits together as seen from a client

faster, around 1000 times. Creating 2000 tags takes about 2-2.5 seconds whereas DynamicProcessAPINodeManager used 1.5 seconds per tag. The drawback is the Dictionary doing the mapping which has its definition as:

```
Dictionary<string, Dictionary<string, Dictionary<string, string>>>
```

a rather complex structure. As an example this Dictionary maps an "MA" (FB type) to "AlarmByte" (status word name) to "AlarmByte->HH" (long version) to "HH" (short version). The end result for a client is as seen in figure 4.2.

Clients can subscribe to Nodes in the Address Space for updates. The function UpdateMonitoredItem is called every 500 ms. UpdateMonitoredItem iterates a list of MonitoredItems and publishes updates to the Subscription manager.

4.5.5.4 Read()

This function reads in the current value for a field from DynamicProcessAPI.

Algorithm 4.2 The algorithm for adding the Tag.Field model to the servers Address Space

```

Map Function Block type to status word to long name of bit to short name of bit
Create Function Blocks
Foreach tag in TagsToFbType

    create tag and connect to FB type
    Foreach field in the fields belonging to the tag
        CreateField()
    end (Foreach)

end (Foreach)
Connect Function Block Types to the Root folder for the Address Space
Start MonitoredItem thread

```

4.5.5.5 HistoryRead()

Requests for historical data values for fields are cached to minimize the load on DynamicProcessAPI. The HistoryRead function follows the structure of the SDK where first the type of historical request is identified and then support functions are created to handle each type of request as seen in algorithm 4.4 on page 60. Currently only historical requests for raw sampled data is supported. Other requests return empty values for Nodes belonging to this NodeManager and return a BadHistoryOperationUnsupported status code for the operation. The different requests are for raw historical data in an interval (modified/interpolated or directly sampled), historical values at given times, processed historical data for an interval (like average, standard deviation, median etc) called aggregates and historical events.

Clients can for instance request 10 values between start and end time. If there are more than 10 values in the interval a continuation point is created so that a client can continue the request. Continuation points and data values are saved, retrieved and deleted in the caches by combining a client ID with the tag and field. This means that a client can read historical values for many different fields at a time but can only have one read at a time per field.

Originally continuation points were used directly by returning the next timestamp to the client and the client used that timestamp as either start or end time for the next request. After the cache for data values was implemented continuation points are no longer needed. The framework for using them is still

4.5. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER59

Algorithm 4.3 The algorithm for reading the current value for a field

```
for each Node in NodesToRead
    if Node.Processed
        continue
    if Node.AttributeID == VALUE and the Node belongs to the NodeManager
        Create a DataValue
        Read in a value from DynamicProcessAPI
        Save the value in DataValue
        Save DataValue in a List of returned values
        Set the processed flag for the Node
end (for)
Let the base class Quickstart NodeManager handle other attribute requests
```

there but it is not used. The reason for this is that a fresh request for data values either returns all values or returns x number of values, these x values are removed before caching the data. When the client continues the request the cache is used directly.

4.5.5.6 HistoryReadRawData()

This is the support function for reading raw historical data from DynamicProcessAPI. The sequence of function calls is as shown in figure 4.3. Its main purpose is to first convert the OPC UA request into a request that can be used in DynamicProcessAPI, then it will get data values either from the cache or DynamicProcessAPI and convert these into historical data values for OPC UA.

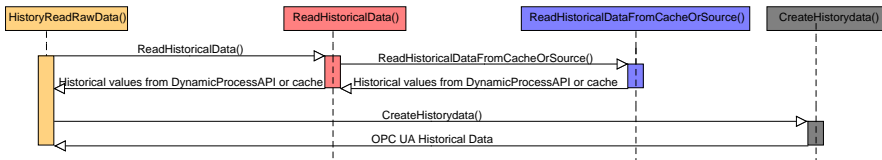


Figure 4.3: Sequence diagram for reading raw historical data values

Algorithm 4.4 The algorithm for HistoryRead

```

if releaseContinuationPoint
    lock
    Foreach HistoryReadValueId Node in nodesToRead
        remove request from data values cache
        remove continuation from continuation point cache
    end (Foreach)
    unlock
    return

if invalid timestamp request
    throw ServiceResultException and return empty data values

if raw historical data request
    if modified raw data
        HistoryReadEmptydata()
    if raw data request
        HistoryReadRawData()

if read historical value at times request
    HistoryReadEmptydata()

if read processed historical values request
    HistoryReadEmptydata()

if historical events request
    HistoryReadEmptydata()

```

4.5.5.7 ReadHistoricalData()

Before using DynamicProcessAPI or the cache to get data values this function handles the continuation point, in the case of a backwards request the end time is switched to the continuation point timestamp and in the case of a forwards request the start time is switched to the continuation point timestamp.

4.5.5.8 ReadHistoricaldataFromCacheOrSource()

This function gets the data values from the cache or DynamicProcessAPI. In the case of a backwards request it post processes the values from DynamicProcess-

4.5. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER61

Algorithm 4.5 The algorithm for HistoryReadRawData

```
if midnight year 1 request
    HistoryReadEmptyData()
    return

if number of values to read per Node == 0
    set number of values to read = Int32.MaxValue

//convert historical request to DynamicProcessAPI request
if no start time is given
    backwards == true
else if only start time is given
    end time = current time
else
    if start time > end time
        backwards = true
        switch start and end time
for each Node in nodesToRead
    if Node.Processed
        continue
    if Node belongs to the NodeManager
        if backwards

            DynamicProcessAPIdata values = ReadHistoricalData(start, end, backwards)
            UA Historydata data = CreateHistoricalData(values, backwards)
            if there are values left
                Create a continuation point
            else
                Remove request from data values cache
            Node.Processed = true
        else

            DynamicProcessAPIdata values = ReadHistoricalData(start, end, backwards)
            UA Historydata data = CreateHistoricalData(values, backwards)
            if there are values left

                (for instance average)Create a continuation point
            else
                Remove request from data values cache
            Node.Processed = true
    else
        continue
end (for)
```

