KAUNAS UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

Ignas Jaruševičius

Research and development of OPC UA based data exchange system

Master's thesis

Supervisor
Prof. Dr. Vytautas Deksnys

KAUNAS, 2015

KAUNAS UNIVERSITY OF TECHNOLOGY

FACULTY OF ELECTRICAL AND ELECTRONICS ENGINEERING

DEPARTMENT OF ELECTRONICS ENGINEERING

Research and development of OPC UA based data exchange system

Master‘s thesis

Embedded Systems (621H61004)

Supervisor
(parašas) Prof. dr. Vytautas Deknys
2015-05-27

Reviewer
(parašas) Doc. dr. Tomas Adomkus
(data)

Author
(parašas) Ignas Jaruševičius
2015-05-27

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Elektros ir Elektronikos fakultetas

(Fakultetas)

Ignas Jaruševičius

(Studento vardas, pavardė)

Įterptinės sistemos (621H61004)

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Research and development of OPC UA based data exchange system“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 15 m. gegužės 27 d.

Kaunas

Patvirtinu, kad mano **Igno Jaruševičiaus** baigiamasis projektas tema „Research and development of OPC UA based data exchange system“ yra parašytas visiškai savarankiškai, o visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka) *(parašas)*

Jaruševičius, Ignas. Research and development of OPC UA based data exchange system. Final project for a Master‘s degree / supervisor Prof. dr. Vytautas Deksnys; Kaunas University of Technology, Faculty of Electrical and Electronics Engineering, Department of Electronics Engineering.

Kaunas, 2015. 64 p.

ABSTRACT

The Internet of Things has found application in many fields. It provides additional services and opens new possibilities by connecting various devices, which are not smart in nature, to the internet. Precision agriculture is one of the fields, which are expected to benefit from the IoT. The main problems with IoT are the absence of internet connectivity in "things" and security concerns. The internet connectivity problem is usually solved by using a gateway device to connect a local network to the internet, while security is still an open issue. In the industrial automation sector security and device interoperability issues are addressed by a new OPC UA standard. In this thesis, an OPC UA based gateway device is developed and its suitability for IoT applications is studied. The main focus is on its applications in agriculture.

During the course of this thesis, an OPC UA based data exchange solution for IoT was developed and was found suitable for agricultural applications. A literature review was done to identify the main data sources and applications of IoT and cloud services in agriculture. Then, the data transfer requirements for these applications were established. An overview of OPC UA standard was done. A flexible model for data aggregation into the OPC UA server from a wide diversity of field devices was proposed, detailed and implemented. The proposed concept was demonstrated to be feasible by aggregating data from various sources (wireless sensors, machinery, camera) into the OPC UA server and then accessing it remotely over the internet. The experiments were carried out to investigate the performance of data aggregation from field devices into the OPC UA server, as well as the performance of data acquisition from the OPC UA server via various connection options (Ethernet, Wi-Fi, VPN). The power consumption of the developed prototype was also tested.

Keywords: OPC UA, Internet of Things, Gateway, Wireless, Precision Agriculture.

Jaruševičius, I. Duomenų mainų sistemos su OPC UA sąsaja sukūrimas ir tyrimas. Magistranto baigiamasis projektas / vadovas Prof. dr. Vytautas Deksnys; Kauno technologijos universitetas, Elektros ir elektronikos fakultetas, Elektronikos inžinerijos katedra.

Kaunas, 2015. 64 psl.

SANTRAUKA

Daiktų internetas yra pritaikomas įvairiose srityse. Jo idėja yra suteikti naujas paslaugas bei atverti naujas galimybes, prijungiant prie interneto įrenginius, kurie įprastai nėra išmanus. Precizinis žemės ūkis yra viena iš sričių, kuriose tikimasi naudos iš daiktų interneto. Pagrindinės daiktų interneto problemos yra prisijungimo prie interneto galimybės nebuvimas „daiktuose“ bei saugumo klausimai. Prisijungimo prie interneto problema paprastai išsprendžiama panaudojant tinklų sąsajos įrenginius, kurie sujungia vietinį įrenginių tinklą su internetu, tuo tarpu saugumo klausimai vis dar yra atviri. Pramoninės automatikos srityje saugumo ir įrenginių tarpusavio suderinamumo klausimus bandoma spręsti naujame OPC UA standarte. Šiame darbe yra projektuojama ir tiriama duomenų mainų sistema, naudojanti OPC UA standartą. Pagrindinis dėmesys yra skiriamas daiktų interneto taikymui preciziniame žemės ūkyje.

Darbo eigoje buvo suprojektuotas OPC UA naudojantis duomenų mainų įrenginys pritaikytas įvairių daiktų interneto sprendimų įgyvendinimui. Tai pat buvo nustatyta, jog jis yra tinkamas taikymui preciziniame žemės ūkyje. Literatūros apžvalgoje buvo išsiaiškinti galimi daiktų interneto taikymai žemės ūkyje, bei iškelti šiems taikymams reikalingi duomenų perdavimo reikalavimai. Tai pat buvo atlikta OPC UA standarto apžvalga. Toliau buvo pasiūlytas ir įgyvendintas lankstus duomenų iš įvairių įrenginių surinkimo į OPC UA serverį modelis. Šio modelio veiksmingumas buvo pademonstruotas surenkant duomenis iš kelių skirtingų šaltinių į OPC UA serverį, iš kurio jie tapo prieinami iš bet kurios pasaulio vietos per internetą. Taip pat buvo atlikti bandymai, kuriuose buvo tiriamas duomenų iš vietinio įrenginių tinklo surinkimo į OPC UA serverį našumas bei duomenų paėmimo iš OPC UA serverio našumas, naudojant skirtingus komunikacijų kanalus. Sukurto įrenginio prototipo energijos sąnaudos taip pat buvo ištirtos.

Reikšminiai žodžiai: OPC UA, Daiktų internetas, Tinklų sąsaja, Bevielis ryšys, Precizinis žemės ūkis.

CONTENTS

ABBREVIATIONS

AES	Advanced Encryption Standard
AP	Access Point
CAN	Controller Area Network
COM	Component Object Model
CPU	Central Processing Unit
CBC	Cipher Block Chaining
DCOM	Distributed Component Object Model
FD	File Descriptor
GPS	Global Positioning System
GPMC	General Purpose Memory Controller
GSM	Global System for Mobile communications
GUID	Globally Unique Identifier
IoT	Internet of Things
ISO	International Organization for Standardization
OLE	Object Linking and Embedding
OPC	OLE for Process Control
OPC UA	OPC Unified Architecture
OS	Operating System
PA	Protocol Adapter
PPP	Point-to-Point Protocol
RAM	Random Access Memory
RTD	Round Trip Delay
SHA	Secure Hash Algorithm
SDK	Software Development Kit
SOM	System on Module
UART	Universal Asynchronous Receiver/Transmitter
UML	Unified Modelling Language
USB	Universal Serial Bus
VPN	Virtual Private Network
XML	Extensible Markup Language
WAN	Wide Area Network
WAP	Wi-Fi Protected Access
WSN	Wireless Sensor Network
WSAN	Wireless Sensor and Actuator Network

1. INTRODUCTION

The recent development in embedded systems and wireless connectivity has led to the large scale deployment of the Internet of Thing (IoT) networks [1]. The IoT has found applications in many fields such as environmental monitoring, energy management, manufacturing, building and home automation, transportation, medical and healthcare systems, smart cities, etc. Precision agriculture is also one of the fields that are expected to benefit from the IoT [2]. Connecting various field devices to the internet would allow a large amount of agricultural data to be gathered and processed by cloud based systems. The results of the data analysis would be used to improve the yields by predicting disease outbreaks and controlling fertilization, spraying, and irrigation [3]. A cloud based system would also have a benefit of providing a farmer with a unified interface for management of resources and field equipment. The main difficulties in adapting the IoT for agriculture are security concerns [4] and a lack of internet connectivity in the existing field devices. To connect such devices to the internet an additional gateway device, preferably with a standardized data transfer protocol, is required.

OPC UA (OLE (Object Linking and Embedding) for Process Control: Unified Architecture) is a fairly new standard, which aims to solve interoperability issues in the industrial automation sector [5]. It provides object-oriented information modelling capabilities, a service-oriented communication model and a security mechanism, which fulfils requirement for data transfer at enterprise level. OPC UA is well suited for integration with the IoT systems and should also be applicable in the agricultural sector.

In this thesis an architecture for data transfer between field devices and information systems (usually cloud based) using OPC UA based IoT gateway is presented and studied. OPC UA based data transfer solves standardization and security issues in the IoT model. The proposed architecture could be applied in various fields, however the main focus of this thesis is an on-line data transfer from agricultural field devices over a mobile network.

This thesis is made as part of CLAFIS research project. The project focuses on integrating research results, experience, and technology from automation, IoT, and agriculture sectors. The goal of the project is a cloud based integration platform that combines existing knowledge with data mining and analysis to reduce production costs as well as increase profitability and competitiveness of the European agriculture.

1.1. Objective

This thesis focuses on data transfer from various agricultural field devices using an OPC UA based IoT gateway. The main objective is to develop an IoT gateway prototype running an OPC UA server, and research its suitability for such task. This thesis attempts to accomplish the following tasks:

- Investigate data sources and possible applications of data analysis in agriculture
- Establish the requirements for data transfer from field devices to cloud services.
- Propose a flexible and secure solution for data aggregation from field devices.
- Research the performance of data aggregation from field devices into the OPC UA server;
- Research the performance of data acquisition from the OPC UA server running on the IoT gateway;

1. DATA TRANFER BETWEEN FIELD DEVICES AND CLOUD SYSTEMS

In this chapter, a literature review is done to identify the main data sources in agriculture. Additionally, the applications, which could benefit from IoT infrastructure (Fig. 1.1) and cloud services, as well as data transfer requirements for these applications are analysed.

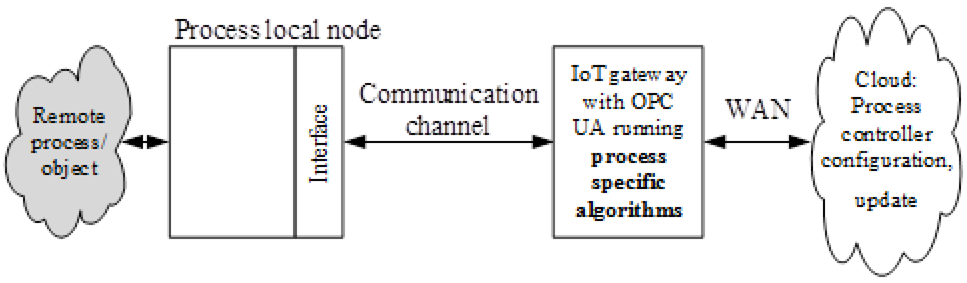

Process local node
Remote process/ object
Interface
Communication channel
IoT gateway with OPC UA running process specific algorithms
WAN
Cloud: Process controller configuration, update

Fig. 1.1. The data flow in the IoT infrastructure

The data flow in the IoT based system is shown in Fig. 1.1. Sensors and actuators are connected to a local network (usually wireless), while the IoT gateway connects the local network with a wide area network (WAN). The cloud services access the IoT gateway via the WAN and is responsible for data storage and analysis. The IoT gateway may also run some data filtering or automation algorithms locally to decrease bandwidth and response time requirements for the WAN connection. These algorithms could be updated or their parameters could be adjusted from the cloud.

1.2. Data sources in agriculture

1.2.1. Wireless sensors and actuators networks in agriculture

In agriculture, monitoring of environment may be used to improve yields and reduce costs by predicting decease outbreaks, managing irrigation, and fertilization [3]. The observation of various parameters is usually implemented using wireless sensor networks (WSN), which are researched in many scientific publications [7-12]. Some processes such as irrigation or greenhouse climate control may also be automated by using wireless sensor and actuator networks (WSAN), as shown in wireless sensor application reviews [8][10]. In the livestock sector, WSN are used to maintain good animal health by monitoring climate-related variables in animal houses or by attaching health-monitoring systems to animals [8]. The typical structure of a WSN consists of multiple sensor nodes and a sink node, which is used for long range data transfer (Fig. 1.2). The publications on WSN topic suggests that the sensor nodes usually communicate using 2.4 GHz or 868 MHz radio, while the gateway node uses GSM or a wired connection.

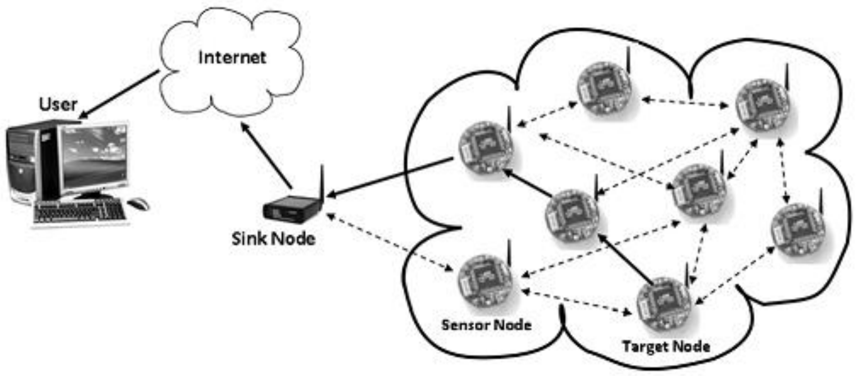


Fig. 1.2: The typical structure of WSN [10]

Some notable commercial WSN systems are:

- The *"VineSense"* system [13], which is targeted at vineyards, and is based on the research [12] done by the University of Florence.
- The solution from the *"Libelium"* company, which consists of *"WaspMote"* [14] sensor nodes and the *"Meshlium"* gateway [15].

1.2.2. Agricultural machinery

Agricultural machinery such as tractors, harvesters and various implements are another source of data. The data from the machinery can be obtained through Controller Area Network (CAN) physical interface. The message format and content [19] on the CAN bus is based on J1939 standard. Going further, there is J1939 based ISO 11783 (ISOBUS) standard which is specifically aimed at agricultural machinery. It defines specific messages for tractors [16] and implements [17], as well as implement communication with a virtual terminal [18] in the cabin of a tractor. The ISOBUS standard is designed to solve interoperability problems with modern agricultural machinery by providing a common way of communication between agricultural equipment.

Modern tractors and harvesters are also equipped with Global Positioning System (GPS) and can provide information about resources used or gathered. In addition to this, it possible to apply a different amount of fertilizer, water, seeds, pesticides, herbicides and other resources to particular parts of the field by using ISOBUS task controller [20].

1.3. Applications of agricultural data analysis

1.3.1. Decease control

The idea of decease control is to predict decease outbreaks by analysing micro-climatic data in a crop field [21]. Modern methods of developing prediction models utilizes machine learning techniques [22], which require lots of statistical data to train the model. In case of the IoT based solution, the statistical agricultural data can be gathered to a cloud service from multiple farms, thus simplifying the process of decease prediction model creation.

The decease control allows to improve the crop yields and reduce the amount of chemical used for spraying.

1.3.2. Variable application rate technology

Soil and crop conditions within a field may vary from place to place and also during and between growth seasons. The most important variabilities are [23]:

- The variation of the yield amount between seasons and its distribution within a field
- The variation in field topography (elevations, slopes, streams, etc.)
- The variation of soil fertility (nutrient content), physical properties (texture, density, moisture content, etc.) and chemical properties (pH, organic matter, salinity water-holding capacity, etc.).
- The different types of crops: crop density, height, nutrient stress, water stress and biophysical properties (leaf-area index, intercepted photo synthetically active radiation, crop leaf chlorophyll content, and crop grain quality).
- The variation of weed, pest and decease infestations, wind and hay damage.

The variabilities are addressed by applying different amounts of seeds, fertilizer, herbicides, etc. to the different parts of a field. In general, there are two approaches to managing the variabilities: the map-based approach and the sensor-based approach. The map-based approach uses a pre-generated map to control a variable-rate applicator, while the sensor-based approach uses real-time sensors to adjust application rate on the fly. The map-based approach is well suited to be used with cloud-based services and ISOBUS task controller. An ISOBUS task file (application rate map) could be generated by a cloud service and then uploaded to the task controller for execution [20] [24].

|In this use case scenario, the main task of cloud services is to analyse data from crop fields using sophisticated decision making models and attempt to generate optimal seeding,

fertilization and spraying maps. In addition to this, large amounts of data gathered from various farms could be used to improve the decision making models. The relevant data could be gathered from both WSN and machinery. WSN could provide data about weather and soil conditions, while machinery could provide data about seeding, fertilization, spraying, and yield amount in the particular parts of crop fields.

1.3.3. Irrigation and greenhouse climate control

The conservation of water is very important in some counties (e.g. Egypt) [26]. Some studies show that intelligent irrigation control can reduce amount of water used by 30-60% [27]. The irrigation control algorithm needs to be run locally on the IoT gateway, but some parameters may be changed remotely. The data could also be logged and transferred to cloud services. In general, the irrigation control should be based on plant type, its growth stage, environmental sensors (soil humidity, air temperature, solar radiation, etc.) readings, and weather forecasts.

A similar approach can also be used for greenhouse climate control systems, but in this case there would be more types of sensors and controllable parameters (Fig. 1.3). In both cases data logging plays an important role, as it can be analysed to improve control algorithms over the time.

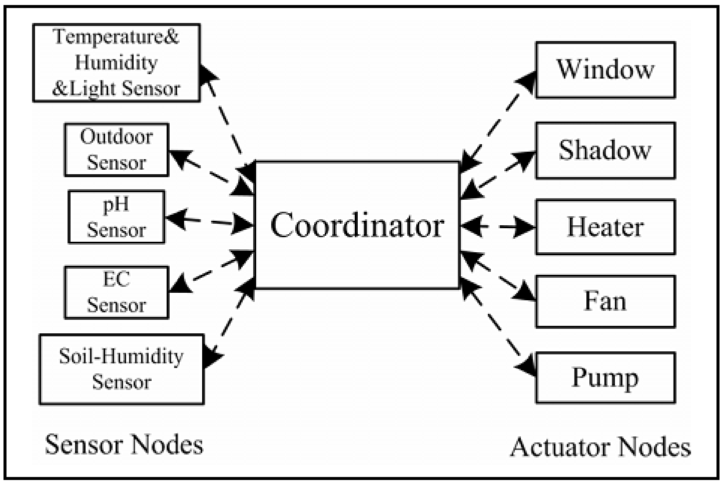


Fig. 1.3: The structure of greenhouse climate control system [28]

1.3.4. Fleet management

The main concern of fleet management is managing the operation of a machine fleet in an optimized manner. It includes resource allocation within machine fleet, schedule planning, and routing machines while performing field operations or moving between locations. The IoT based fleet management solution allows to track machines in real-time, log total working hours, area, and resources used. Furthermore, it would allow to upload task files to the task controller remotely (normally it is done using USB stick).

The maintenance of machines is another important issue in fleet management. Agricultural operations has a short time window, which requires to detect, diagnose and repair machine faults as quickly as possible. Constantly monitoring machine fault codes would allow to detect early symptoms of fault before machine malfunctions. It can be implemented by logging the fault information from machines and transferring it to a cloud-based maintenance service for analysis.

1.4. Requirements for data transfer

1.4.1. Local wireless network

The local wireless network is used to obtain data from distributed sensors, which are used to monitor environment. The high data rate is not important in this case, as environmental parameters are slowly changing and, for application like decease management, sampling interval of 1 hour should be enough, as stated in publications [3][25]. Meanwhile, weather monitoring services usually update weather parameters at 10-15 minute interval, so this interval could also be used. For automation purposes, shorter sampling interval may be required, for example, publication on greenhouse automation [28] states sampling period of 1 minute. However, even in the latter case the packet rate requirements are not high. The gateway would only have to be able to send request at ~17 Hz rate to poll 1000 nodes at 1 minute interval. So it can be assumed that the requirement for the local wireless network is 15-20 Hz polling rate (polling single node).