Algorithm 4.6 The algorithm for ReadHistoricalData

```
if backwards
    if continuation point
        if the continuation point is cached
            ReadHistoricalDataFromCacheorSource(start, continuation point time, backwards)
        else
            ReadHistoricalDataFromCacheorSource(start, end, backwards)
    else
        ReadHistoricalDataFromCacheorSource(start, end, backwards)
else
    if continuation point
        if the continuation point is cached
            ReadHistoricalDataFromCacheorSource(continuation point time, end, backwards)
        else
            ReadHistoricalDataFromCacheorSource(start, end, backwards)
    else
        ReadHistoricalDataFromCacheorSource(start, end, backwards)
```

4.5. THE IMPLEMENTED CLASSES FOR THE PROTOTYPE SERVER63

Algorithm 4.7 The algorithm for ReadHistoricalDataFromCacheOrSource

```
if the request is cached
    return cached values
else
    Dictionary values = read data from DynamicProcessAPI
    if no values are returned
        create an empty value with bad quality
        add the empty value to values
    if backwards
        Foreach date, value and quality in values
            add date, value and quality to reverse sorted Dictionary
        end (Foreach)
        add reverse sorted Dictionary to cache
        return reverse sorted Dictionary
    else
        add values to cache
        return values
```

API and creates a reverse order sorted Dictionary. Because of this, backwards requests use more time and are slightly less efficient.

4.5.5.9 CreateHistoryData()

After data has been retrieved from the cache or DynamicProcessAPI it must be converted into the format/data structure OPC UA uses for historical values. CreateHistoryData converts DynamicProcessAPI values into OPC UA DataValues.

4.5.5.10 ReadDatavaluesFilter()

This function reads in the data values that is defined in DataValuesFilter.xml and saves them in a dictionary.

Algorithm 4.8 The algorithm for CreateHistoryData

```
OPC UA Historydata data
if the number of values > number of values returned per Node
    stop = number of values returned per Node
else
    stop = number of values
//convert to OPC UA Historydata
while number of values > 0 and number of values in data < stop
    current = get timestamp from Dictionary
    value = get value from Dictionary
    quality = get quality from Dictionary
    //handle not returning bounds
    if backwards
        if not return bounds and current == start time
            continue
    else
        if not return bounds and current == end time
            continue
    Create a DataValue using value and current
    Convert the DataValue to correct data type
    Convert quality to quality for the DataValue
    Save the DataValue in data
    Remove current from values
end (while)
return data
```

Algorithm 4.9 The algorithm for CreateAddressSpace for the ISO model NodeManager

```

Load in the platform model
Convert the untyped object to a typed Platform object
Read in the filter which maps Displaynames to tags
Add the platform object to a stack
//Do a depth first search on the model
while stack size != 0

    Node = stack.pop()
    Add Node.children to the stack
    if Node.Displayname exists in the filter
        Create a new field "ProcessValue"
        Convert the field to the correct data type
        Connect the field to Node

end (while)
Connect the platform object to the root folder for the Address Space
Start MonitoredItem thread

```

4.5.6 ISOModelNodeManager

The NodeManager for the ISO model shares most of the functions in the TagFieldNodeManager, these are explained earlier. The ISOModelNodeManager does not add the ISO model to the server's Address Space dynamically but it loads the model from a binary file, ISO 15926.PredefinedNodes.uanodes, and then connects tags from DynamicProcessAPI to Nodes in the model.

4.5.6.1 CreateAddressSpace()

First the Snorre B platform ISO model is loaded from the binary file and then converted into a PlatformState (type definition). To connect DynamicProcessAPI tags to Nodes in the platform model a filter was created, this filter is further explained in section 4.6.5 on page 71. The filter maps Displaynames in the ISO model to tag names in DynamicProcessAPI. Only the ProcessValue and ControlOutput fields are connected at the moment, this is hard coded in. In the future which fields to connect can be decided and hard coded or the filter can be extended. As it is now, it shows how to connect the model.

4.5.6.2 ReadModelFilter()

This function reads in values from ISOModelFilter.xml and saves these in a Dictionary. The Dictionary maps Displaynames for Nodes in the ISO 15926 model to tag names in DynamicProcessAPI. Currently only one tag can be mapped to each Node. It can be extended to handle multiple tags per Node by mapping Displaynames to lists of tags.

4.5.7 Conversion functions

The NodeManagers may use four functions to implement Node, data type and quality conversion.

4.5.7.1 CreateField()

Creates field Nodes under a tag or a status word collection. The Nodes themselves are created using CreateFieldOfType().

4.5.7.2 CreateFieldOfType()

Creates field Nodes of the correct type (currently string/double data variable or string/double property) using the Dictionary created from DataValuesFilter.xml.

4.5.7.3 DataTypeConversion()

This function implements data type conversion between a DynamicProcessAPI value to an OPC UA DataValue.

4.5.7.4 QualityConversion()

This function implements quality conversion between a DynamicProcessAPI quality to quality for an OPC UA DataValue.

4.6 Filters and Configuration Files

Starting the server begins in the Program.cs file where an ApplicationInstance is created. The ApplicationInstance class creates, installs and runs an UA application by using the configuration files. The sequence of calls is shown in figure 4.4.

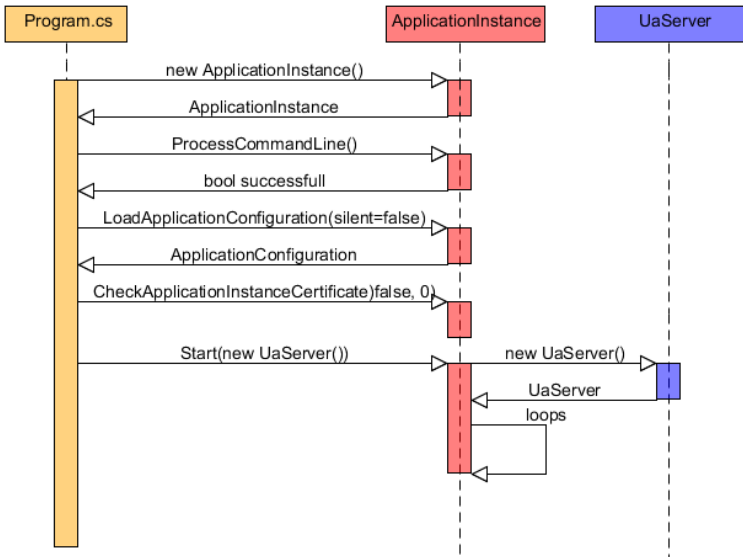


Figure 4.4: Sequence diagram for creating and starting an OPC UA server

The prototype server is configured with six XML files. For this server the settings was kept more or less unchanged from the SDK with the exception of adding logs, changing the server name, certificate location, and signing/encryption was removed from the communication options. It was found that the ability to view logs helped with debugging.