1.4.2. Internet connection

As discussed earlier, data acquisition from distributed sensors does not require high transfer rates. The amount of data transmitted can be further reduced by transferring data only when parameter value changes.

The upload of a task file to machinery and reading diagnostics information [29] does not put high requirements on the internet connection either, as these tasks should be only performed on-demand. Active fault codes (if there are any) are transferred on the bus every second.

The highest data throughput requirement comes from real-time machine tracking and application or yield rate logging. This requires to transfer position, speed, application rate, and possibly some other parameters. The repetition rates, defined in the machinery standards [17][20], for messages with relevant parameters are in the range of 1-10 Hz (actual applications may require a lower rate). Also, active fault codes (if there are any) are transferred on the bus every second. Taking this into account, it can be assumed that transferring 10 parameters at 10 Hz rate should be enough in most cases.

1.4.3. Temporary data storage

Some temporary data storage is necessary in case the internet connection is not available. This can be achieved by utilizing OPC UA services, which are discussed in chapter 2.2. For example, historical access service [30] allows to store historical data of value, which can be read at any time. Meanwhile, subscription service [31] automatically queues messages for transfer to ensure data delivery in case the connection is temporarily not available.

2. OPC UNIFIED ARCHITECTURE

Conventional OPC is a vendor independent specification set which allows systems to communicate with each other by monitoring and writing parameters, sending and receiving event notifications, and providing historical data. It was designed to solve interoperability issues between systems, devices and software from different vendors. While the conventional OPC solved the interoperability problem and achieved a high adaption rate in the industrial automation sector, it has many limitations. The most notable limitations of the conventional OPC are dependency on Microsoft Component Object Model (COM) and Distributed COM (DCOM) technologies and insufficient security level for enterprise level applications. These issues among many others were addressed in the next OPC standard named OPC Unified Architecture (OPC UA) [5]. OPC UA can be run on non-Microsoft operating system platforms, including embedded devices. It uses a service oriented architecture based on open protocols, where all OPC data models (data access, alarms and events, historical data access …) are exposed by a single set of services. OPC UA features an object oriented information modelling which enables efficient and reliable transfer of higher-level structured data. The security model in OPC UA provides user authorization functionality and ensures confidentiality, integrity, and authenticity of the data transferred [36].

2.1. Address space model

The OPC UA address space is a set of objects, which an OPC UA server makes available to OPC UA clients [32]. It is designed to convey the information of both process data and metadata to an OPC UA client. The address space consists of nodes and references, which form a full mesh network. The nodes represent various entities such as variables, objects and data types, while the references represent relationships between these nodes. The whole address space is represented to a client as a hierarchical tree which starts at the root node.

OPC UA uses object-oriented modelling, where nodes of different classes represent different elements of the object model (Fig. 2.1). The OPC UA services are used to access the objects and their components, like reading or writing a variable value, calling a method, or receiving events from the object. The browse service can be used to explore relationships between objects and their components.

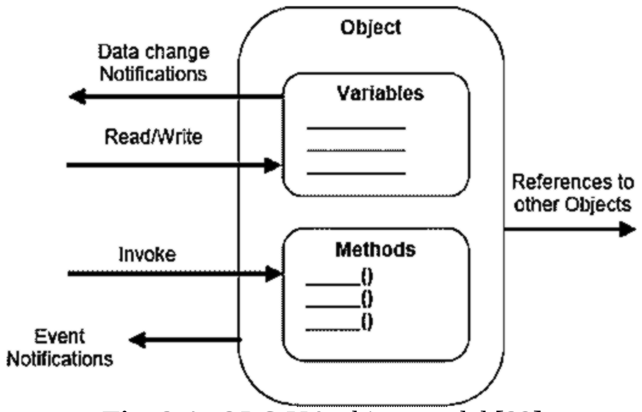


Fig. 2.1: OPC UA object model [32]

2.1.1. OPC UA Nodes

A node in OPC UA is a data structure which consists of a predefined list of attributes and a varying number of references (Fig. 2.2). The exact number of attributes depends on a node class which the particular node belongs to, however the following attributes are common for all node classes:

- NodeID – a structure used to uniquely identify a node in the address space. It consists of a namespace index (16-bit integer) and a unique identifier. The unique identifier must be unique within the namespace and it can be one of the following: a 32-bit integer, a string or a 128-bit globally unique identifier (GUID).
- NodeClass – identifies a node class the node belongs to. OPC UA specification defines 8 node classes:
    - Variable – represents a value.
    - Object – represents a real-world or software object, system component, etc.
    - Method – represents a function that can be called by an OPC UA client.
    - View – organizes a subset of nodes in the address space.
    - DataType – represents a data type.
    - VariableType – represents a variable type
    - ObjectType – represent an object type (e.g. boiler).
    - ReferenceType – represents a reference type.
- BrowseName – a string which is used to identify paths in server's address space.
- DisplayName – the name of a node, which is displayed to an OPC UA client.
- Description – an optional explanation of the meaning of a node.

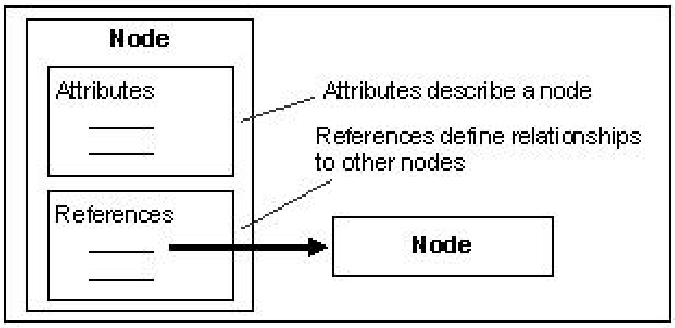

Node
Attributes
References
Attributes describe a node
References define relationships to other nodes
Node

Fig. 2.2: OPC UA node structure [32]

2.1.2. Type definitions

Type definitions are nodes which belong to DataType, VariableType, ObjectType or ReferenceType class. These nodes are supposed to present metadata to OPC UA clients. Each type definition class forms a separate hierarchical tree in the OPC UA address space which starts at the base type node. OPC UA specification provides built-in hierarchies of type definitions which can be extended by application using inheritance, thus creating new customized node types. As an example of node type hierarchy, the tree of built-in OPC UA variable types is shown in Fig. 2.3.

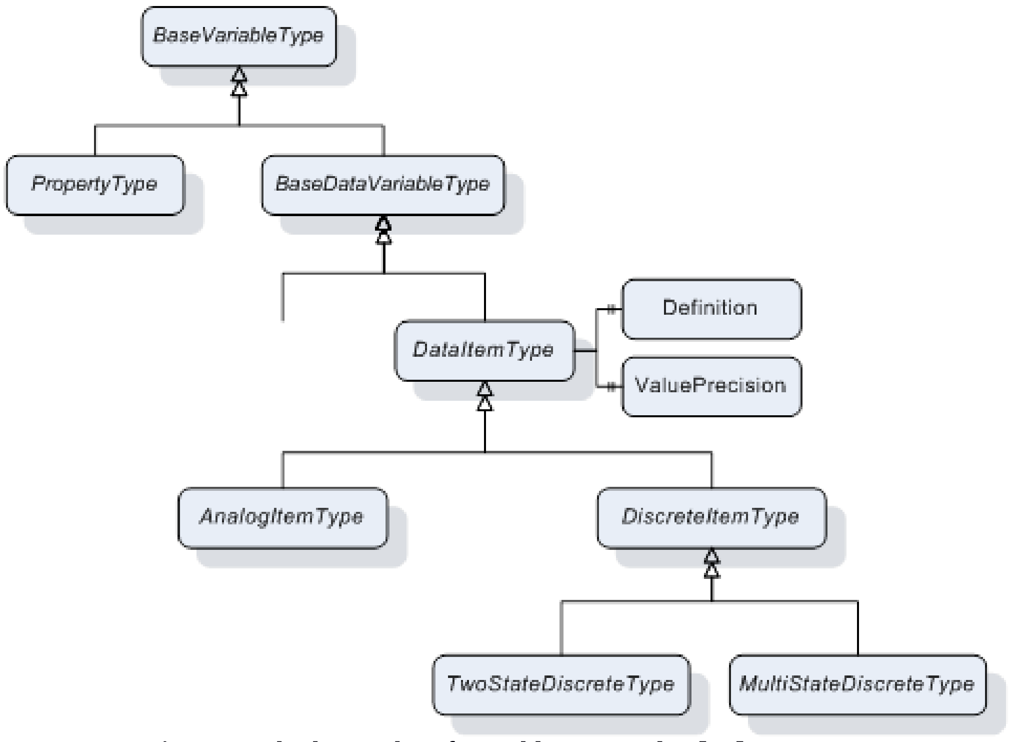


Fig. 2.3: The hierarchy of variable type nodes [33]

2.1.3. References

References are used to create relationships between nodes in the address space. They are defined by the nodes of ReferenceType class. Unlike other OPC UA types, references are not instantiated in OPC UA address space, but they are added to other node instances and are part of them. The standard references are divided into two distinct groups: the hierarchical references and the non-hierarchical references. The hierarchical references are used to build the information architecture and can be discovered as branches/paths by the browsing mechanism in a client. An example of hierarchical relationship could be a parent-child relationship between a folder and the items in that folder. The non-hierarchical references are used to build dependencies between elements which indicate that the reference source element depends on the reference target element. An example of such relationship could be a variable (source) and its data type relationship (target). These relationships convey metadata of the nodes and the browsing mechanism in a client should not create branches/paths based on the non-hierarchical references. The hierarchy of build-in OPC UA reference types is shown in Fig. 2.4.

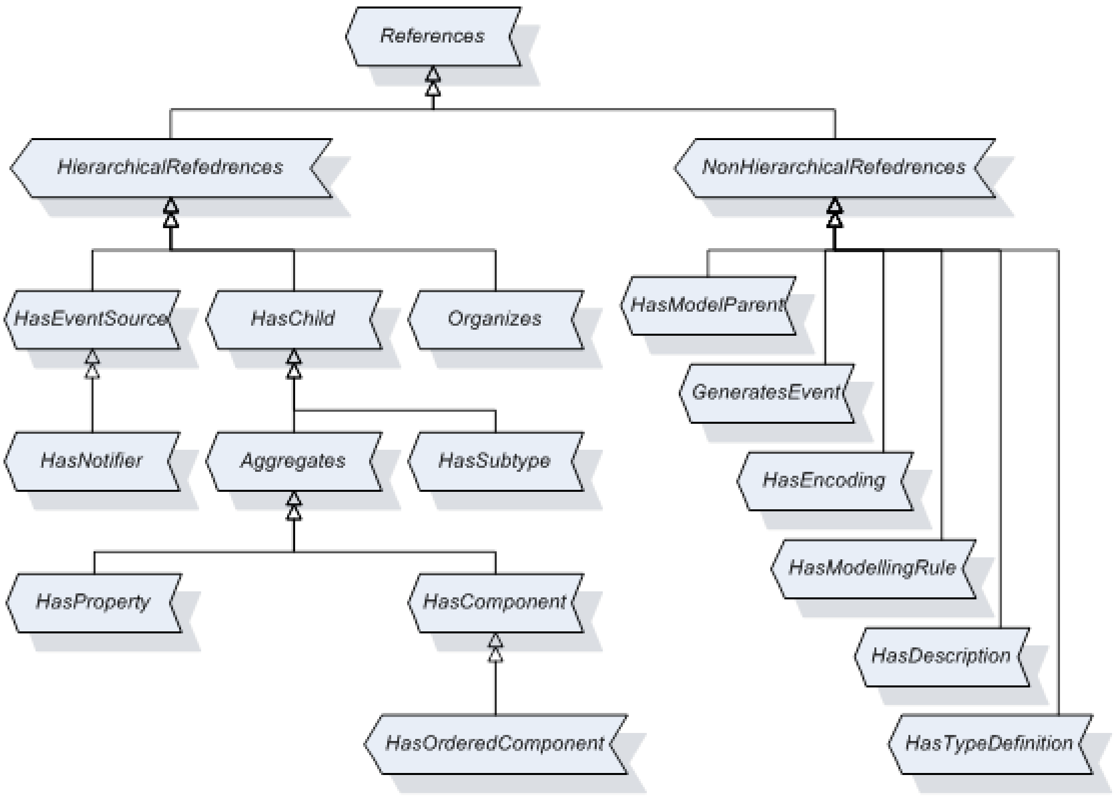


Fig. 2.4: The hierarchy of reference type nodes [33]

2.1.4. Variables

The nodes of the Variable class are used to represent values in the address space. There are two major types of variables nodes – data variables and properties. The property nodes provide additional characteristics (metadata) of other OPC UA nodes (for example, allowed range of variable or engineering units), while the data variable nodes represent the value itself. The main purpose of such differentiation is to convey additional information to an OPC UA client about how the value should be interpreted.

The additional attributes which are specific to the Variable class are listed in Table 2.1.

Table 2.1: Attributes specific to the Variable class nodes

Attribute	**Description**
Value	The most recent value of the variable.
DataType	Data type of the Value attribute.
AccessLevel	Indicates how the Value can be accessed (readable, writeable). It should represent the nature of the underlying hardware.
UserAccessLevel	User rights to access the Value attribute.

The *Value* attribute is a structure, which is defined in OPC UA part 6 [37] and incorporates Value, Status (Quality) and Timestamp (source and server) information.

2.1.5. Object nodes

The primary use of the object nodes is to organize other nodes (they act like folders). Object nodes usually represent real-world or software objects, system components, etc. Each object node must have a reference to its type definition (ObjectType class node). Another important feature of object nodes is an ability to send event notifications to OPC UA clients.

Besides the common attributes, objects expose one additional attribute:

Table 2.2: Attributes specific to object class nodes

Attribute	**Description**
EventNotifier	Indicates if the object provides events or historical events. Reading and writing historical events may be supported.

2.1.6. Method nodes

The method nodes expose commands that can be triggered by an OPC UA client. The actual actions performed, when a specific method node is activated, are entirely up to the developer of an OPC UA server. The methods can have a return value and variable number of input arguments.

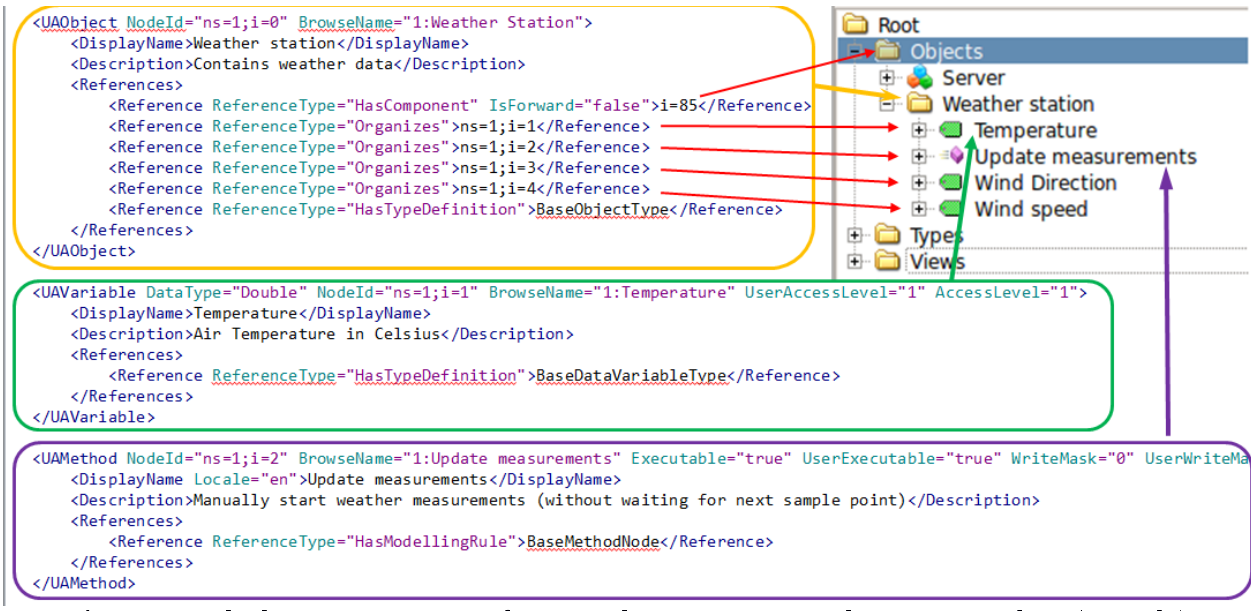
Besides the common attributes, method nodes expose the attributes listed in Table 2.3.

Table 2.3: Attributes specific to method class nodes

Attribute	**Description**
Executable	Indicates if the method is executable; it does take user rights into account. Typically, methods are executable, unless their execution is somehow disabled by the underlying hardware.
UserExecutable	Indicates if the current user has permissions to execute the method.

2.1.7. OPC UA Nodeset 2.0 XML files

Nodes in an OPC UA server can be instantiated utilizing the XML files that are formed according to the OPC UA Nodeset 2.0 specification [33]. This feature provides a standard way of adding nodes to an OPC UA server at the runtime. The XML file is basically a list of OPC UA nodes. An example of node definitions in the XML file is provided in Fig. 2.5. The definition of each node starts with a node class, which the node belongs to. It is then followed by initialization of the node attributes and a list of references to other nodes.



```
<UAObject NodeId="ns=1;i=0" BrowseName="1:Weather Station">
    <DisplayName>Weather station</DisplayName>
    <Description>Contains weather data</Description>
    <References>
        <Reference ReferenceType="HasComponent" IsForward="false">i=85</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=1</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=2</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=3</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=4</Reference>
        <Reference ReferenceType="HasTypeDefinition">BaseObjectType</Reference>
    </References>
</UAObject>
<UAVariable DataType="Double" NodeId="ns=1;i=1" BrowseName="1:Temperature" UserAccessLevel="1" AccessLevel="1">
    <DisplayName>Temperature</DisplayName>
    <Description>Air Temperature in Celsius</Description>
    <References>
        <Reference ReferenceType="HasTypeDefinition">BaseDataVariableType</Reference>
    </References>
</UAVariable>
<UAMethod NodeId="ns=1;i=2" BrowseName="1:Update measurements" Executable="true" UserExecutable="true" WriteMask="0" UserWriteMa
    <DisplayName Locale="en">Update measurements</DisplayName>
    <Description>Manually start weather measurements (without waiting for next sample point)</Description>
    <References>
        <Reference ReferenceType="HasModellingRule">BaseMethodNode</Reference>
    </References>
</UAMethod>
```

Fig. 2.5: Node descriptions in XML format and representation in the OPC UA client (top right)

2.2. Services

All communications between an OPC UA server and a client are based on services [34]. The OPC UA specification defines functionality of services in abstract way without binding it to specific programming languages or environments. These abstract services might be mapped into WEB services or TCP/IP services. To communicate with each other, applications are required to support the same mapping, but they can be written using completely different programming languages. OPC UA defines a total of 34 services, which are grouped into the following service sets:

- Discovery – allows clients to discover OPC UA servers and their connection information
- Secure Channel – allows OPC UA clients to establish secure connection to a server
- Session – provides the client authentication functionality
- Node management – allows OPC UA clients to modify the address space
- View – allows clients to browse the address space of an OPC UA server

- Query – allows clients to apply complex filters when browsing the address space
- Attribute – allows reading and writing of node attributes
- Method – allows clients to call methods
- Monitored Item – monitors nodes and generates notifications
- Subscription – queues and sends notifications to the subscribed clients.

Some services related to data access and security are more important in context of this thesis and are described in more detail in the further sub-chapters.

2.2.1. Attribute service set

The attribute service set provides some of the key services that are implemented by the most, if not all, OPC UA servers and clients. The read and write services, which belong to this set, provides an ability to read or write the value of any node attribute. While these services can be used to access any attribute of any node type, they are mainly used to read or write the *Value* attribute of the Variable class nodes. This is the simplest data access technique in OPC UA, which provides an ability to send or read data from a device in on-demand fashion.

This service set also includes historical data services - HistoryRead and HistoryUpdate. These services allow a server to expose historical data of nodes [30]. However, OPC UA specification only defines the methods, how historical data should be accessed and transferred, but is does not define the way, how it should be stored on the OPC UA server. This allows developers of an OPC UA server to keep historical data however they want.

2.2.2. Subscription mechanism

The subscription mechanism of OPC UA utilizes Monitored Item and Subscription service sets [31]. The Monitored Item service set is used to generate and/or filter notifications, while the Subscription service set is responsible for delivering these notifications to the client that made the subscription.

Subscriptions to variable values allows the client to monitor server variables without constantly polling them. The value change notifications are generated by Monitored Item service set based on the parameters set by the client, which include: a sampling interval, value dead band levels (filter), and one of the three change detection modes (the change of a value status, value data, or a value timestamp). The sampling interval is defined as the best-effort and is limited by the actual value update interval on the server. Compared to polling using Read service, this mechanism greatly reduces amount of data transferred and is the recommended way to obtain data from a server.

It is also possible to subscribe to events. Subscriptions to events woks in a similar way, however in this case Monitored Item service set does not generate notifications itself, but is only responsible for filtering of already generated event notifications.

The structure of the subscription mechanism is shown in Fig. 2.6. The value change or event notifications are queued by the Monitored Item service set. The queues are used for two purposes. The first one is to combine multiply notifications into one notification message. The second one is to buffer data in case the connection with the client is lost. Subscription service set takes notifications from the Monitored Item queues and sends them to the OPC UA client. A publishing interval defines how often messages are sent to the client and is usually longer than the sampling interval of a variable value, which makes several notifications to be sent to the client in one message.

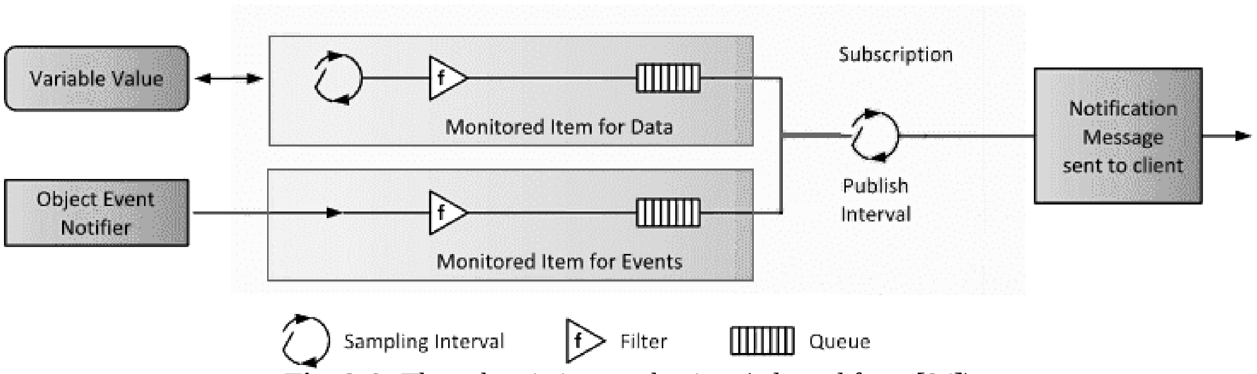


Fig. 2.6: The subscription mechanism (adapted from [34])

2.2.3. Secure channel

The secure channel service set provides an ability to establish a secure connection which ensures data confidentiality, message integrity, and message authenticity. The procedure of secure channel creation is shown in Fig. 2.7. It requires both the client and the server to authenticate. The certificates conforming to X509 standard are used for this purpose. Once the secure channel is established, there are three possible security modes which can be used for the data exchange:

- ‘None’ – no security is applied but unique logical channel is maintained. This security mode can only be trusted when operating in a secure physical network or a low level security protocol such as IPsec is used.
- ‘Sign’ – all messages between the server and the client are signed using their private keys. This ensures the data integrity and the message authenticity.
- ‘Sign and encrypt’ – all messages are signed and their contents are encrypted. Compared to previous method, this methods additionally ensures the data confidentiality.

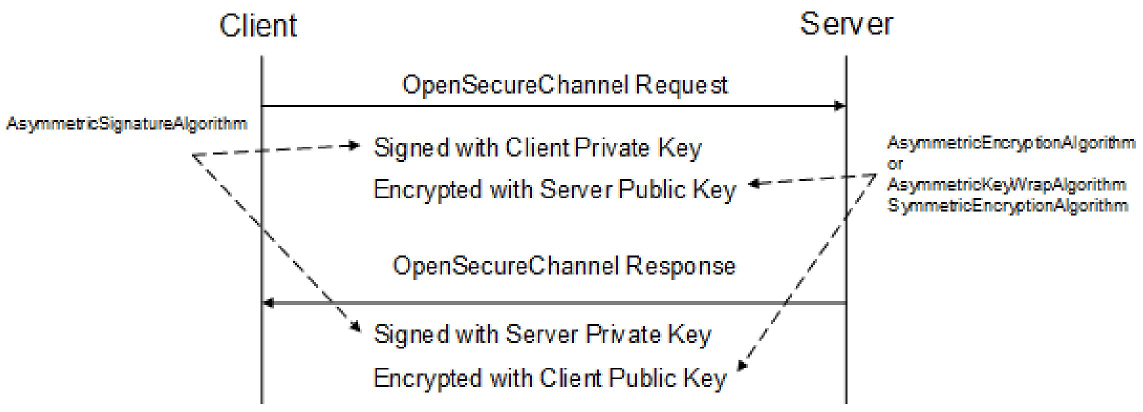

Client
Server
OpenSecureChannel Request
AsymmetricSignatureAlgorithm
Signed with Client Private Key
Encrypted with Server Public Key
AsymmetricEncryptionAlgorithm
or
AsymmetricKeyWrapAlgorithm
SymmetricEncryptionAlgorithm
OpenSecureChannel Response
Signed with Server Private Key
Encrypted with Client Public Key

Fig. 2.7: Establishment of a secure channel

2.2.4. Session service set

The secure channel establishment is followed by a session activation. During this procedure a user is identified. The user identity is required to determine what resources are allowed to be accesses by the client. The proof of the user identity, which the client must pass to the server during creation of session, is called a security token. OPC UA supports three types of security tokens:

- User name – allows clients to authenticate by passing a username and a password (or the hash of a password) to a server
- X509 – allows clients to authenticate using X509 certificate. This certificate is usually not the same as the one used to create a secure channel between a client and a server (it can be the same).
- Kerberos – allows a user to authenticate using trusted third party which issues Kerberos tokens. This model avoids the passing of a user credentials directly to an OPC UA server.

Once a user identity is established, the user access right assignment is implementation dependent and is responsibility of an application developer.

3. SYSTEM ARCHITECTURE

This chapter describes the system architecture of the proposed IoT gateway solution. It presents an overview of hardware and software components in the gateway and focuses on a software model for data aggregation from various devices into the OPC UA address space.

The task of the IoT gateway is to collect information from various devices and sensors and expose it via OPC UA server. The devices attached to the gateway communicate using various physical interfaces and protocols, which means that every new device requires a piece of software which interprets data from a particular device and places it into the address space. One way of solving this issue is implementing specific functions for communication with external devices in the OPC UA server. However, this requires the OPC UA server to be recompiled in order to add support for a new device type. To remove the necessity of OPC UA server modification, it was decided to introduce an additional abstraction layer, based on protocol adapters (PA). The purpose of PA is to communicate with external devices and fill the address space using a unified protocol (which is defined further in this thesis). This architecture allows adding support for a new type of devices to the gateway without modifying or even restarting the OPC UA server. The data flow from a sensor to an OPC UA client in the proposed data acquisition model is shown in Fig. 3.1:

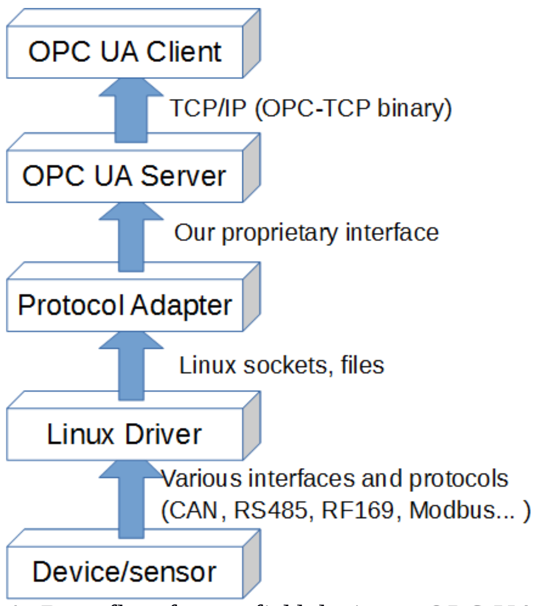


Fig. 3.1: Data flow from a field device to OPC UA Client

3.1. IoT Gateway hardware

The IoT gateway is built around VAR-SOM-AM33 system on module (SOM) [38]. The SOM includes ARM cortex A8 processor working at 1 GHZ frequency, 512 MB of DDR3 random access memory, 512 MB of NAND flash memory, and a Wi-Fi module.

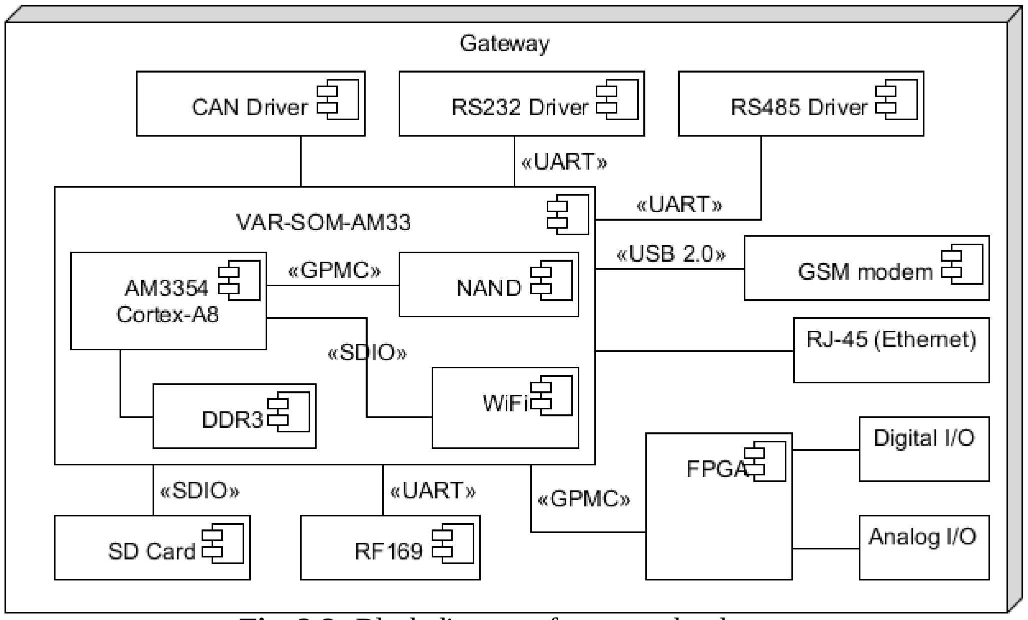


Fig. 3.2: Block diagram of gateway hardware

The block structure of the IoT gateway hardware is shown in Fig. 3.2. The SOM is placed into motherboard, which also hosts a GSM modem module and a 169 MHz radio module. Both of these modules are pluggable. The GSM module is connected to the SOM via USB 2.0 interface, while UART interface is used for communication with 169 MHz radio module. The motherboard also has driver chips for various peripheral interfaces (CAN, RS232, RS485), which convert CMOS voltage level signals coming from the SOM to appropriate bus voltage levels. The analog and digital I/O capabilities in the IoT gateway are realized via FPGA. The FPGA is connected to the SOM using general purpose memory controller (GPMC) interface, which ensures high data transfer rate as it uses 8-bit wide parallel bus. The gateway can be powered from 8-36 V voltage power source.

The picture of the IoT gateway prototype is presented in Fig. 3.3. The SOM can be seen in the middle of the board. The connectors used in gateway provide humidity and dust protections, as it must be suitable for operation in outdoor conditions.

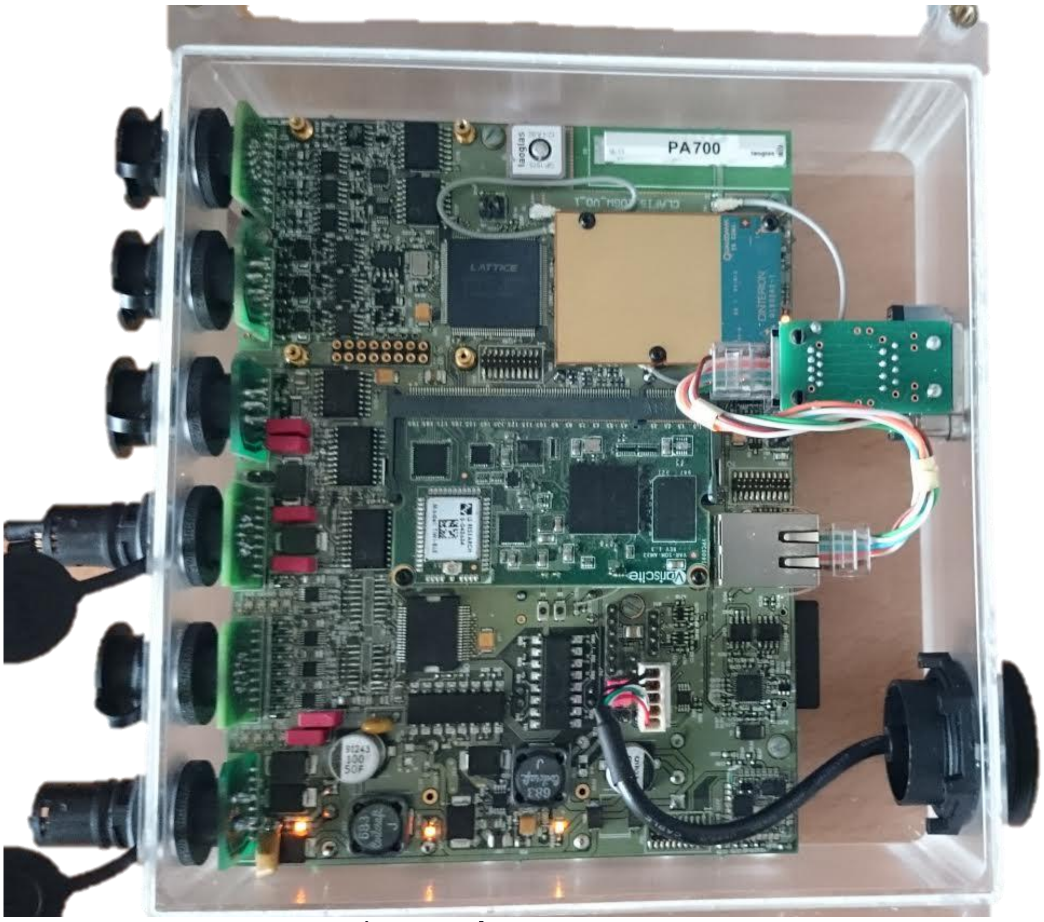


Fig. 3.3: The IoT gateway prototype

3.2. IoT Gateway software modules

The gateway runs Linux Debian distribution, which gives us an ability to easily install and run a lot of the existing Linux software. The main software components of the IoT gateway are presented in Fig. 3.4 and are discussed in more detail in the further sub-chapters.

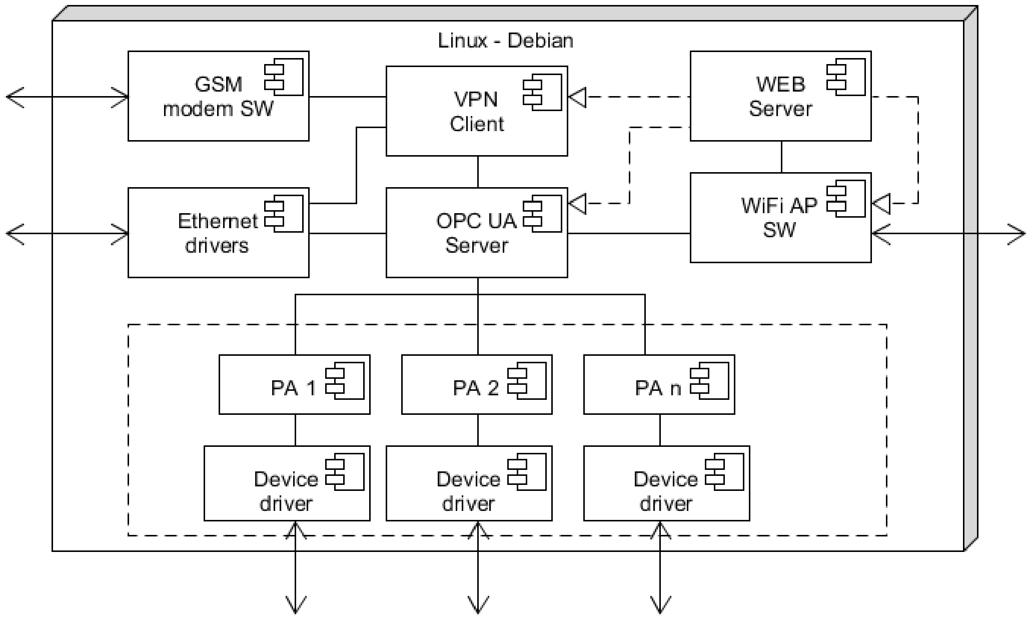


Fig. 3.4: The main software blocks of the IoT gateway

3.2.1. Virtual private network client

The gateway runs an OPC UA server, which means that it needs to have a static IP address. This is not a problem if the gateway resides on a local network and is accessed only locally. However, the task of the IoT gateway is to expose data to cloud services and it is usually done via GSM connection that does not provide a static global IP address. To cope with the static server IP address requirement, a virtual private network (VPN) client is installed on the gateway. We chose to use OpenVPN [39] software as it is free and available on many platforms. In addition to providing an ability to assign a virtual static IP address, OpenVPN also adds a layer of security, as all transferred data is encrypted and only authenticated users are able to enter the VPN (X.509 certificates are used for the authentication).

3.2.2. GSM connectivity software

The GSM modem requires USB virtual serial port drivers, GSM modem drivers, and point-to-point protocol (PPP) support. These drivers are included in kernel provided by Texas Instruments (TI) Linux Software Development Kit (SDK), however they are not selected by default, so it is necessary to manually select them using kernel configuration tool and recompile the kernel. In addition to this, a point-to-point protocol daemon [40] has to be installed in the system. The configuration settings (e.g. SIM PIN code, number to dial when connecting to the internet) for the PPP daemon will be supplied through the web-page running on the IoT gateway.

3.2.3. Web server

The purpose of the web page running on the gateway is to provide a user-friendly way of configuring the gateway. It is used to configure GSM modem settings, OPC UA server conditional access rights and Wi-Fi access point Service Set Identifier (SSID) and password. Two different web servers were tested on the gateway: the Apache 2 web server [41] and the lighttpd web server [42]. The Apache 2 web server has more features but requires quite a lot of resources (considering this is an embedded device), while the lighhtpd is a very simple web server designed to run with minimal resource usage. The testing showed that the lighttpd web server provides all the necessary functionality to run the maintenance web-page, so it was decided to settle with this web server solution.