4.6.1 App.Config

App.Config.xml defines the behaviors of the services used for communication, and logging and diagnostics options. It also links to the configuration file for the specific server as seen in listing 4.3 on the facing page.

4.6.2 UaFramework.Config.xml

On the last lines App.Config.xml links to UaFramework.Config.xml. UaFramework.Config.xml configures the server with a name, a certificate location, trusted certificate authorities, communication options (like ports and the security for the communication), timeouts, sampling intervals and log locations. An abridged version of UaFramework.Config.xml is shown in listing 4.4 on page 70.

The server prototype is configured with TCP/IP and SOAP/HTTP communication with binary and XML data encoding. As the server is a prototype, and the focus of the thesis is information modeling, security is not used and anonymous clients are accepted. The only requirement for communication is a valid certificate. The prototype uses the “Quickstart Information Model Server” certificate installed by the SDK.

4.6.3 InstallConfig.xml

This configuration file allows one among others things to set the name of the server, define a trusted certificate store and sets the “install the server as a windows service” option. An example is seen in listing 4.5 on page 71.

4.6.4 DynamicProcessAPIInterfaceConfig.xml

When connecting to/starting DynamicProcessAPI the data providers must be loaded. They are loaded by naming the correct dll and using a connection string. Choosing the providers is done by first defining the provider dll and then defining the connection string for the provider. As seen in listing 4.6 on page 71 the connection string for the Staticprovider consist of either relative or absolute paths to cvs files separated by a “;”. For a remote IP21 database the connection string consists of an IP address and communication options and then which NORSOK function blocks types to load separated by a “;”.

Listing 4.3: App.Config.xml example

```

1 <?xml version="1.0"?>
2 <configuration>
3   <configSections>
4     <section name="UaFramework" type="Opc.Ua.ApplicationConfigurationSection,Opc.Ua.Core"/>
5   </configSections>
6   <system.serviceModel>
7     <!--
8       UI servers built with the SDK use the Opc.Ua.SessionEndpoint class to implement the ISessionEndpoint
9       contract. It is possible to add additional behaviors in the configuration file by referencing this
10      service. For example, the configuration in this file turns on meta data publishing.
11    -->
12    <services>
13      <service name="Opc.Ua.SessionEndpoint" behaviorConfiguration="Opc.Ua.SessionEndpoint.Behavior">
14        <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange"/>
15      </service>
16    </services>
17    <!-- Servers deployed in production environments should turn the
18    httpGetEnabled and includeExceptionDetailInFaults options off -->
19    <behaviors>
20      <serviceBehaviors>
21        <behavior name="Opc.Ua.SessionEndpoint.Behavior">
22          <serviceMetadata httpGetEnabled="true"/>
23          <serviceDebug includeExceptionDetailInFaults="true"/>
24        </behavior>
25      </serviceBehaviors>
26    </behaviors>
27    <!--
28      Uncommenting this <diagnostics> block will turn on message logging. The contents and the location of
29      the
30      log file are specified in the <system.diagnostics> block.
31    -->
32    <!--
33    <diagnostics>
34      <messageLogging logEntireMessage="true" maxMessagesToLog="3000"
35      logMessagesAtServiceLevel="true" logMalformedMessages="true" logMessagesAtTransportLevel="true"/>
36    </diagnostics>
37    -->
38    </system.serviceModel>
39    <!--
40    <system.diagnostics>
41      <sources>
42        <source name="System.ServiceModel" switchValue="Verbose,ActivityTracing">
43          <listeners>
44            <add type="System.Diagnostics.DefaultTraceListener" name="Default"/>
45            <add name="ServiceModelListener"/>
46          </listeners>
47        </source>
48        <source name="System.ServiceModel.MessageLogging">
49          <listeners>
50            <add type="System.Diagnostics.DefaultTraceListener" name="Default"/>
51            <add name="ServiceModelListener"/>
52          </listeners>
53        </source>
54      </sources>
55      <sharedListeners>
56        <add initializeData="Quickstarts.BoilerServer.svclog" type="System.Diagnostics.
57        XmlWriterTraceListener,
58        System,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b77a5c561934e089"
59        name="ServiceModelListener"
60        traceOutputOptions="LogicalOperationStack,DateTime,TimeStamp,ProcessId,ThreadId,Callstack"/>
61      </sharedListeners>
62    </system.diagnostics>
63    -->
64    <UaFramework>
65      <ConfigurationLocation xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd">
66        <FilePath>UaFramework.Config.xml</FilePath>
67      </ConfigurationLocation>
68    </UaFramework>
69  </start-up></configuration>