3.2.4. Wi-Fi access point software

The gateway creates a Wi-Fi access point (AP), which allows a user to connect to the gateway using a mobile device. The access point can be used to access configuration web-page or to obtain data from the OPC UA server. The software required for hosting Wi-Fi access point includes the following:

- Wi-Fi module firmware provided by Variscite
- Wireless connectivity drivers provided by TI
- User-space software for AP configuration (hostapd)
- DHCP server for IP address assignment to connected devices (udhcpd was used)

3.2.5. OPC UA server

The OPC UA server exposes all the data collected from the devices connected to the gateway in its address space. The server was developed using OPC UA server SDK for Linux from Softing. It provides data access, historical data access and methods capabilities. User authentication and authorization based on X.509 certificates was also implemented. The access rights of particular users are configured by the gateway owner though the configuration web page.

The server implementation features a dynamic address space that is loaded from Nodeset 2.0 compliant XML files. This approach was chosen to facilitate support for a wide diversity of devices. A static address space compiled into the server would allow for a much better representation of individual devices, but such an approach is impractical when dealing with multiple devices from different vendors.

3.2.6. Device drivers and protocol adapters

Protocol adapters and device drivers are used to connect various devices to the gateway and expose their data in the address space. The protocol adapters for various devices may differ greatly from one another, but they must interface with the OPC UA server using the common protocol, which is described in 3.3. For concept testing purposes some low-complexity PAs were developed and are detailed in chapter 4.

The gateway kernel is configured to include a lot of commonly used device drivers (CAN, RS232, etc.). In case some specific driver is required, it may be compiled as a loadable kernel module and installed into gateway, as the kernel is compiled with loadable kernel module support.

3.3. Communication interface between OPC UA server and protocol adapters

Protocol adapters are software applications that communicate with various external devices and/or sensors using particular physical interfaces and protocols. The task of a PA is to translate between a protocol used by an external device and the OPC-PA protocol which is described further in this chapter. The proposed data exchange model between PA and the OPC UA server can be split into three main parts:

1) The OPC UA address space is loaded from OPC UA NodeSet 2.0 compliant XML files.
2) The PA connects to the OPC-UA server and obtains its namespace identifier.
3) The PA and the OPC UA server exchange data using the protocol defined in this thesis

3.3.1. Initialization of OPC UA server address space

The OPC UA address space is created using the XML files conforming to the OPC UA Nodeset 2.0 XML specification. Each PA must initialize its part of the OPC UA address space. This is done by placing the XML files to a certain directory in Linux file system. The XML files can either be supplied with a PA or generated by a PA. Each PA also has a unique namespace name which is defined in the XML file of that PA and is used to identify the PA when communicating with the OPC UA server.

The nodes can be loaded to the OPC UA server's address space by using any OPC UA NodeSet 2.0 compliant XML file, however for communication interface between PA and the OPC UA server to work, there are few additional requirement for the nodes defined in the XML file:

- Node IDs defined in the XML file must have numeric identifiers.
- Methods should not have any input or output arguments (other nodes may be used to pass arguments as a workaround if necessary).

3.3.2. Inter-process communication interface

The communication between a PA and the OPC UA server is implemented utilizing UNIX Domain sockets [43]. A sequential packet socket type and a client-server model is used for the OPC-PA interface. The OPC UA server also functions as a server in the OPC UA interface and has a constant socket name. The connection between a PA and the OPC UA server is established when the PA connects to the OPC UA server socket.

Use of sockets allows to easily adapt this protocol for communication with a remote server by changing the socket type to the internet domain sockets (e.g. TCP/IP).

3.3.3. Namespace index request

Each PA has its namespace name which is defined in the XML file. However, the more efficient and preferred way of addressing nodes is through the namespace index that is assigned by the OPC UA server when it loads the address space. After connecting to the OPC UA server, the PA needs to obtain the namespace index of its namespace for further communications. It is accomplished by sending a namespace index request message, containing the namespace name of the PA, to the OPC UA server. The server then responds with the namespace index assigned to that namespace.

The structures of a namespace request packet (sent from a PA) and a response packet (sent from the OPC UA server) are shown in Table 3.1 and Table 3.2, accordingly.

Table 3.1: Namespace index request packet structure

Filed name	**Size**	**Description**
Packet length	2 bytes	The number of bytes in the packet excluding this field.
Packet type	2 bytes	Indicates a packet format and the function that should be performed. 4 – Namespace index request
Namespace name	1-256 byte	The namespace name of the PA (same as defined in the XML file)

Table 3.2: Namespace index response packet structure

Filed name	**Size**	**Description**
Packet length	2 bytes	The number of bytes in the packet excluding this field.
Packet type	2 bytes	Indicates a packet format and the function that should be performed. 4 – Namespace index response
NSID	2 bytes	The unique OPC UA namespace identifier assigned to the protocol adapter. Used to identify the PA in the further communication.
StatusCode	2 bytes	Indicates, whether the operation was successful

3.3.4. Data update operation

The update operation is used to inform the receiving party about errors, events, or changes of parameters. The message used for this operation shall be later referred as *Update Message*. The *Update Message* contains information required to identify a certain parameter or action and data, which shall be used to update the value of a parameter or to perform an action.

The packet structure of *Update Message* is described in the Table 3.3.

Table 3.3: The structure of the *Update Message*

Filed name	**Size**	**Description**
Packet length	2 bytes	The number of bytes in the packet excluding this field.
Packet type	2 bytes	0 – for *Update Message*. Indicates a packet format and the function that should be performed.
NSID	2 bytes	The unique OPC UA namespace identifier that was assigned to the PA. Used to identify the PA.
StatusCode	2 bytes	A numeric value used to identify the state of parameter or action.
NodeID	4 bytes	A numeric node identifier. It is used to specify a particular parameter or action.
TimeStamp	8 bytes	Defines a Gregorian calendar date. The date and time is stored in UTC time. The date and time value is based on 100 nanosecond ticks since January 1, 1601 00:00:00 (FILETIME).
ValueData	Determined from the packet length	A value of the parameter.

3.3.5. OPC UA address space modification

Normally, the OPC-UA address space is loaded from the OPC UA Nodeset 2.0 XML files on the start-up. If it is necessary to update the address space during the runtime, a PA should create a new XML file and signal the OPC UA server to update the address space. The OPC UA server shall respond with the status indicating whether the address space was modified successfully.

The request to modify the address space (from a PA) specifies the filename of the OPC UA Nodeset 2.0 XML file which shall be used to update address space. Nodes defined in this file may either replace all previous nodes in the namespace of the PA (P*acket type* 2 – Reload namespace) or add new nodes to the namespace without discarding existing nodes (P*acket type* 3 – Append namespace). The structures of request and response packets are described in Table 3.4 and Table 3.5, respectively.

Table 3.4: Address space modification request

Filed name	**Size**	**Description**
Packet length	2 bytes	The number of bytes in the packet after this field.
Packet type	2 bytes	Indicates a packet format and the function that should be performed. 2 – for *Reload namespace operation.* 3 – for *Append namespace* operation.
NSID	2 bytes	The unique OPC UA namespace identifier that was assigned to the PA. Used to identify the PA.
Filename	1-256 byte	The OPC UA Nodeset 2.0 XML file to load nodes from.

Table 3.5: Address space modification response

Filed name	**Size**	**Description**
Packet length	2 bytes	The number of bytes in the packet after this field.
Packet type	2 bytes	Indicates a packet format and the function that should be performed. 2 – for *Reload namespace operation.* 3 – for *Append namespace* operation.
NSID	2 bytes	The unique OPC UA namespace identifier that was assigned to the PA. Used to identify the PA.
StatusCode	2 bytes	Indicates whether the address space was updated successfully

3.3.6. Implementation in protocol adapters

A function library was developed to facilitate a PA communication with the OPC UA server. The library defines OPC-PA interface management functions, data exchange functions and utility functions (e.g. to change the time format). All functions return a status code indicating, whether it succeeded or failed. The C header file of the library is presented in Appendix 1.

The most important function in the library is the interface initialization function. This function connects to the OPC UA server via UNIX domain sockets and creates a new thread for receiving messages from the OPC UA server. It allows to specify a function (call-back function) which will be called when an *Update Message* is received from the OPC UA server. The initialization function takes the namespace name of the PA and a call-back function as parameters. It is also allowed to specify the call-back function as NULL, so no action will be performed, when an *Update message* is received. The algorithm of initialization function is presented in the UML activity diagram in Fig. 3.5. At first, the function attempts to connect to the OPC UA server and, once connected, it sends a namespace index request. If a positive response is received, the PA memorizes the namespace index and starts waiting for messages from the OPC UA server by creating a new thread for this task. If the function fails at any point, it exits returning an appropriate status code.

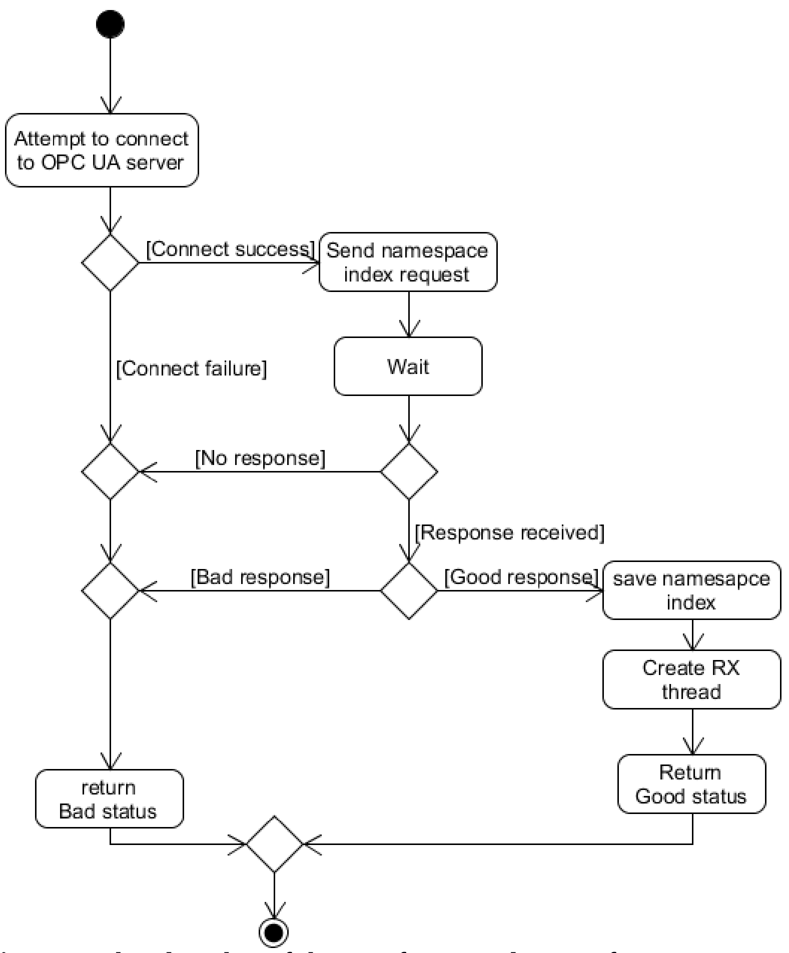


Fig. 3.5: The algorithm of the interface initialization function in a PA

The message reception algorithm running in the thread that is created by the initialization function is depicted in the UML activity diagram in Fig. 3.6. The thread continuously waits for packets from the OPC UA server. When a packet is received, its namespace and type are checked. If the namespace in the packet does not match the namespace of the PA, a packet with a status code indicating an invalid node ID is send to the OPC UA server. In case of a valid namespace, the call-back function is executed for the packets of the *Update Message* type, and for the packets that are namespace modification responses - a variable indicating the status of the namespace modification attempt is set. The implementation of call-back function is application specific. Generally, it should check the node ID to determine which node the request came from and then decide what actions must be performed. The usual actions are: send data to an external device, set the internal parameter of the PA (e.g. a sampling rate) or perform some specific action (if the node ID indicates a method).

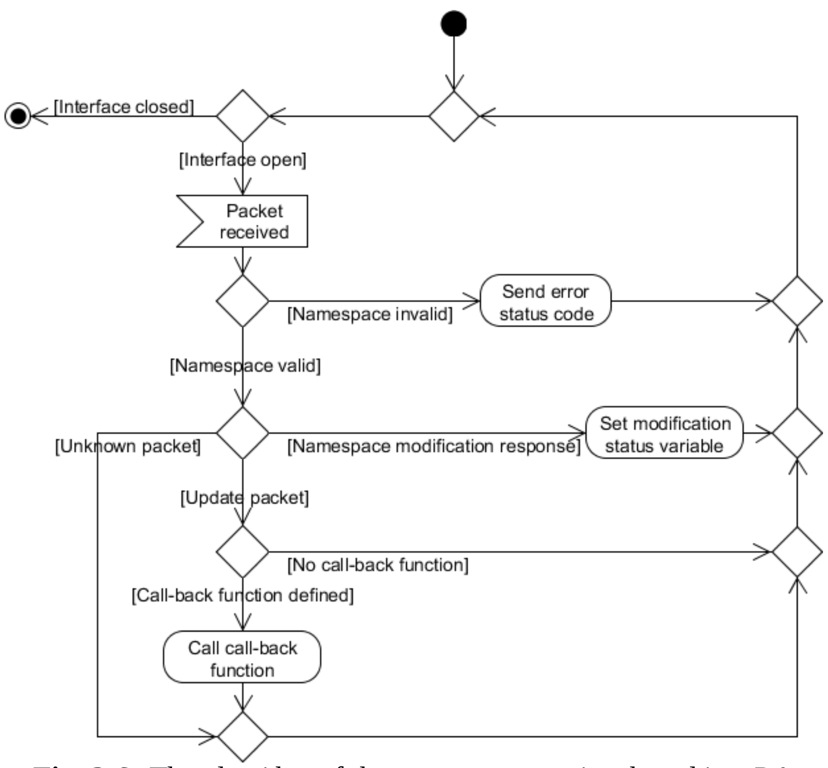


Fig. 3.6: The algorithm of the message reception thread in a PA

The data exchange functions defined in the library assists in creating namespace modification or the *Update Message* type packets and sending them to the OPC UA server. The full algorithm of those functions is shown in the UML activity diagram in Fig. 3.7. Note, that the full algorithm only applies to the namespace modification operations, as there is no response defined for messages of the *Update Message* type.

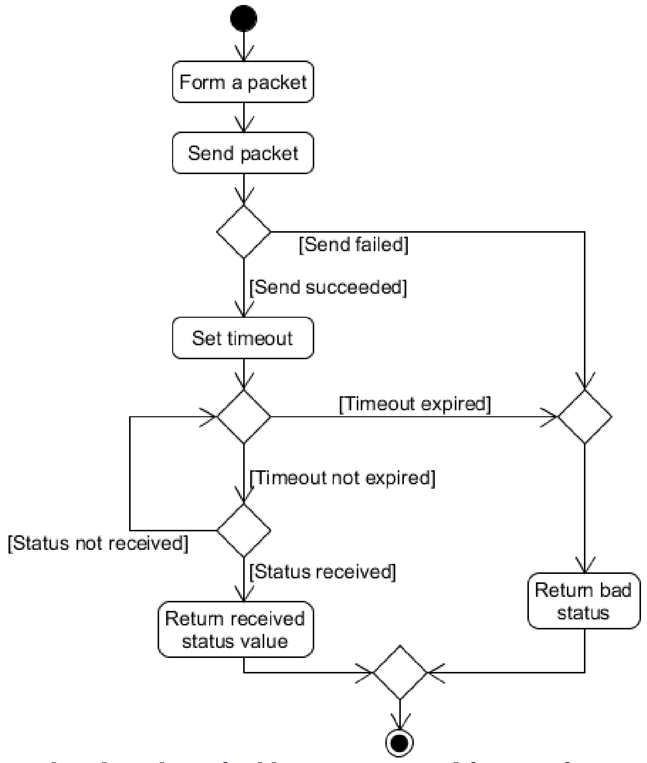


Fig. 3.7: The algorithm of address space modification function in PA

The function forms a packet, filling its field based on function type and input arguments (node ID, data value, filename, etc.), and attempts to send it to the OPC UA server. Functions used for node value modification stop at this point returning status indicating whether sending of the packet was successful or not (as there is no response to wait for). The namespace modification functions stop here only if sending of the packed fails, otherwise they wait for the response from the OPC UA server. Because the actual reception of messages is handled by the other thread, this function is only checking the status variable to see, whether response has been received. If response is not received within a 100 ms time frame, the function returns with a bad status, otherwise it returns the status received from the OPC UA server.

3.3.7. Implementation in the OPC UA server

The OPC UA server also works as a server for the OPC-PA interface. All operation related to accepting connections from PAs and handling their requests run in the separate thread which is created by the OPC-PA interface initialization function.

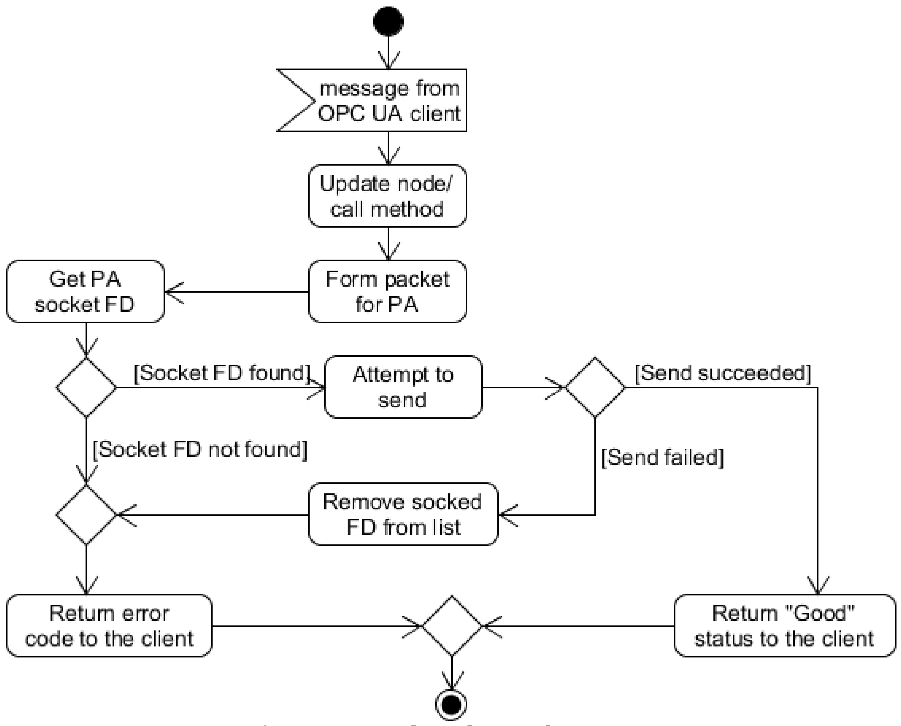


Fig. 3.8: Sending the packet to PA

The OPC UA server sends an *Update Message* to a PA when either the value of a variable node is changed or a method is activated by an OPC UA client. It was implemented by creating new custom types of method and variable nodes and then overriding default *"node write"* and *"call method"* functions. The algorithm of write operations is shown in Fig. 3.8. Basically, after performing the standard node update in the address space, it forms and sends a packet to the PA in order to inform about this change. It uses the information about the connected PA adapters, their socket file descriptors (FD), and namespace indexes to determine, which socked FD must be used to reach the required PA.

The algorithm of the thread responsible for accepting PA connections and handling their requests is shown in the UML activity diagram in Fig. 3.9.

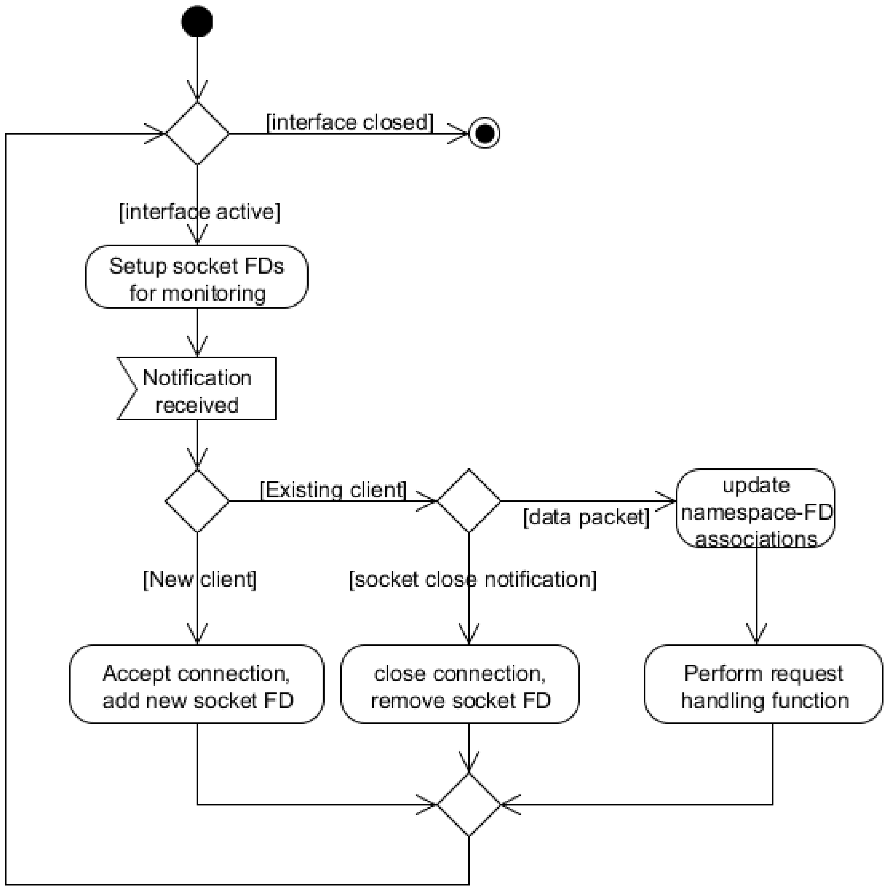

[interface closed]
[interface active]
Setup socket FDs for monitoring
Notification received
[Existing client]
[data packet]
update namespace-FD associations
[New client]
[socket close notification]
Accept connection, add new socket FD
close connection, remove socket FD
Perform request handling function

Fig. 3.9: The algorithm of the thread responsible for handling PA requests

When the thread is created by the initialization function, it enters a loop and starts monitoring the list of socket FDs. At first there is only one socket FD which is used to accept PA connections, but the list grows as new PA connections are accepted, (connection with each PA has an individual socket FD). Once the monitoring of sockets is set-up, the thread starts to wait for activity on these sockets. There may be three types of activities from clients. The first is a new connection request, which is handled by establishing a connection with a client and adding its socket FD to the list. The second is connection close notification from the existing client, which is handled by closing connection with the client and removing its socket FD from the list. The third one is a data packet from the client. When such packet is received, at first the namespace index of the packet is used to update namespace and socket FD associations. These associations are used by the OPC UA server to determine which socket FD it must use to reach the PA managing the particular namespace. After that, the handling of the request is performed.

The algorithm of the request handling operation is detailed in the UML activity diagram in Fig. 3.10.

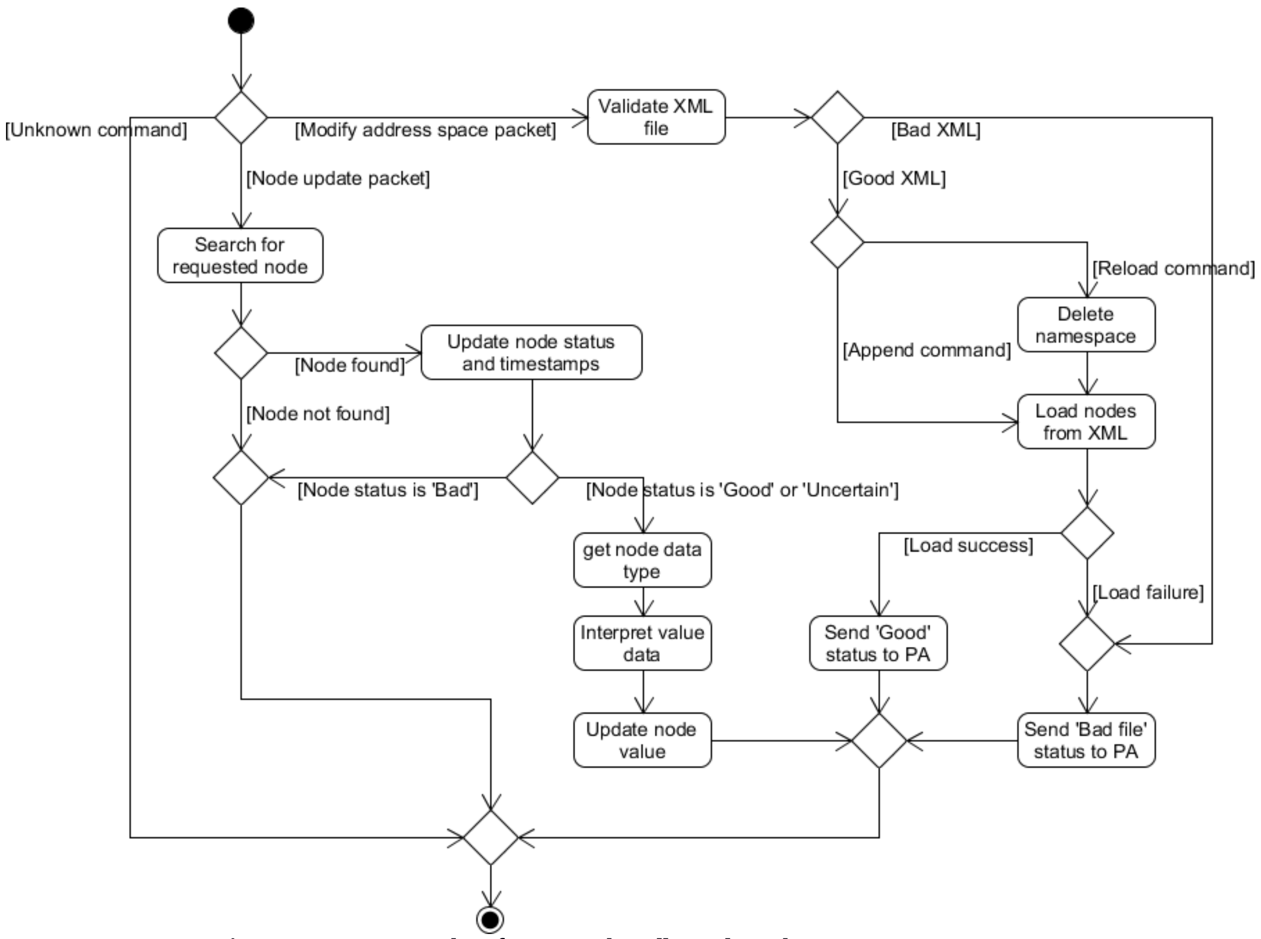


Fig. 3.10: Data packet from PA handling algorithm in OPC UA server

The request received from the PA may be of two types: the address space modification or the node value modification. If a received packet is node value modification, the OPC server searches the address space for the node specified in the packet. Once the node is found, its data type is obtained to determine, how to interpret the data received. The value of the node is updated only if the packet contains good or uncertain status code, otherwise only timestamp and status of the node are updated, thus leaving the last usable value of the node. For the address space modification packets, the validation of the specified XML file is the first action which is performed (file must exist, the namespace defined in the file must match namespace managed by the PA sending the request, etc.). After that, the OPC server proceeds to loading nodes from the XML file, removing the old nodes if necessary. The PA is informed about the status of namespace modification by sending the response with an appropriate status code.

3.4. OPC UA server conditional access control

Our implementation of user conditional access in OPC UA server is based on X.509 certificates. The identity of a user is determined by the common name (CN) field in the certificate. OPC UA allows to assign different conditional access rights for each node, however it was decided to implement conditional access on the namespace basis, as assigning access right to each node would be too complicated from the gateway owner's point of view. The namespaces are directly related to PAs, which in turn are related to a particular device or the type of devices. This makes the conditional access mechanism based on namespaces very convenient, as it allows the owner to configure the user access rights for particular devices attached to the gateway.

The assignment of conditional access rights in the OPC UA server is managed as a two dimensional array, where users and PA can be considered as rows and columns, respectively. By default, the table is filled with zeros, which gives no access rights to any of the users. The table is modified by loading access rights from a file. The file consists of multiple lines. Each line is made of PA name, user name, and conditional access value separated by spaces. It is interpreted as: a particular PA can be accessed by the specified user with the rights indicated by the conditional access value. This file is modified by assigning conditional access rights through the configuration web page running on the IoT gateway.

4. PROOF OF CONCEPT

The software architecture for data aggregation into the OPC UA server was proposed and detailed in the previous chapter. This chapter seeks to demonstrate the feasibility of the architecture for various data acquisition purposes. For this task four low complexity PA were developed. They implement data acquisition from different data sources, which are listed below:

- Data acquisition from a weather station (WS) via wireless link
- Reading machinery data, which comes in J1939 format via CAN bus
- Visual observation by periodically taking a photo
- Displaying internal status of the gateway itself

4.1. Data acquisition from a weather station

A-Lab AWS-1 weather station was connected to the IoT gateway via 169 MHz wireless link to demonstrate data acquisition from wireless sensor nodes. The WS provides measurements every 10 minutes. The PA that was developed listens for packets arriving from the wireless link module and places the received WS data to the OPC UA address space. The PA also allows the owner to specify location (longitude and latitude) of the WS, as the weather station does not provide this information. The location information is saved in the file system and can be viewed by other OPC UA users (if they have required permissions). The algorithm of WS PA is presented in Fig. 4.1.

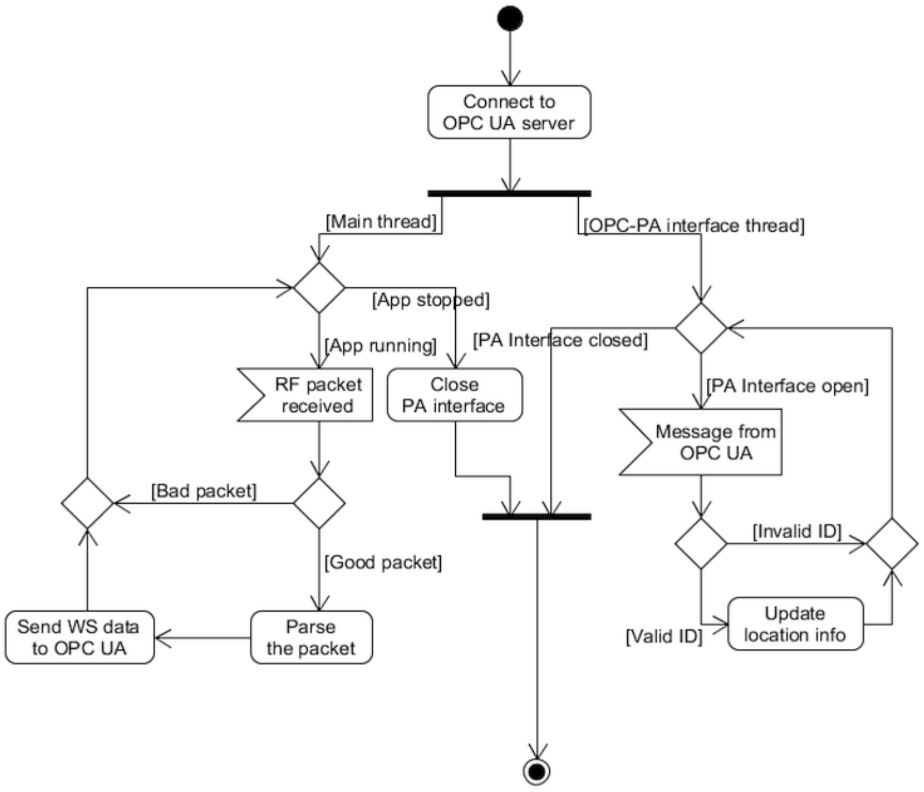
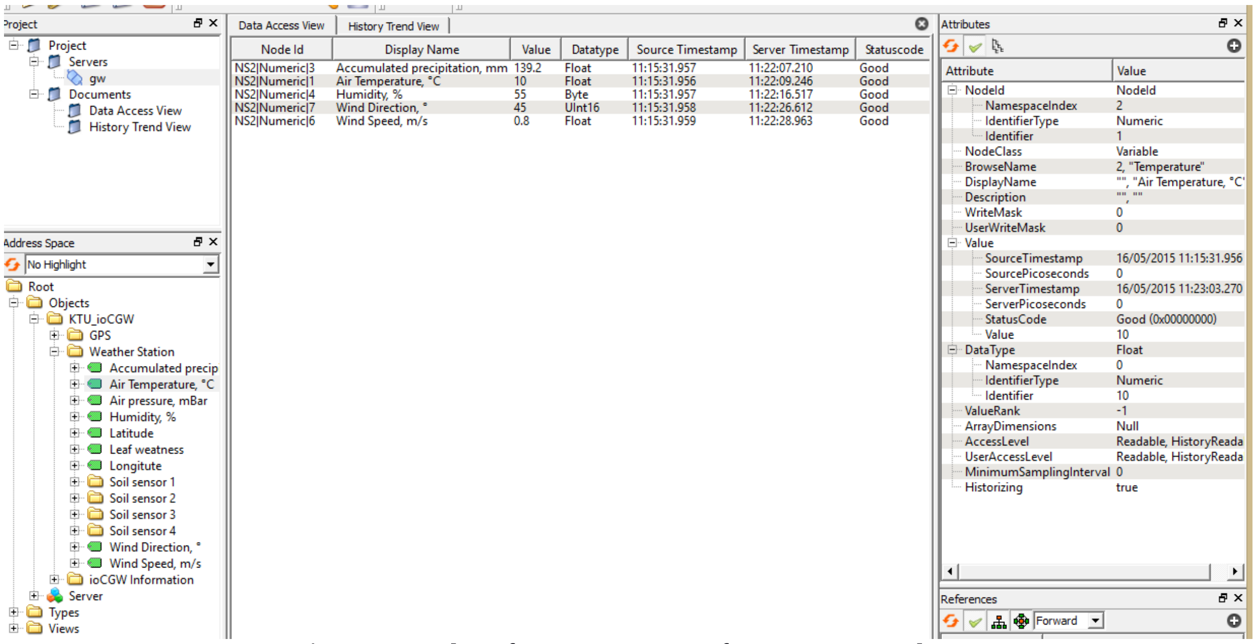


Fig. 4.1: The algorithm of WS PA

The PA connects to the OPC UA server via OPC-PA interface. At this point, a new thread is created for receiving messages from the OPC UA server. This PA uses the OPC-PA interface thread for the acquiring the location information (which is provided by the owner) from the OPC UA server. The location information is saved to a file and is later sent to the OPC UA server together with weather data.

The main thread of the PA is responsible for receiving packets from the radio module. When a packet is received, it is checked for length and data errors. After the check is successfully performed, the packet is parsed and the weather data, extracted from the packet, is sent to the OPC UA server, where it becomes available for clients in the address space.

The WS part of OPC UA address space was loaded from the XML file (see Appendix 2). We were able to browse the address space and monitor the WS readings using the standard OPC UA client. The address space together with some parameter values as displayed by UAExpert OPC UA client are shown in Fig. 4.2.



```
Project
  Project
    Servers
      gw
    Documents
      Data Access View
      History Trend View

Address Space
No Highlight
  Root
    Objects
      KTU_ioCGW
        GPS
        Weather Station
          Accumulated precip
          Air Temperature, °C
          Air pressure, mBar
          Humidity, %
          Latitude
          Leaf weatness
          Longitude
          Soil sensor 1
          Soil sensor 2
          Soil sensor 3
          Soil sensor 4
          Wind Direction, °
          Wind Speed, m/s
        ioCGW Information
    Server
  Types
  Views
```

Data Access View | History Trend View

Node Id	Display Name	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
NS2|Numeric|3	Accumulated precipitation, mm	139.2	Float	11:15:31.957	11:22:07.210	Good
NS2|Numeric|1	Air Temperature, °C	10	Float	11:15:31.956	11:22:09.246	Good
NS2|Numeric|4	Humidity, %	55	Byte	11:15:31.957	11:22:16.517	Good
NS2|Numeric|7	Wind Direction, °	45	UInt16	11:15:31.958	11:22:26.612	Good
NS2|Numeric|6	Wind Speed, m/s	0.8	Float	11:15:31.959	11:22:28.963	Good

Attributes

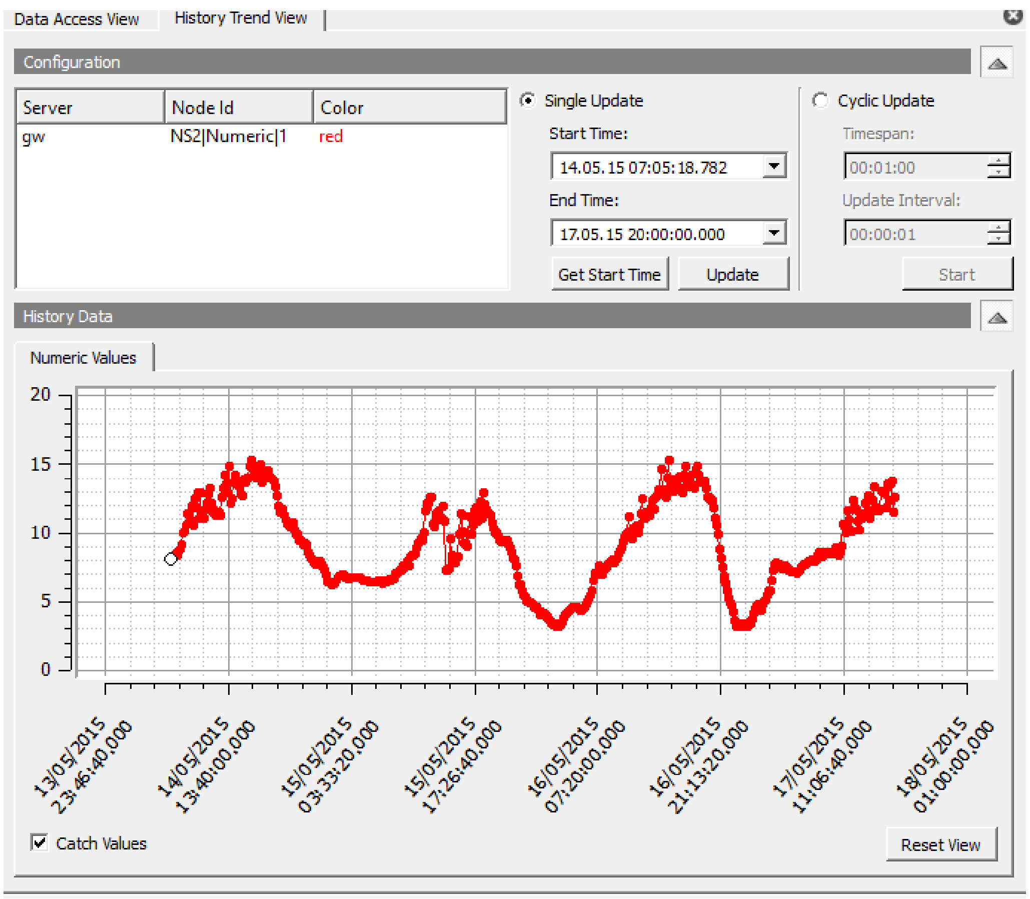
Attribute	Value
NodeId	NodeId
NamespaceIndex	2
IdentifierType	Numeric
Identifier	1
NodeClass	Variable
BrowseName	2, "Temperature"
DisplayName	"", "Air Temperature, °C'
Description	"", ""
WriteMask	0
UserWriteMask	0
Value	
SourceTimestamp	16/05/2015 11:15:31.956
SourcePicoseconds	0
ServerTimestamp	16/05/2015 11:23:03.270
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	10
DataType	Float
NamespaceIndex	0
IdentifierType	Numeric
Identifier	10
ValueRank	-1
ArrayDimensions	Null
AccessLevel	Readable, HistoryReada
UserAccessLevel	Readable, HistoryReada
MinimumSamplingInterval	0
Historizing	true

References

Forward

Fig. 4.2: Nodes of WS PA as seen from OPC UA client

The historizing flag was also set in the XML file for some of the nodes. The implementation in the OPC UA server checks this flag to determine if the value needs to be save (as discussed before this is not done automatically by the server as SDK and was implemented manually). The historical data feature was tested to work. The historical data of air temperature obtained from the IoT gateway is presented in Fig. 4.3.