```

Listing 4.4: An abridged UaFramework.Config.xml example

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <ApplicationConfiguration
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xmlns:ua="http://opcfoundation.org/UA/2008/02/Types.xsd"
5   xmlns:s1="http://opcfoundation.org/UA/Sample/Configuration.xsd"
6   xmlns:s2="http://opcfoundation.org/UA/SDK/COM/Configuration.xsd"
7   xmlns="http://opcfoundation.org/UA/SDK/Configuration.xsd">
8   <ApplicationName>PINAQ UA Server</ApplicationName>
9   <ApplicationUri>urn:localhost:PINAQ</ApplicationUri>
10  <ProductUri>http://NTNU.no/PINAQ</ProductUri>
11  <ApplicationType>Server_0</ApplicationType>
12  <SecurityConfiguration>
13    <ApplicationCertificate>
14      <StoreType>Directory</StoreType>
15      <StorePath>%CommonApplicationData%\OPC Foundation\CertificateStores\MachineDefault</StorePath>
16      <SubjectName>Quickstart Information Model Server</SubjectName>
17    </ApplicationCertificate>
18    <TrustedPeerCertificates>
19      <StoreType>Windows</StoreType>
20      <StorePath>LocalMachine\UA Applications</StorePath>
21    </TrustedPeerCertificates>
22  </SecurityConfiguration>
23  <TransportConfigurations></TransportConfigurations>
24  <TransportQuotas>
25    <OperationTimeout>600000</OperationTimeout>
26    <MaxStringLength>1048576</MaxStringLength>
27    <MaxByteStringLengthImplementationOfThePrototypeth>1048576</MaxByteStringLength>
28    <MaxArrayLength>65535</MaxArrayLength>
29    <MaxMessageSize>4194304</MaxMessageSize>
30    <MaxBufferSize>65535</MaxBufferSize>
31    <ChannelLifetime>300000</ChannelLifetime>
32    <SecurityTokenLifetime>3600000</SecurityTokenLifetime>
33  </TransportQuotas>
34  <ServerConfiguration>
35    <BaseAddresses>
36      <ua:String>http://localhost:62540/Quickstarts/PINAQ</ua:String>
37      <ua:String>opc.tcp://localhost:62541/Quickstarts/PINAQ</ua:String>
38    </BaseAddresses>
39    <SecurityPolicies>
40      <ServerSecurityPolicy>
41        <SecurityMode>None_1</SecurityMode>
42        <SecurityPolicyUri>http://opcfoundation.org/UA/SecurityPolicy#None</SecurityPolicyUri>
43        <SecurityLevel>0</SecurityLevel>
44      </ServerSecurityPolicy>
45    </SecurityPolicies>
46    <UserTokenPolicies>
47      <ua:UserTokenPolicy>
48        <ua:TokenType>Anonymous_0</ua:TokenType>
49      </ua:UserTokenPolicy>
50    </UserTokenPolicies>
51    <DiagnosticsEnabled>true</DiagnosticsEnabled>
52    <AvailableSamplingRates>
53      <SamplingRateGroup>
54        <Start>5</Start>
55        <Increment>5</Increment>
56        <Count>20</Count>
57      </SamplingRateGroup>
58    </AvailableSamplingRates>
59    <MaxRegistrationInterval>30000</MaxRegistrationInterval>
60    <NodeManagerSaveFile>PINAQ.nodes.xml</NodeManagerSaveFile>
61  </ServerConfiguration>
62  <TraceConfiguration>
63    <!--<OutputFilePath>%LocalApplicationData%\OPC Foundation\Logs\UaFramework.log.txt</OutputFilePath-->
64    <OutputFilePath>Logs\PINAQ.log.txt</OutputFilePath>
65  </TraceConfiguration>
66 </ApplicationConfiguration>

```

Listing 4.5: InstallConfig.xml for the prototype

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <s0:InstalledApplication xmlns:s0="http://opcfoundation.org/UA/SDK/Installation.xsd" xmlns="http://
  opcfoundation.org/UA/SDK/Configuration.xsd" xmlns:ua="http://opcfoundation.org/UA/2008/02/Types.xsd"
3   >
4   <Name>Siemens PIMAQ UA Server</Name>
5   <ConfigurationFile>UaFramework.Config.xml</ConfigurationFile>
6   <TrustedPeerStore>
7     <StoreType>Windows</StoreType>
8     <StorePath>LocalMachine\UA Applications</StorePath>
9   </TrustedPeerStore>
10  <s0:DeleteCertificatesOnUninstall>true</s0:DeleteCertificatesOnUninstall>
11  <s0:ConfigureFirewall>false</s0:ConfigureFirewall>
12  <s0:SetConfigurationFilePermissions>false</s0:SetConfigurationFilePermissions> <
13  <s0:SetExecutableFilePermissions>false</s0:SetExecutableFilePermissions>
14  <s0:InstallAsService>false</s0:InstallAsService>
15  <s0:TraceConfiguration>
16    <OutputFilePath>Logs\PIMAQ.InstallLog.xml</OutputFilePath>
17    <DeleteOnLoad>true</DeleteOnLoad>
18    <TraceMasks>1023</TraceMasks>
19  </s0:TraceConfiguration>
20 </s0:InstalledApplication>

```

Listing 4.6: Example of DynamicProcessAPIinterfaceConfig.xml

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <dynamicProcessAPI>
3   <provider>SiemensIP21ProviderRemote</provider>
4   <connectionString>10.234.239.178 200 /FATAL;Custom Records;MAObjectDef,CSObjectDef</connectionString>
5   <provider>StaticDataProvider</provider>
6   <connectionString>MLS-LIC-113.csv;20BL104007.csv</connectionString>
7 </dynamicProcessAPI>

```

4.6.5 ISOModelFilter.xml

This filter connects tags in DynamicProcessAPI to Nodes in the ISO 15926 model. This is done by writing the Displayname for the Node from the model and adding a connection with the name of the DynamicProcessAPI tag. Currently only one tag with the field ProcessValue can be connected to a ISO model Node. It should however be easy to expand this by changing the ReadModelFilter function in ISOModelNodeManager. An example is seen in listing 4.7 on the following page.

4.6.6 DatavaluesFilter.xml

The data values filter is used to identify fields in the tag.field model which are considered as data values for the tag, other fields are then by default properties of the tag. If it is left empty all fields are properties of the tag in the server's Address Space.

Listing 4.7: Example of the ISO model filter

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <SnorreB>
3   <C6_Subsea_Sand_Detector>
4     <Connection>11B-LST_0367</Connection>
5   </C6_Subsea_Sand_Detector>
6   <C6_Subsea_Pressure_Element_1>
7     <Connection>11C-LST_4018</Connection>
8   </C6_Subsea_Pressure_Element_1>
9   <C6_Subsea_Temperature_Element_1>
10    <Connection>11D-LST_4013A</Connection>
11  </C6_Subsea_Temperature_Element_1>
12  <C5_Producing_Well>
13    <Connection>MLS-LIC-113</Connection>
14  </C5_Producing_Well>
15 </SnorreB>
```

Listing 4.8: Example of the DataValues filter

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <DataValues>
3   <DataValue>ProcessValue</DataValue>
4   <DataValue>ControlOutput</DataValue>
5   <DataValue>AnalogInputValue</DataValue>
6 </DataValues>
```

Chapter 5

Testing

The OPC Foundation provides the Compliance Test Tools (CTT) which can test an OPC UA application for its compliance to the specifications. The functionality of the OPC UA application is described by its Profile(s). A Profile is built up using ConformanceUnits and Facets. ConformanceUnits are unique features of an OPC UA application, like the ability to read attributes for Nodes or an application that supports TCP communication. A facet is a collection of ConformanceUnits that is intended to be part of a complete Profile, as the facet does not have enough features by itself. When describing the compliance of an OPC UA application one will say that the application conforms to for instance Profile1, Profile2 and Profile 3[6, 38].

To setup the CTT:

1. First the URL and port to the server must be defined along with the server's certificate.
2. After this it is possible to connect to the server and add the References and Nodes in the Address Space to the CTT. As seen in figure 5.1 on the next page the ReferenceType hasDownhole is used as the ReferenceType for the CTT. The CTT will then use this Reference for any tests involving ReferenceTypes.

Once the CTT is configured the OPC UA application can be tested. Tests can be done on Profiles, ConformanceUnits and individual tests within each Confor-

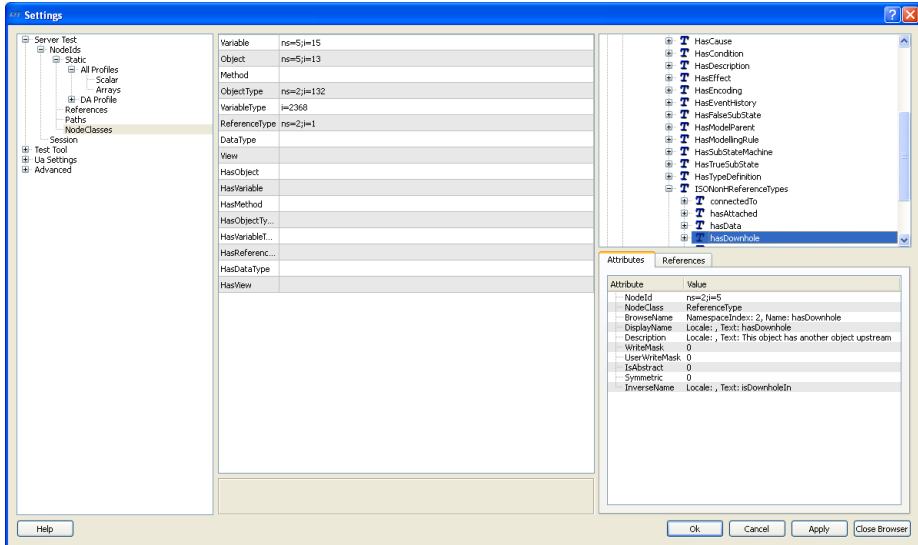


Figure 5.1: Connecting Nodes in the server to the CTT

manceUnit. Figure 5.3 on page 77 shows an example of a ConformanceUnit test while figure 5.4 on page 78 shows an example of a Profile test. The tests can be debugged within the CTT and one can define new tests or change existing tests. Figure 5.2 on the facing page from [39] shows to how to interpret the results of testing.

The CTT cannot test all ConformanceUnits yet. For this thesis this means that the TCP and SOAP communication protocols cannot be tested along with reading of historical values. Testing of historical read/write is expected in Q3 2011.

5.1 Read and Write Compliance Test

While writing of values has not been implemented, the Write() function itself shall let clients know that the operation is unsupported. If an operation is unsupported it should show up as a question mark in the CTT test. The Read function is implemented and was tested.






Icon	Meaning	Weighting	Description
	Not Implemented	Highest = 5	Indicates that a Service or some functionality that was going to be tested was reported by the Server as not supported/implemented.
	Error	4	An error occurred in a script. This may include a non-compliance, or a scripting error.
	Warning	3	A warning was detected. This might mean that a minor non-compliance was detected or that there was a configuration problem.
	Skipped	2	The test was skipped. This might be because of a configuration problem or because the test-environment is not applicable for the test.
	Pass	1	The test completed successfully and is Compliant!

Figure 5.2: Guide to interpreting CTT results

The first compliance test revealed that the returned service result for the Write function was wrong and the service result was changed to BadWriteNotSupported.

The final result of testing the ConformanceUnits for read and write is as seen in figure 5.3 on page 77. By reading the results it is seen that the Read ConformanceUnit passes the compliance tests with warnings for minor non-compliance. For Write there is one error in test 5.8.2-004.js else it passes, because of writing to invalid Nodes, which is detected, or it is returned that Write is not supported. When examining test 5.8.2-00.js closer it was found that the test assumes that Write is supported. This means that even though the server returns not supported the test does not take this into consideration and it fails with the result that the Write ConformanceUnit test failed rather than being unsupported.

5.2 Server Compliance Test

A complete server consists of many different Profiles, these have been collected into server Profiles which should fit into different hardware. The server Profiles in the CTT are:

- Standard UA Server, standard UA server running on a PC.
- Embedded UA Server, for devices with more than 50mb of memory and a CPU equivalent of a Intel 486 processor.

- Low End Embedded Device Server, for small devices with limited resources. This server assumes that security is not used.

As part of this thesis is about building a server that is to be used in an industrial context, the server was tested against the Standard UA Server Profile. The results are in figure 5.4 on page 78. As can be seen the prototype is not ready to be deployed as an OPC UA server. Most of these ConformanceUnits and Facets are handled by the Quickstart NodeManager in the SDK. As the Quickstart NodeManager is under the RCL license and can not be modified without sharing, a custom NodeManager should be built from scratch.