Server	Node Id	Color
gw	NS2|Numeric|1	red

[✓ Single Update] Cyclic Update

Start Time: 14.05.15 07:05:18.782

End Time: 17.05.15 20:00:00.000

Timespan: 00:01:00

Update Interval: 00:00:01

☑ Catch Values

Fig. 4.3: Historical data viewed using UAExpert

4.2. Reading data from machinery

The IoT gateway uses socketCAN [44] drivers and networking stack for CAN communications. Data acquisition through CAN interface was tested by connecting the gateway to *Wabco* Electronic Braking System (EBS). The EBS communicates according to ISO 11992 standard, which is based on J1939 standard and has the same message format. This setup allowed us to test transmission and reception of messages using CAN bus, although in real applications it would not be possible to control something as critical as brakes through tractor-implement bus. The overall structure of CAN setup software and hardware layers is shown Fig. 4.4:

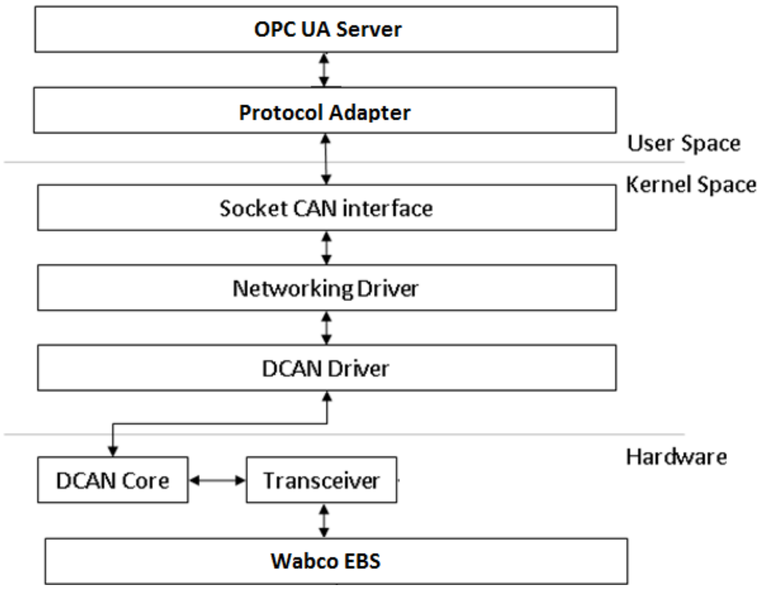
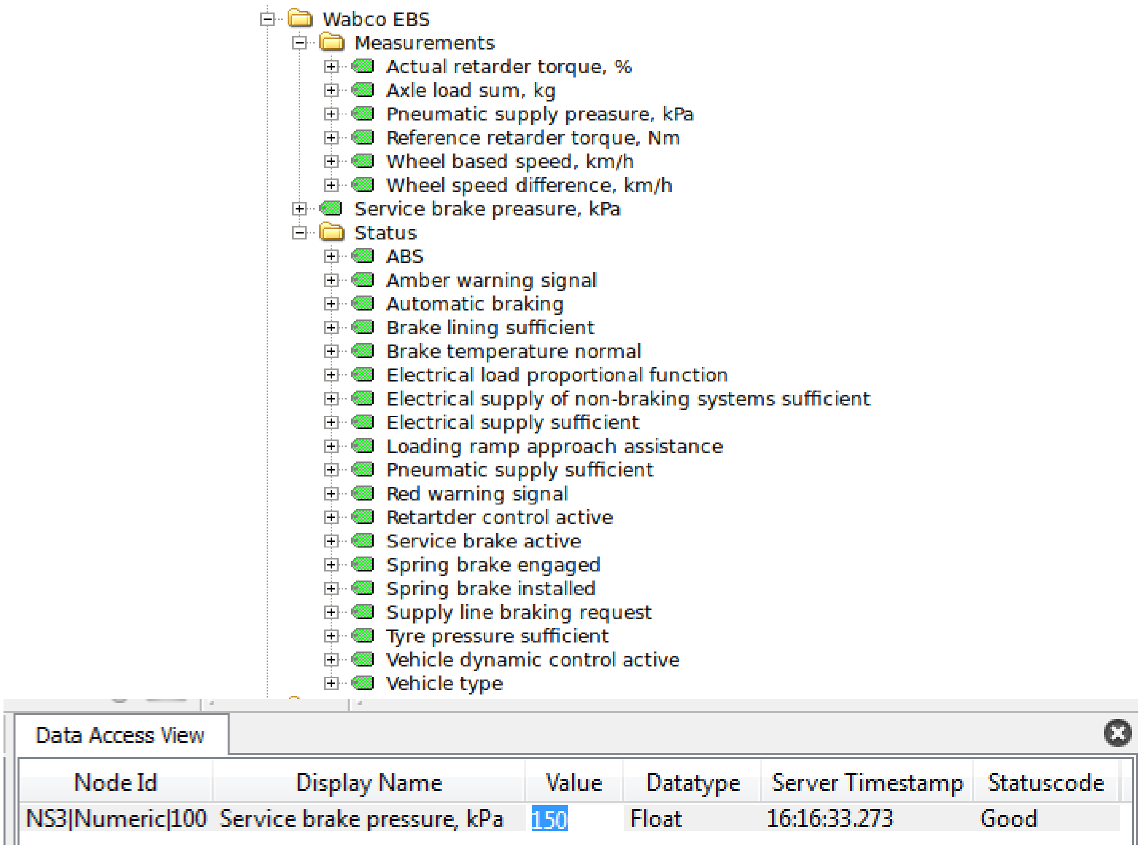


Fig. 4.4: Structure of CAN implementation in the gateway

The PA obtains parameters and status values by reading CAN messages received from the EBS. The ability to send a request to set specific brake pressure to the EBS was also implemented. The nodes created by this PA are shown in Fig. 4.5



```
Wabco EBS
  Measurements
    Actual retarder torque, %
    Axle load sum, kg
    Pneumatic supply preasure, kPa
    Reference retarder torque, Nm
    Wheel based speed, km/h
    Wheel speed difference, km/h
  Service brake preasure, kPa
  Status
    ABS
    Amber warning signal
    Automatic braking
    Brake lining sufficient
    Brake temperature normal
    Electrical load proportional function
    Electrical supply of non-braking systems sufficient
    Electrical supply sufficient
    Loading ramp approach assistance
    Pneumatic supply sufficient
    Red warning signal
    Retartder control active
    Service brake active
    Spring brake engaged
    Spring brake installed
    Supply line braking request
    Tyre pressure sufficient
    Vehicle dynamic control active
    Vehicle type
```

Data Access View

Node Id	Display Name	Value	Datatype	Server Timestamp	Statuscode
NS3|Numeric|100	Service brake pressure, kPa	150	Float	16:16:33.273	Good

Fig. 4.5: Nodes of gateway CAN PA as seen from OPC UA client

The parameter and status values provided by this PA feature a relatively high update rate as some messages from EBS arrive at 10 ms interval and others at 100 ms interval. In addition to this, a message containing brake pressure demand value must be sent to the EBS at 10 ms interval.

The algorithm of CAN PA is presented in Fig. 4.6 (the source code of this PA can be found in Appendix 3). At first, the protocol adapter connects to OPC UA server via the OPC-PA interface. At this point, a thread for handling messages received from OPC UA server is created (it is done by the initialization function of the OPC-PA protocol). The main thread proceeds by connecting to socketCAN broadcast manager (BCM) and setting up receive filters for CAN messages. Once the receive filter setup is completed, the main thread starts waiting for messages from the BCM. When a message from BCM is received, the PA checks CAN ID of the message to determine, what parameters are in the data field of the CAN frame (it is done according to ISO 11992 standard). After this, the parameter values are extracted from data field and sent to the OPC UA server.

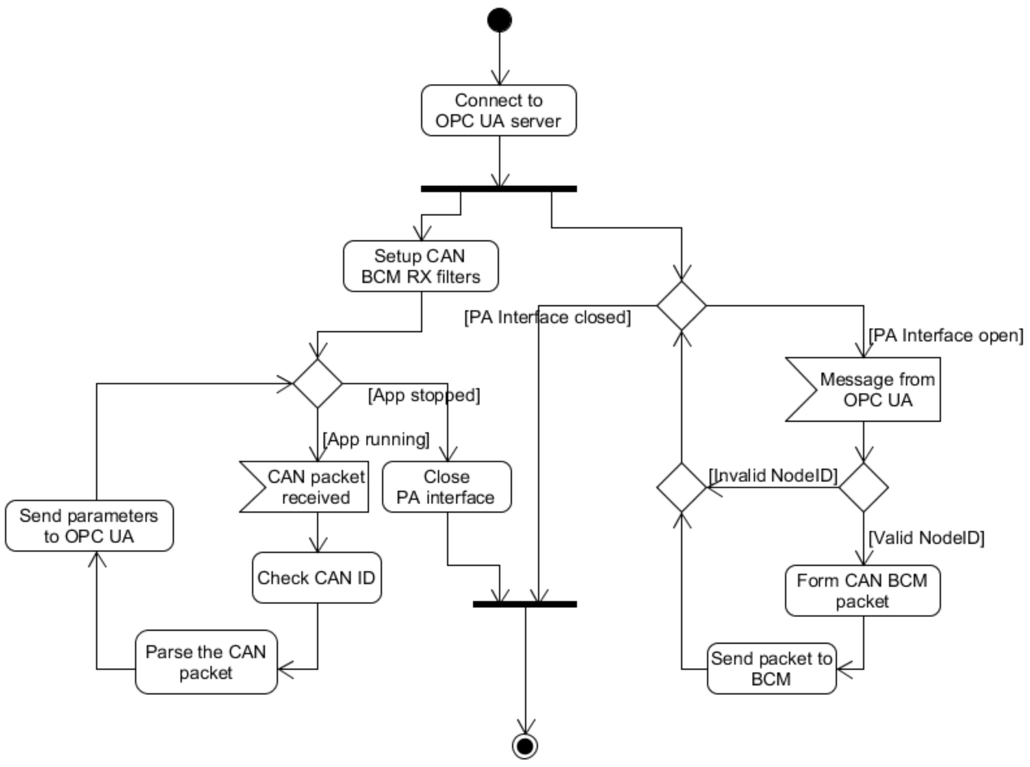


Fig. 4.6: The algorithm of CAN PA

The PA-OPC interface thread waits for messages from the OPC UA server arriving via the PA interface. Once a message is received, it is checked if the NodeID received is 100, which was defined as "Brake pressure node" in XML file of this PA. If the NodeID is not 100, it sends a message to the OPC UA server with the status code indicating invalid node ID. Otherwise, the PA takes data from the message (which contains brake pressure value) and forms a packet for CAN BCM. This packet is then send to BCM to start transmitting CAN messages with the updated brake pressure value.

4.3. Transfer of images

The observation using camera was tested in the gateway. For this purpose Linux kernel had to be configured to include the required USB video input device drivers, also FFmpeg tool was installed. The PA is used to take a photo every 6 seconds and place it to OPC UA server address space via PA-OPC interface. The UML sequence diagram of the procedure is presented in Fig. 4.7.

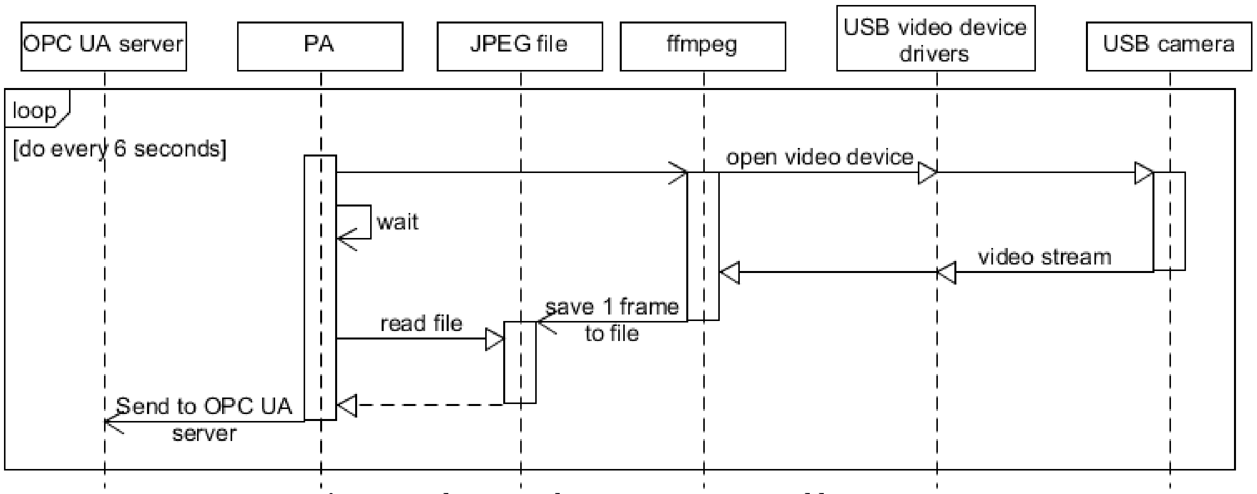


Fig. 4.7: Placing a photo into OPC UA address space

Protocol adapter tracks the time and executes FFmpeg every 6 seconds. FFmpeg opens a video capture device, takes one frame, converts it to JPEG format and stores it to a file. The file resides on a temporary file system in the RAM. Once FFmpeg finishes, PA reads the file and transfers its contents to OPC UA Server using the OPC-PA protocol. The image node in OPC UA server contains binary data of JPEG image, which was successfully viewed using UAExpert's Image viewer tool, as shown in Fig. 4.8.

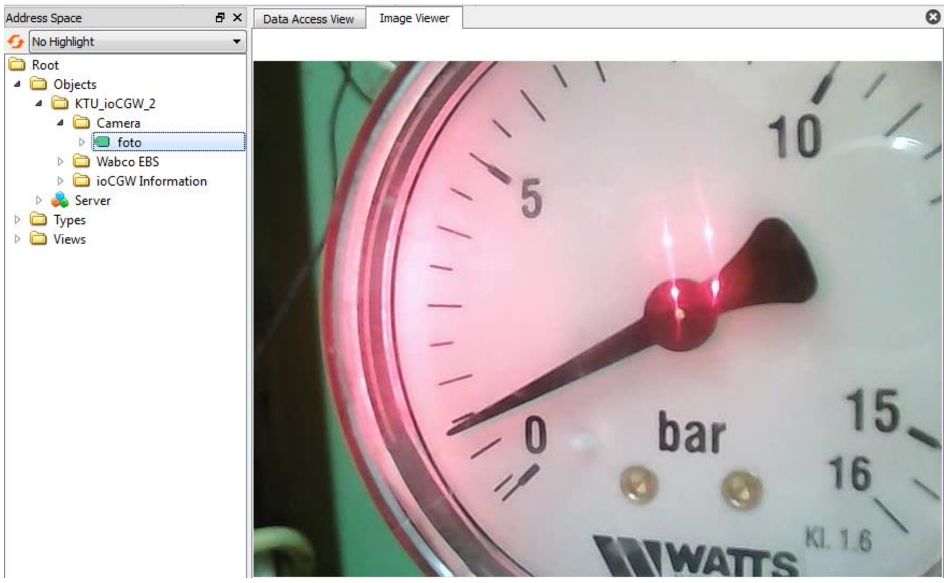


Fig. 4.8: A photo in OPC UA server viewed using UAExpert

4.4. Displaying status of the gateway

The gateway status PA provides information about gateway version, ID, CPU utilization, and status of various connections. It also allows to turn off or on some of the interfaces from an OPC UA client, given the user has sufficient conditional access rights. This PA utilizes various Linux system files and services to attain its functionality. The nodes managed by this protocol adapter are presented in Fig. 4.9

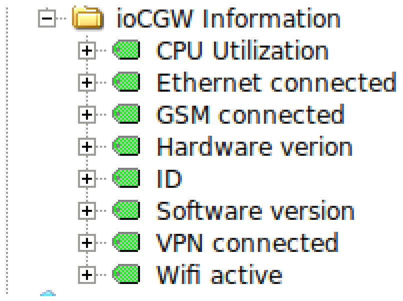

ioCGW Information
CPU Utilization
Ethernet connected
GSM connected
Hardware verion
ID
Software version
VPN connected
Wifi active

Fig. 4.9: Nodes of gateway status PA as seen from OPC UA client

4.5. Stability of the system

The stability of the systems has proved to be good. The setup utilizing the four PAs was running for over the month without any crashes. It was tested that PA can be freely disconnected and reconnected to the OPC UA server. The restart of OPC UA server also didn't cause any of the PA to fail, as they managed to successfully reconnect.

We were able to access data via various interfaces including local Ethernet, VPN and Wi-Fi. The VPN connection allowed us to remotely access the gateway over the internet.

5. EXPERIMENTAL TESTING

This section describes the experiments that were carried out during the course of this thesis. The main purpose of these experiments is to evaluate the suitability of the OPC UA based IoT gateway for agricultural applications.

5.1. Performance of different connectivity options

The IoT gateway has several connectivity options, which may be used to access the OPC UA server. In this experiment we seek to evaluate the performance of those connectivity options. The purpose of this experiment is to get an idea of connection capabilities before testing data transfer performance of the OPC UA server. The four main connectivity options for accessing the IoT gateway together with condition used for testing are listed in the Table 5.1.

Table 5.1: The connectivity options used in experiments

Connection option	**Description**
Ethernet	The gateway and the PC with an OPC UA client are connected to ASUS RT-N53 router and resides on the same local network.
Wi-Fi	The PC is connected directly to the AP hosted by the gateway. The AP is configured to work in IEEE 802.11g mode and is secured using Wi-Fi Protected Access II (WPA2) security protocol.
VPN	The gateway and the PC are connected to the internet via wired Ethernet connections. They have OpenVPN clients running on them and are in the same VPN. The cryptographic functions used by OpenVPN are left at defaults (BlowFish CBC 128, SHA1, RSA 1024).
GSM	Same as the VPN connection option, but in this case the IoT gateway is connected to the internet using GSM modem instead of wired connection.

5.1.1. Test method

The performance of each connection option, listed in the Table 5.1, was evaluated by measuring connection throughput and latency. The throughput was investigated by copying a 50 MiB file from the Linux PC to the gateway using network file system (NFS). The PC was set up as NFS server, while the gateway was configured as NFS client. The file was filed with random data and was copied to temporary file system in the gateway RAM. The latency of connection was tested by sending ping request from the PC to the gateway. For each connection option, the ping request was sent 100 times and the average latency was calculated.

5.1.2. Test results

The measured latency, file transfer time, and calculated throughput of each connection option are presented in the Table 5.2.