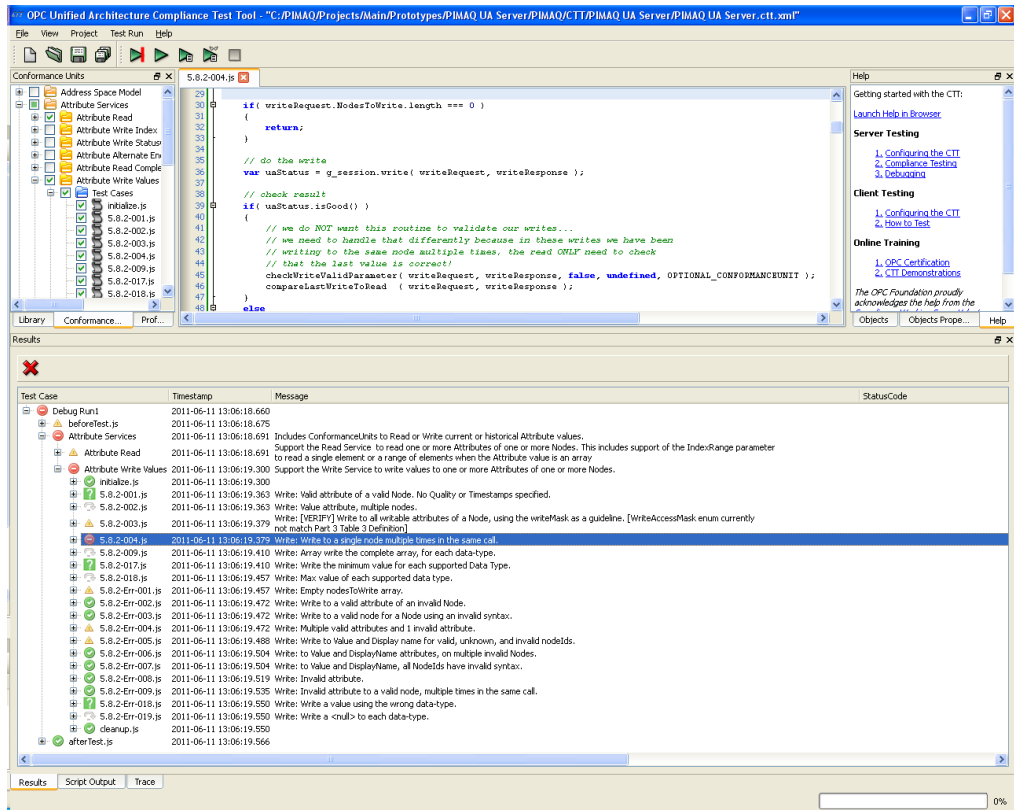


Figure 5.3: The results of the Read and Write attributes compliance tests

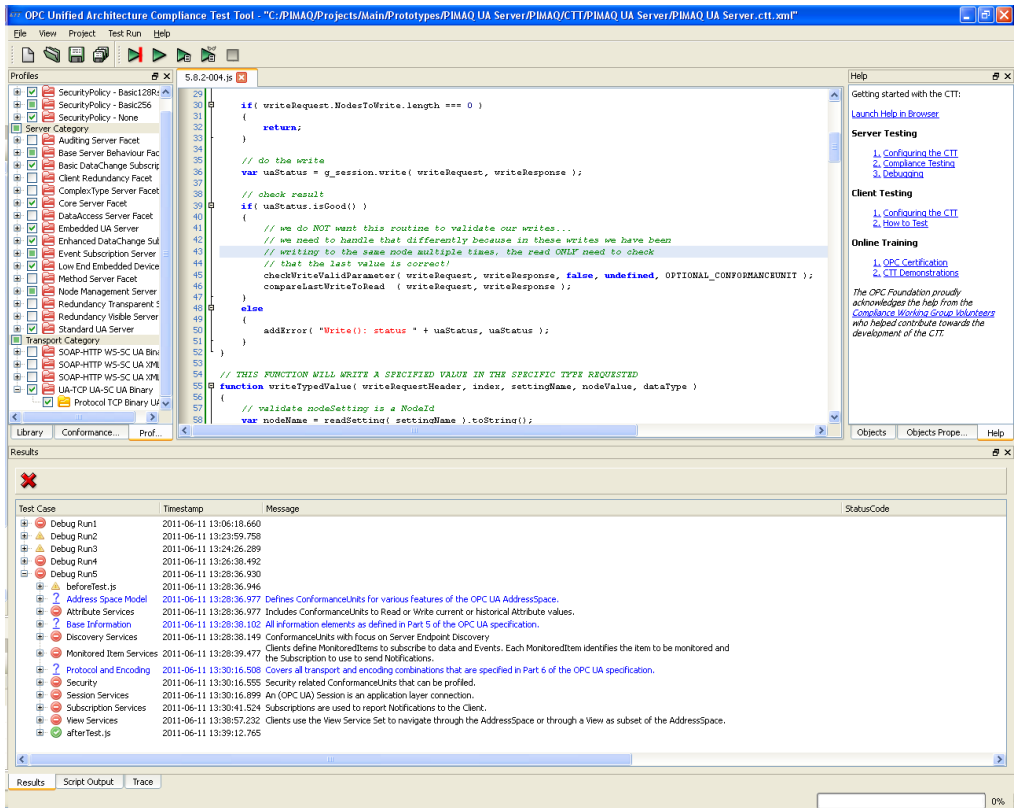


Figure 5.4: The results for the Standard UA Server Profile test

Chapter 6

Discussion

Chapter 6 discusses some of the lessons learned during implementation and design, and the weaknesses of the prototype server.

6.1 Unifying OPC UA and DynamicProcessAPI

One of the properties for the server Siemens specified was to have as little configuration as possible before starting the server. This property has resulted in problems when trying to balance between the property and trying to unify OPC UA and DynamicProcessAPI. There are fundamental differences between what OPC UA specifies/assumes and how DynamicProcessAPI works. The differences that have been balanced are:

- Data type conversion of fields.
- Fields vs properties and values.

Field values are saved as string but they may be for example floats, bits or integers when they are used in the process. Conversion could have been hard coded into the source code, not very flexible, or a filter could have been created which maps field names to data type, resulting in more configuration. The solution was to assume a future extension of DynamicProcessAPI would also include information about data type.

For the Tag.Field model all fields are considered as properties of the tag. OPC UA differentiates between properties, which describe a value, and the data value(s) itself. As with data type conversion two solutions presents themselves; hard coding the conversion into the source or creating a filter which maps fields as properties. This filter should be less work than data type conversion as there are less data values than properties and anything not a data value is automatically a property. Currently a filter has been created which may be left empty, in which case all fields are modeled as properties of the tags.

Other differences has resulted in special solutions when implementing, these are

- Read requests.
- Time.

Historical read requests have to be converted from OPC UA format into a request DynamicProcessAPI can understand. This adds to the response time between clients and the server as it involves both preprocessing of the requests and post processing of the returned values. Unfortunately it also can crash the server if the conversion (or the original request) results in a too large time span and too many returned values. Currently request asking for data from 1970.1.1 to now are filtered out. In the future it should ideally be possible to restrict the number of values in DynamicProcessAPI. An solution would be to restrict the time frame of all requests into something which is considered safe. This restriction should be a part of any commercial deployment unless DynamicProcessAPI is changed.

OPC UA assumes UTC time whereas DynamicProcessAPI uses epoch time in local time. This has resulted in conversion between the different formats and currently it has been tested and it works. However there can be software/hardware conditions later which may not have been accounted for.

The small differences between OPC UA and DynamicProcessAPI have in some cases resulted in special solutions where requests are ignored or data is post processed. This seems to be inevitable when trying to unify two data models.

By trying to respect Siemens's property the server implementation has been affected and the property has placed restrictions on the attempt to unify OPC UA and DynamicProcessAPI. Solutions have been suggested but they do require changes in the existing code base, a code base which is proven to work. Or the solutions can be hard coded which makes the server non flexible. Changes

in existing code base may also affect old applications and may render them incompatible. Siemens may have to modify or remove the property if they decide to create a commercial solution. To put it simply, without configuration or with little configuration `DynamicProcessAPI` must be changed to conform more to OPC UA or the server must be non flexible.

Another problem is that ISO 15926 models are expected to be rich in information[19], so that they may be used to automate tasks as much as possible. Lack of information about data types and properties reduces the richness of the model and may make it unsuitable.

6.2 Inheritance

As both the `ISOModelNodeManager` and the `TagFieldNodeManager` consist of mostly the exact same functions, then ideally a base class should have been implemented for the read, the history read and the support functions which could be inherited by the `NodeManagers`. At the moment most changes done in one `NodeManager` must be mirrored in the other. This is an easy source of errors and it is not good objective design. The main reason this happened was because the SDK overrides the Quickstart `NodeManager` to implement its own `NodeManagers` and this recipe was followed. As it is now it is not an ideal design.

6.3 Modularity

As mentioned in the chapter 3 on page 25 there were not many design choices to make for the server. Siemens specified that the server should be modular so that it is easy to extend and easy to modify. Though not a lot design choices were made, the prototype is modular and consists of classes with clear and separate functionality.

6.4 HistoryRead and Cache

Reading historical values has not been solved in the most efficient way as the values could have avoided post processing by moving the creation of the reverse

sorted Dictionary into DynamicProcessAPIwrapper. The DynamicprocessAPI to OPC UA Historydata conversion may also be moved into the wrapper.

The cache works but if a client crashes the values will not be deleted from the cache. This can cause memory issues in the long run.

Chapter 7

Conclusion

When trying to unify OPC UA and DynamicProcessAPI there was a need for configuration and special solutions. The special solutions are an inevitable result of small differences between the two. But the configuration that was needed is a direct result of the larger differences and assumptions made in the design of OPC UA and DynamicProcessAPI.

When Siemens requested to have as little configuration as possible this was a hard request to respect as the only other methods is to make DynamicProcessAPI conform more to OPC UA or hard code the configuration. Hard coding makes the server non flexible and changing DynamicProcessAPI may not be ideal for the reasons mentioned in 6.1 on page 80. The end result is a compromise with a server requiring configuration and which also requires future changes in DynamicProcessAPI. Without configuration there is a loss of meta data which is not ideal as ISO 15926 models requires as much data as possible to facilitate automation.

The server has been tested with different clients using TCP and SOAP/HTTP and the Read and HistoryRead functions works. The server can reflect data in DynamicProcessAPI in its Address Space. The compliance testing of the server revealed that it is not ready for industrial deployment, however the read attribute service for string has only minor non-compliance.

The ISO model of Snorre B has been converted into an Information Model for the OPC UA server using the method described on page 36. The server presents

this model for clients and connects current and historical data from PIMAQ to the model. While the entire platform Snorre B has not been modeled because of time constraints, enough has been modeled to show that OPC UA is capable of representing ISO 15926 models. This also leads to the conclusion that OPC UA is a technology that can be used when a company implements Integrated Operations.

7.1 Future Work

A commercial UA server should implement NodeManager(s) from scratch, both for efficiency and because the Quickstart NodeManager is under a license which means it cannot be modified without sharing the change or payment. For compliance testing it will be a must to be able to change the entire NodeManager.

Implement writing of current and historical values, the framework is there but it is not connected to DynamicProcessAPI.

There was a wish to change the Information Model without recompiling, just restarting the server, this is not possible directly. However the Address Space model service set can modify (add/delete Nodes) and save the Address Space while the server is running. The functionality for this is very similar to the way the Address Space is created in the Tag.Field NodeManager. This service set can be implemented in the future.

Add data type information to the ReadField function in DynamicProcessAPI for conversion in the server.

Restrict the number of returned values in DynamicProcessAPI or filter out requests in the server by not allowing requests to span more than a safe time period.

Allow DynamicProcessAPI to read backwards and forwards.

Make the reading of historical values more efficient as discussed in chapter 6.4.

Bibliography

- [1] Statistics Norway. (2011) Decreased petroleum production in 2010. Statistics Norway. [Online]. Available: http://www.ssb.no/english/subjects/10/06/20/ogprodre_en/
- [2] Det Kongelige Olje og Energidepartement, *Stortingsmelding 38 - om petroleumsvirksomheten*, 2003-2004.
- [3] The Norwegian Oil Industry Association(OLF), "Dataintegrasjon," 2011, in Norwegian. [Online]. Available: <http://olf.no/Var-virksomhet/HMS-og-Drift/Integrerte-operasjoner/Dataintegrasjon/>
- [4] POSC Caesar, "Introduction to ISO15926," POSC Caesar, 2011. [Online]. Available: <https://www.posccaesar.org/wiki/ISO15926>
- [5] Wikipedia, "ISO15926," 2011. [Online]. Available: http://en.wikipedia.org/wiki/ISO_15926