Table 5.2: The performance of various connectivity options

Connection type	File copy time, s	Throughput, Mbit/s	Latency (ping), ms
Ethernet	4.765	88.0	0.396
VPN	13.89	30.2	2.98
Wi-Fi	24.63	17.0	2.42
GSM	411.22	1.02	50.3

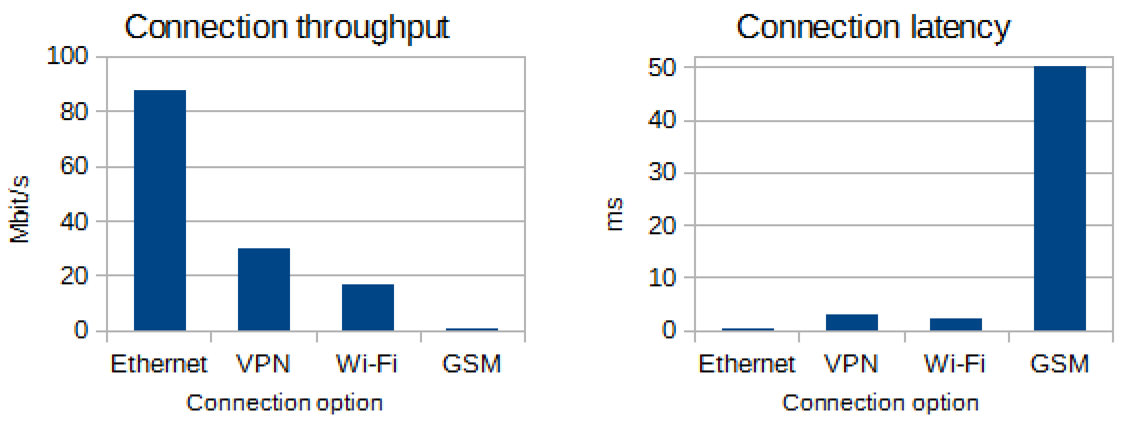


Fig. 5.1: Comparison of connection throughputs (left) and latencies (right)

5.2. Data acquisition rate from OPC UA server

The data acquisition rate experiment seeks to find out, how fast the parameter values can be obtained from the OPC UA server. The tests were performed with several OPC UA request sizes to investigate the benefits of combining multiple readings into a single message (e.g. by subscription services). In general, small request should be limited by the connection latency and increasing the packet size should make it more dependent on the connection throughput.

5.2.1. Test method

The performance of the OPC UA server running on the IoT gateway was evaluated using the performance plugin of the UAExpert OPC UA client from Unified Automation. The plugin measures the delay between the request and response for OPC UA service calls. In this experiment, Attribute-Read and Attribute-Write request calls were used. The performance using subscription notifications cannot be measured directly without developing a custom client, however the results should be very similar.

The tests were performed using four different connectivity options, which are described in the Table 5.1. Each connection option was tested using three different request sizes (1, 10 and 30 nodes in a single request). In addition to this, to investigate impact of using multiple clients, read tests were also carried out using two clients instead of one. In the latter case, both clients were sending the same type of requests using the same type of connection option and the performance was measured in one of them.

In every test case the client was configured to send 1000 request and all nodes used for read/write operations were of **double** data type. The OPC server channel security level was set to “none”. The approximate CPU usage was estimated by observing values displayed by Linux *top* utility.

5.2.2. Test results

The test results, when a single OPC UA client is loading the server, are presented in the Table 5.3. It includes the average time per service call measured by the client, calculated average requests per second, and the approximate gateway CPU utilization are during the test.

Table 5.3: The performance of data acquisition from OPC UA server

Connection type	Node count	Average time per call, ms		Average requests per second		CPU utilization, %
		Read	Write	Read	Write	
Ethernet	1	1.11	1.09	900.9	917.4	~60
	10	1.82	1.97	549.5	507.6	~70
	30	3.42	3.78	292.4	264.6	~80
VPN	1	3.69	3.63	271.0	275.5	~40
	10	4.74	4.55	211.0	219.8	~50
	30	6.63	6.45	150.8	155.0	~60
Wi-Fi	1	3.16	3.2	316.5	312.5	~50
	10	3.8	3.92	263.2	255.1	~60
	30	5.77	5.85	173.3	170.9	~70
GSM	1	56.35	54.4	17.8	18.4	~5-10
	10	57.83	56.36	17.3	17.7	~5-10
	30	69.38	66.88	14.4	15.0	~5-10

The graphical comparison of request rates, when a single OPC UA client is polling the server, is presented in the Fig. 5.2.

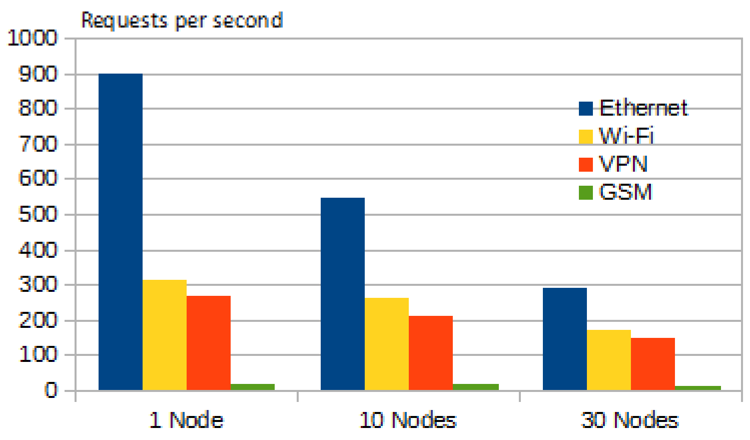


Fig. 5.2: The request throughput of different connectivity options

The results clearly show the benefit of combining several nodes into a single message, especially when dealing with the connections that have high latency. For GSM option, going from one node per request to 30 nodes per request, decreased request throughput by less than 20%.

Overall, the Ethernet connection achieves the best results. The results of VPN and Wi-Fi connections are similar. Both of these connection options use data encryption, which puts additional load on the gateway CPU. Because of this, the gateway CPU utilization is not that much lower than in the local Ethernet connection test case, even though the requests rate is several times lower. Meanwhile, the GSM option is highly limited by slow connection and is not able to put noticeable load on the gateway CPU. Nevertheless, it manages to satisfy the requirements set in 1.4.2.

The individual requests times of some tests are graphically shown in figures below:

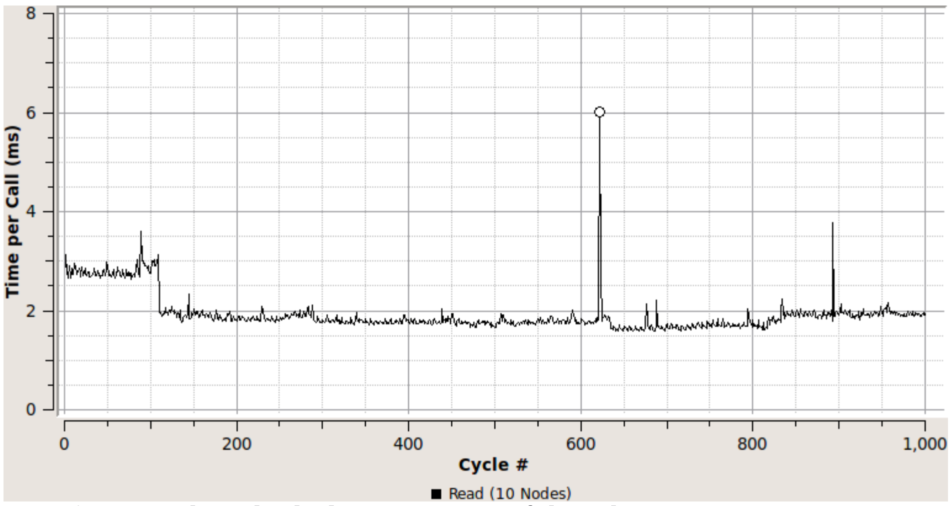


Fig. 5.3: The individual request times of the Ethernet connection option

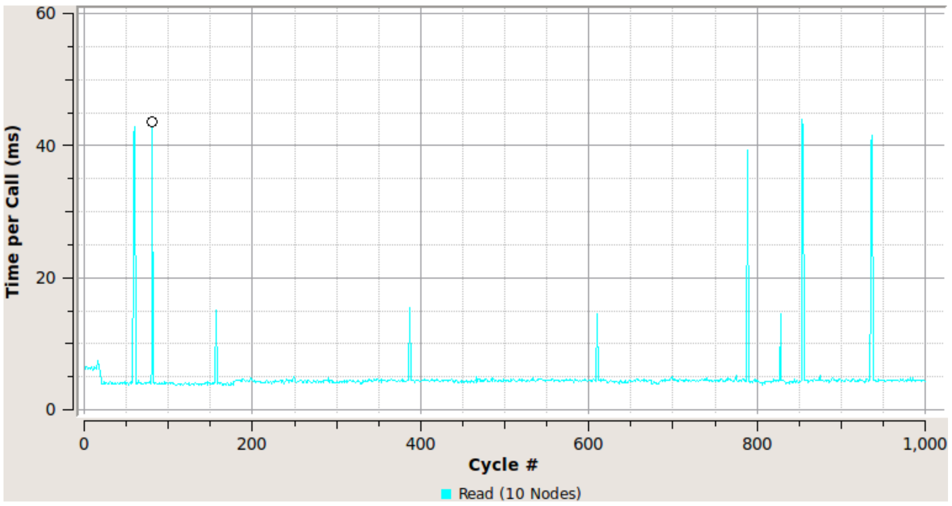


Fig. 5.4: The individual request times of the VPN connection option

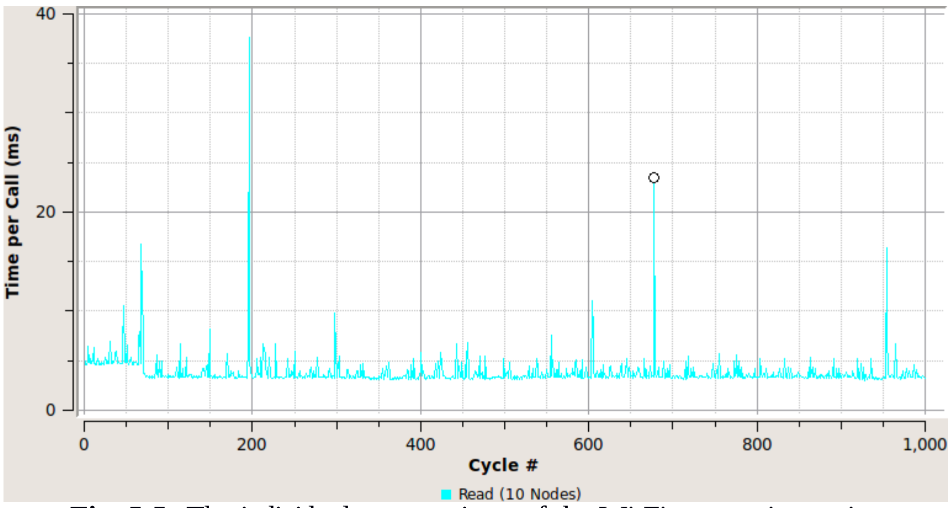


Fig. 5.5: The individual request times of the Wi-Fi connection option

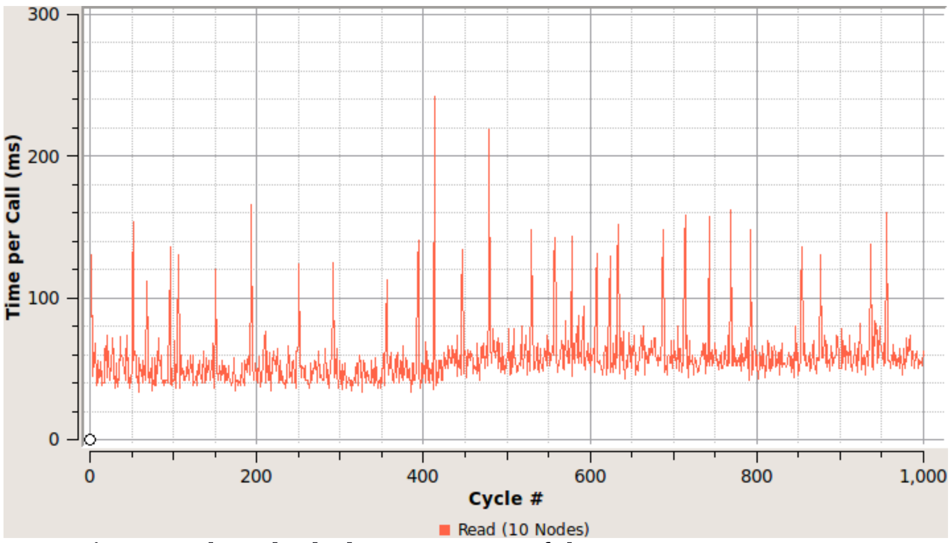


Fig. 5.6: The individual request times of the GSM connection option

The average time per call measured from the client, when another client is loading the server simultaneously, is presented in the Table 5.4.

Table 5.4: The performance of data acquisition from OPC UA server (client perspective)

	Node count	1 client		2 clients	
		Time per call, ms	CPU Utilization, %	Time per call, ms	CPU Utilization, %
Ethernet	1	1.11	~60	1.25	100
	10	1.82	~70	2.64	100
	30	3.42	~80	5.56	100
VPN	1	3.69	~40	4.95	~60
	10	4.74	~50	6.86	~80
	30	6.63	~60	10.87	~85
Wi-Fi	1	3.16	~50	3.74	~75
	10	3.8	~60	5.22	~90
	30	5.77	~70	8.28	~95
GSM	1	56.35	~5-10	105.8	~5-10
	10	57.83	~5-10	109.3	~5-10
	30	69.38	~5-10	122.0	~5-10

The increase in server response times, when another clients is constantly loading server with same type of requests, is shown graphically in Fig. 5.7.

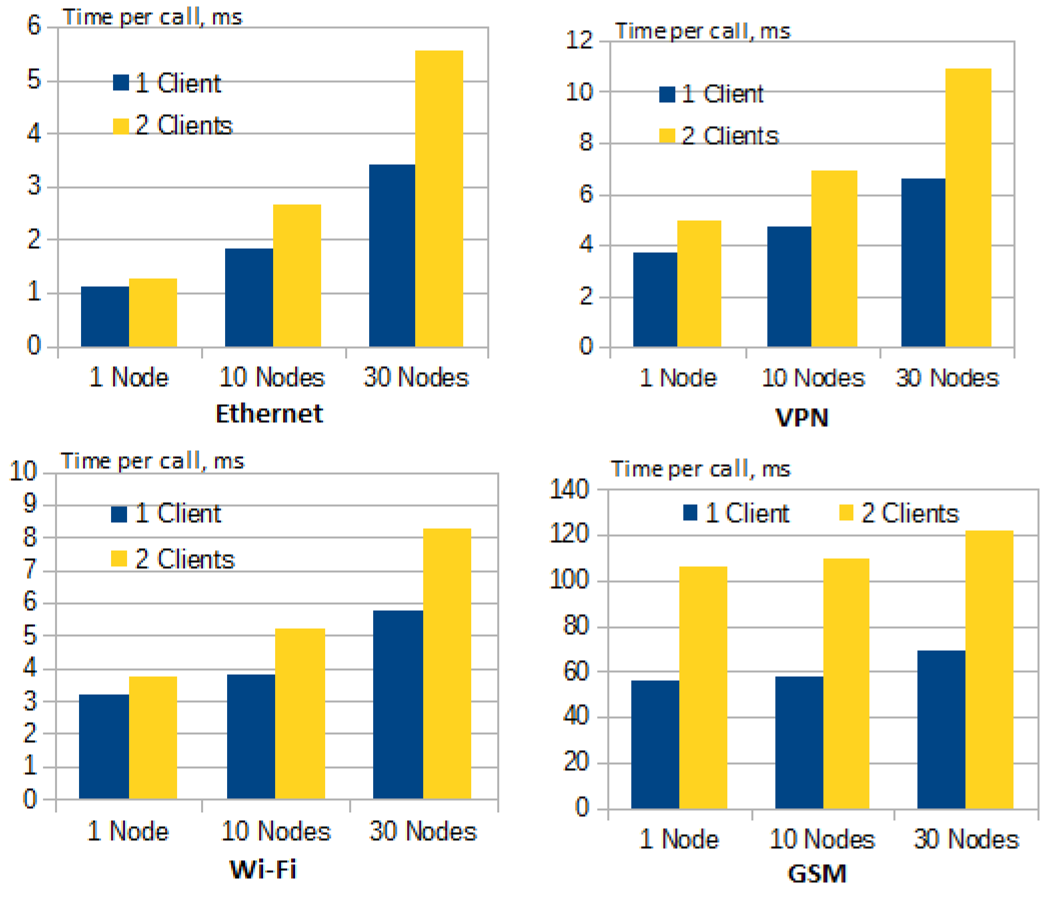


Fig. 5.7: The comparison of response times from OPC client's point of view

The results show, that using a local Ethernet connection, two clients are enough for performance to become limited by the gateway CPU. The VPN and Wi-Fi connection options also seem to be partially limited by CPU, as the relative performance difference between one and two clients loading the server increases when the requests become more complicated and CPU load becomes higher. The GSM connection, on the other hand, seems to be only limited by connection speed.

When two clients are loading the server, the performance from one client's point of view decreases, as the server also handles the requests from the other client. However, the total amount of the request handled by the server increases. Testing showed that, when the clients send the same type of requests to the OPC UA server using the same type of connection, server handles about an equal amount of request from each client. The total amount of request handled by the server when two clients are sending requests simultaneously is shown in the Table 5.5.

Table 5.5: Request handled by the OPC UA server

	Node count	1 client		2 clients	
		Total requests per second	CPU Utilization, %	Total requests per second	CPU Utilization, %
Ethernet	1	900.9	~60	1600	100
	10	549.5	~70	757.6	100
	30	292.4	~83	359.7	100
VPN	1	271.0	~50	404.0	~65
	10	211.0	~55	291.6	~75
	30	150.8	~65	184.0	~85
Wi-Fi	1	316.5	~55	534.8	~75
	10	263.2	~60	383.1	~85
	30	173.3	~68	241.6	~95
GSM	1	17.8	~5	18.9	~5
	10	17.3	~5	18.3	~5
	30	14.4	~5	16.4	~5

The increase of requests handled by the server when two clients are sending requests simultaneously is graphically shown in Fig. 5.8.

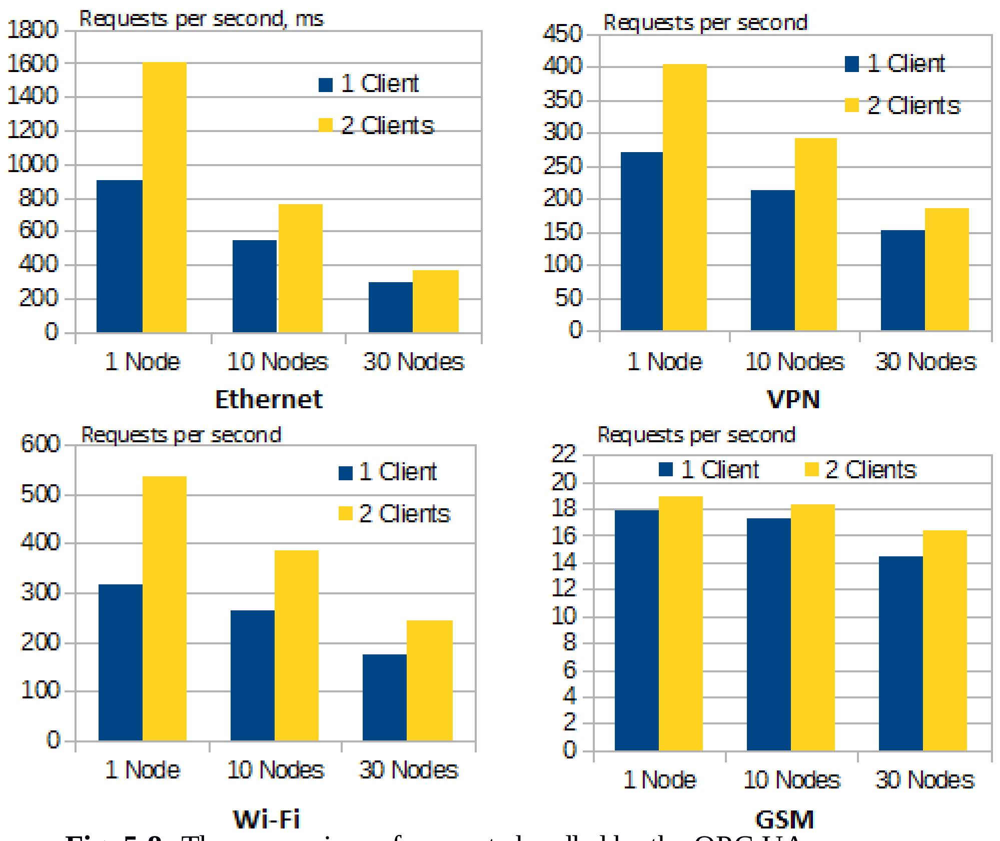


Fig. 5.8: The comparison of requests handled by the OPC UA server

5.3. OPC-PA interface performance testing

The interface between the OPC UA server and PA facilitates the support of wide range devices, however it trades performance for flexibility. To transfer data between the OPC UA server and the PA adapter, an OS switch between processes is required, as the OPC UA server and the PA are the separate processes. Furthermore, some time may be required for other processes running on the system. In order to find out, how much additional latency this interfaces actually introduces, the round trip delay time (RTD) was measured.