- [6] K. Mo, "Investigating OPC Unified Architecture for future implementation in Industrial Information Management Systems," December 2010.
- [7] Siemens, "Automasjons produkt." [Online]. Available: <http://www.nwe.siemens.com/norway/internet/no/produkter/oil/Pages/automasjon.aspx>
- [8] Wikipedia, "The Semantic Web." [Online]. Available: http://en.wikipedia.org/wiki/Semantic_Web
- [9] Rafael Batres, Matthew West, David Leal, David Price, Katsube Masaki, Yukiyasu Shimada, Tetsuo Fuchino and Yuji Naka, "An upper ontology based on ISO 15926," *Computers and Chemical Engineering*, vol. 31, pp. 519–534, 2007.

- [10] Wikipedia, "Integrated Operations," Wikipedia Article, 2011. [Online]. Available: http://en.wikipedia.org/wiki/Integrated_operations
- [11] The Norwegian Oil Industry Association(OLF), "Integrated Operations and the Oil & Gas Ontology," PDF from olf.no, 2009.
- [12] S. Spørkel, "Integrerte Operasjoner i et informasjonsperspektiv," Master's thesis, University of Stavanger, June 2010, chapter 1.1 page 3-6.
- [13] NTNU, "Better resource utilization." [Online]. Available: http://www.ntnu.edu/research/research_excellence/io
- [14] S. L. T. Jon Atle Gulla and D. Strasunskas, "Semantic Interoperability in the Norwegian Petroleum Industry," *Norwegian University of Science and Technology*.
- [15] T. O. Grøtan and E. Albrechtsen, "Risikoplanlegging og analyse av Integrete Operasjoner (IO) med fokus på å synliggjøre kritiske MTO aspekter," SINTEF, Tech. Rep., June 2008.
- [16] T. O. G. Camilla Knudsen, Linda-Sofie Lunde-Hanssen and M. Pedersen, "Hva innebærer egentlig Integrerte Operasjoner?" Report from Sintef, Tech. Rep., June 2006.
- [17] POSC Caesar, "ISO15926 Primers," POSC Caesar, 2011, all of the webpages linked to on the top right-hand side. [Online]. Available: <https://www.posccaesar.org/wiki/ISO15926>
- [18] D. Price and R. Bodington, "Applying Semantic Web Technology to the Life Cycle Support of Complex Engineering Assets," *ISWC LNCS*, pp. 812–822, 2004.
- [19] M. G. S. Johan W. Klüwer and M. Valen-Sendstad, "ISO 15926 templates and the Semantic Web," *Position paper for W3C Workshop on Semantic Web in Energy Industries; Part I: Oil & Gas*, 2008.
- [20] O. Paap. (2008, December) ISO 15926 for interoperability. Presentation for W3C workshop.
- [21] I. G. Det Norske Veritas, "ISO15926 a Technical Introduction. How does it work, and what is involved in using it ?" Presentation, 2 2010.

- [22] B. Smith, "Against Idiosyncrasy in Ontology Development," in *Proceeding of the 2006 conference on Formal Ontology in Information Systems: Proceedings of the Fourth International Conference (FOIS 2006)*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2006, pp. 15–26. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1566079.1566085>
- [23] OPC Foundation, "OPC UA licenses," 2011. [Online]. Available: <http://www.opcfoundation.org/Default.aspx/License/UALicense.asp?MID=AboutOPC>
- [24] CAS, "CAS OPC UA Viewer." [Online]. Available: <http://www.commsvr.com/Products/OPCUAViewer/tabid/520/language/en-US/Default.aspx>
- [25] OPC Foundation, "OPC UA SDK Architecture," 2011. [Online]. Available: <http://www.opcfoundation.org/uasdk/help/html/ff342fec-a506-41b9-8b4c-6b131504caf9.htm>
- [26] Aspentech, "Infoplus21." [Online]. Available: <http://www.aspentech.com/core/aspent-infoplus21.aspx>
- [27] CAS, "CAS OPC UA Address Space Designer." [Online]. Available: <http://www.commsvr.com/Products/OPCUA/UAModelDesigner.aspx>
- [28] *NORSOK STANDARD I-005*, Standards Norway Std.
- [29] Siemens, "Snorre ISO15926 Model UML," Saved under UAModel in the PIMAQ UA server folder under Code.
- [30] Microsoft, "Dispose Method," 2011. [Online]. Available: <http://msdn.microsoft.com/en-us/library/system.idisposable.dispose.aspx>
- [31] C. Ragel, "FOREACH Vs. FOR (CSharp)," Internet Article, 4 2004. [Online]. Available: <http://www.codeproject.com/KB/cs/foreach.aspx>
- [32] Stack overflow, "For vs Foreach loop in CSharp." [Online]. Available: <http://stackoverflow.com/questions/1124753/for-vs-foreach-loop-in-c>
- [33] M. Dunn, "The Complete Guide to C++ Strings, Part I - Win32 Character Encodings." [Online]. Available: <http://www.codeproject.com/KB/string/cppstringguide1.aspx>

- [34] OPC Foundation, *OPC UA Part 11 - Historical Access Specification 1.00*, OPC Foundation Std.
- [35] —, *OPC UA Part 3 - Address Space Model specification 1.01*, OPC Foundation Std.
- [36] Unknown, “Sorted Dictionary in Csharp.” [Online]. Available: <http://stackoverflow.com/questions/931891/sorted-dictionary-in-c>
- [37] OPC Foundation, “OPC message Boards.” [Online]. Available: <http://www.opcfoundation.org/forum/index.php>
- [38] —, *OPC UA Part 7 - Profiles 1.00*, OPC Foundation Std., Rev. 1.00.
- [39] —. CTT Help file.

Appendix A

OPC UA project

Part of the attachments.

Appendix B

Server Documentation for Siemens

Part of the attachments.