5.3.1. Test method

The RTD test of the PA interface was performed using dummy PA, which echoes data send to it back to the OPC UA server. The data flow in the test is shown in the UML sequence diagram in Fig. 5.9.

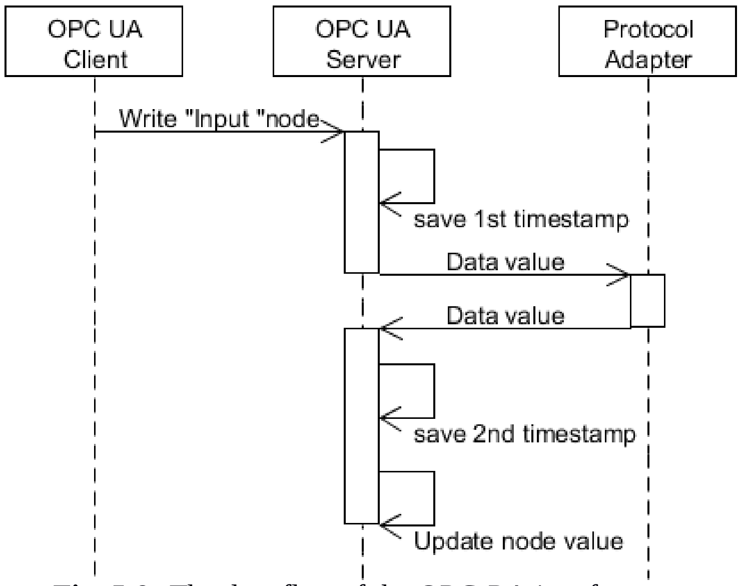


Fig. 5.9: The data flow of the OPC-PA interface test

The sequence depicted in the UML diagram above is repeated 200 times during the test. The difference between the first and the second timestamps was calculated to evaluate the delay caused by the OPC-PA interface. Timestamps were obtained using *clock_gettime()* function. The OPC UA nodes used in this test are of double data type. The system. The tests were performed without additional system load and also with *stress* utility [44] running on the system. The stress utility was configured to use two threads and run at the same scheduler priority as the OPC UA server and the PA.

5.3.2. Test results

The measured RTD of the OPC-PA interface is shown in the Table 5.6. The values of the individual measurement samples are presented in Fig. 5.10.

Table 5.6: The RTD of the OPC-PA interface

	Round trip delay time, ms	
	~5% CPU load	100% CPU (stress –c 1–i 1)
Minimum	0.43	0.74
Average	5.27	4.75
Maximum	10.58	11.44
Standard deviation	2.889	2.787

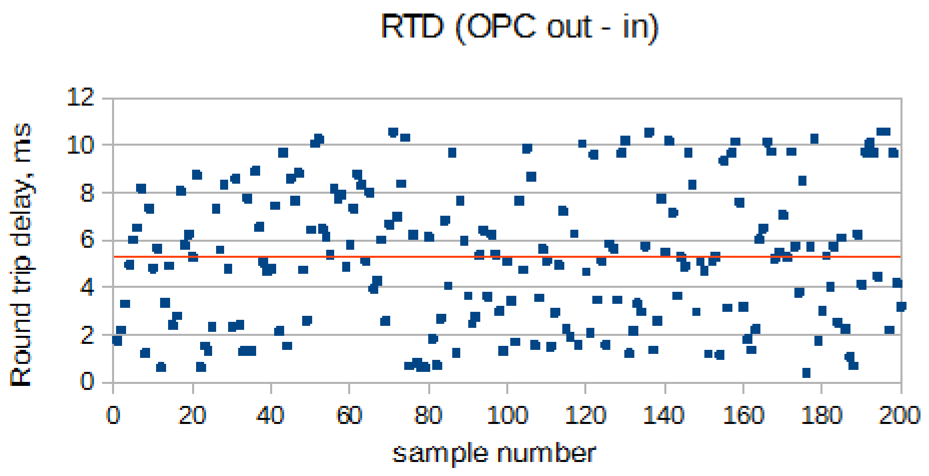


Fig. 5.10: The individual RTD samples of OPC-PA test (~5% CPU load)

The results show that RTD of communication interface between the OPC UA server and PA is 5.27 ms on average, but it varies greatly - from less than 1 ms to ~10-12 ms. Putting additional load on the system actually decreased the average RTD time, however minimum and maximum times increased. At first it was thought that this behaviour may be accidental, however re-running experiments showed the similar trend. We suspect that it may be caused by power saving and CPU frequency scaling functions in Linux kernel and putting constant load on the CPU makes it always run in high performance mode.

5.4. Wireless link performance testing

The local 169 MHz wireless link is used for communication with distributed field devices. The link was tested to evaluate the data acquisition rate from remote nodes. For this purpose RTD test was performed. It investigates how much time it takes to send a request to a remote node and get a response from it. This represents the master-slave communication, where gateway (master) is constantly polling the remote nodes (slaves).

5.4.1. Test setup

Wireless module is connected to the gateway via UART interface. The settings used in the experiment were the following:

- UART: 115200 bit/s baud rate, 8 data bits, No parity, 1 stop bit;
- RF 169MHz: 19.2 kbps data rate (4 GFSK), Receiver filter bandwidth 50 kHz;

The base structure of the radio packet, which was used in the experiment, is shown in Fig. 5.11. The tests were performed with 4 and 16 bytes in the data field (the latter was AES encrypted). The packet with 4 bytes of data was used to represent a simple packet, which could be used to transmit a single parameter or command, while 16 bytes is the size of AES block and denotes the minimum data size, when AES encryption is used.

Preamble 4 bytes	Sync-Word 3 bytes	Data length 1 byte	Data 0-127 bytes	CRC-16 2 bytes

Fig. 5.11: The structure of RF packet

The sequence of round trip delay (RTD) experiment for the wireless radio link is shown in the UML sequence diagram in Fig. 5.12.

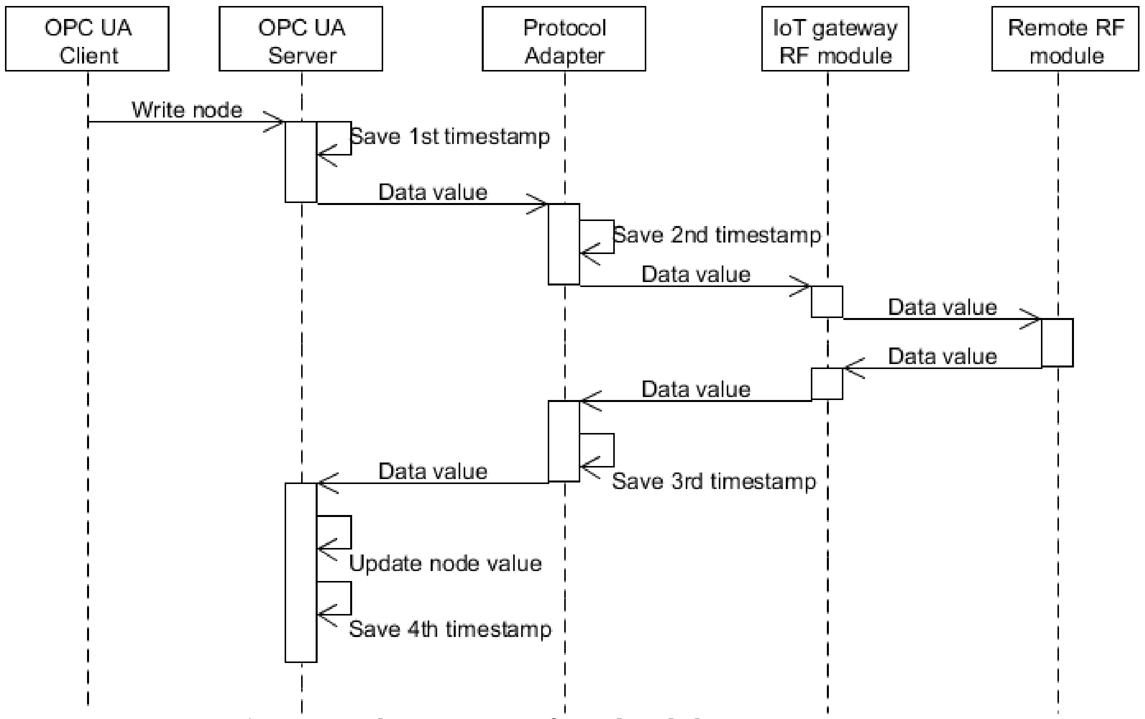


Fig. 5.12: The sequence of wireless link RTD experiment

The RTD was measured in two cases. In the first case, the delay was measured between the moment, when OPC UA server receives a request from the OPC UA client (1st timestamp), and when data becomes visible to OPC UA clients in the servers address space (4th timestamp). In the second case the delay was measured between the moment when PA sends packet to the wireless module (2nd timestamp) and the moment when it receives data back from the wireless module (3rd timestamp). The first case is meant to investigate the control and observation from an OPC UA client possibilities, while the second one - data processing by the PA possibilities. 100 samples were taken for each test case.

5.4.2. Test results

The RTD testing results are shown in the Table 5.7.

Table 5.7: Round trip delay time of the local wireless radio link

		Round trip delay time, ms			
		Avg.	Std.Dev (relative)	Min.	Max.
OPC – ... – OPC	4 bytes payload	28.657	2.676 (9.3%)	24.675	34.689
	16 bytes payload	38.803	2.588 (6.7%)	34.949	44.236
PA – ... –PA	4 bytes payload	23.510	0.019 (0.07%)	23.480	23.608
	16 bytes payload	33.834	0.134 (0.4%)	33.775	34.795

The results show that variation in RTD is mainly caused by OPC-PA interface. Sending packet through Linux drivers and wireless link, on the other hand, always takes about the same amount of time. It is clearly visible when comparing individual RTD samples in Fig. 5.13 and Fig. 5.14.

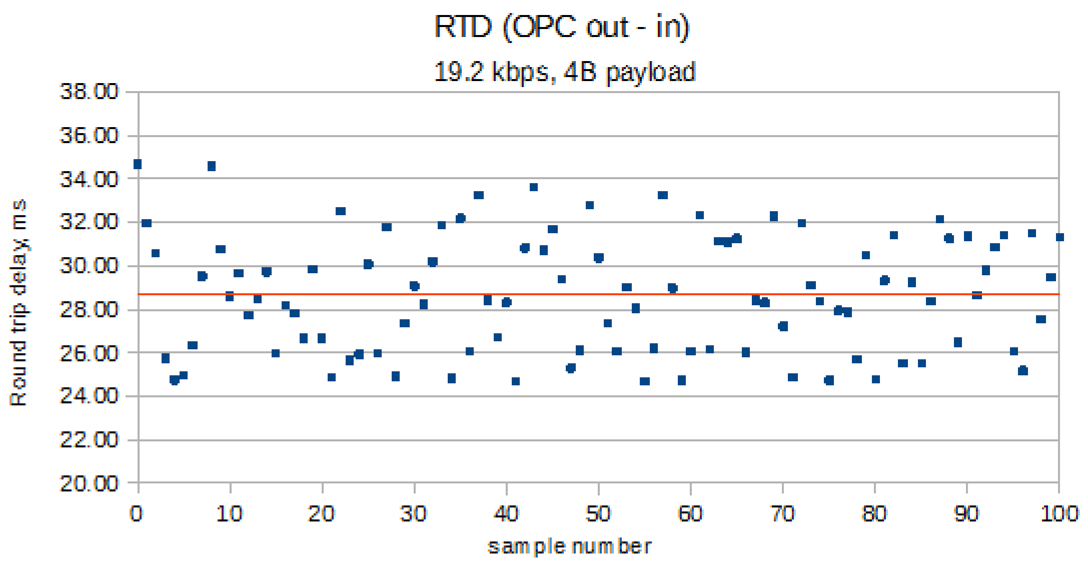


Fig. 5.13: Individual RTD samples

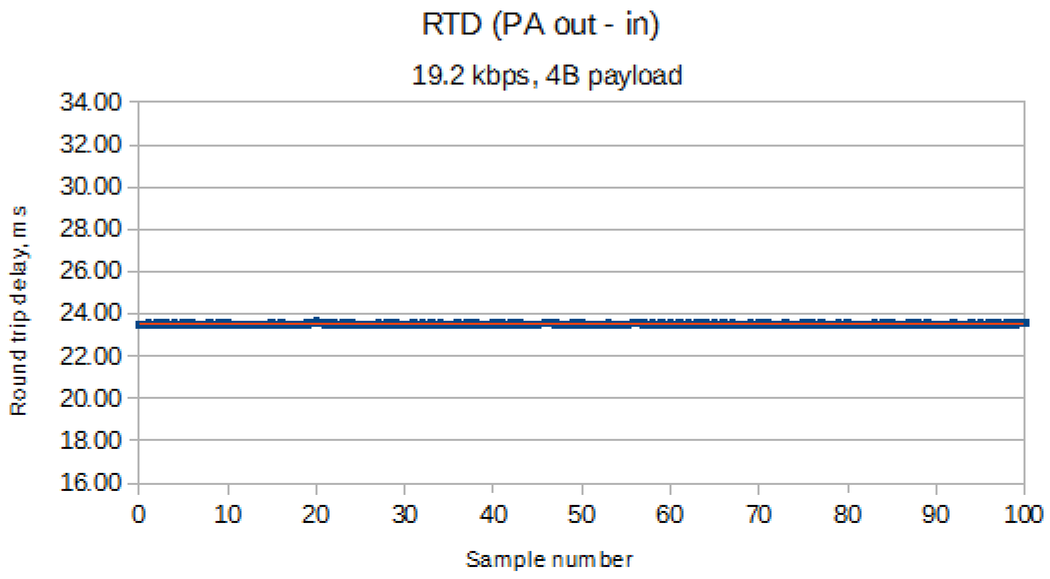


Fig. 5.14: Individual RTD samples

Using the sampling intervals measured in the wireless link RTD test, the sampling rate of remote nodes was calculated. The calculated sampling rate values are presented in Table 5.8.

Table 5.8: Sampling rate of the local wireless radio link

		Sampling rate, Hz	
		Avg.	Std.Dev (relative)
OPC – ... – OPC	4 bytes payload	35.20	3.266 (9.3%)
	16 bytes payload	25.88	1.701 (6.6%)
PA – ... –PA	4 bytes payload	42.53	0.034 (0.08%)
	16 bytes payload	29.56	0.115 (0.4%)

Sampling rate together with standard deviation are graphically shown in Fig. 5.15.

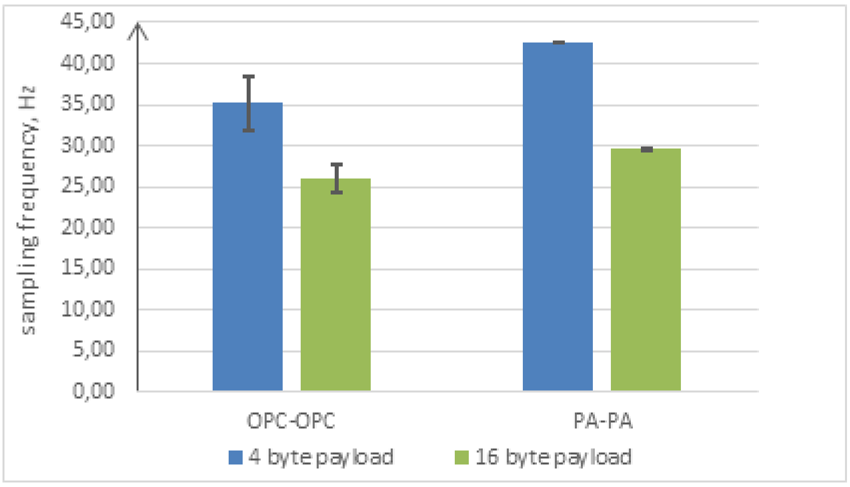


Fig. 5.15: Radio link sampling rate comparison

The results indicate that it would be possible to achieve 25-30 Hz sampling rate using AES encryption. Such sampling rate is enough to satisfy the requirements set in 1.4.1.

5.5. Power consumption testing

Power consumptions testing was performed to investigate how much certain parts of the IoT gateway affect its total power draw. The current implementation only allows us to turn off Wi-Fi and GSM modules. During the tests the 169 MHz radio was always in receive mode. The maximum CPU load was simulated using the Linux *stress* utility.

The power consumption was measured from 12 V DC source, using YOKOGAWA power analyser. The activity on wireless interfaces was simulated using file transfer over NFS as described in 5.1.1. The results of power consumption tests are presented in Table 5.9.

Table 5.9: The results of power consumption testing

Conditions			Power draw (±0.05 W)
CPU load	**Wi-Fi**	**VPN over GSM**	
¬1%	Disabled	Disabled	3.35 W
100 %	Disabled	Disabled	3.90 W
¬1%	Enabled	Disabled	3.75 W
N/A	File transfer	Disabled	4.15 W
¬1%	Disabled	Enabled	4.85 W
N/A	Disabled	File transfer	5.15 W
¬1%	Enabled	Enabled	5.2 W
100 %	Enabled	Enabled	5.8 W

The power consumption without load is relatively high and could be improved in the future by implementing an ability to disable certain components on the gateway motherboard. Loading CPU to 100% adds about 550 mW to total system power consumption compared to it being in idle state. Wi-Fi seems to add about 400 mW, while being on without load, and 800 mW when used for file transfer. Turning on GSM modem and opening VPN connection through it increases power draw by about 1.5 W and using it for file transfer adds another 300 mW.

CONCLUSION

During the course of this thesis, an OPC UA based data exchange system was successfully developed and was found feasible for the agricultural applications discussed in 1.3.

Agricultural machinery and wireless sensor networks were identified as two major data sources in agriculture.

In order to facilitate the support of various devices, a concept named protocol adapters (described in 3.3) was introduced for data exchange between the OPC UA server and field devices. The testing showed that it introduces an additional latency of about 5 ms on average. The proposed concept was tested by aggregating data from various sources into the OPC UA server. The setup proved to be feasible and stable.

The data acquisition from OPC UA server performance is more dependant on connection latency than bandwidth. The benefit of combining multiple node values into one message is also greater when using connection which has high latency. For VPN connection over GSM (high latency), going from 1 node to 30 nodes in one message decreased request rate by about 21%, while for local Ethernet connection (low latency) – about 68%. This can be exploited by OPC UA subscription service by setting a higher publishing interval to combine more values into a single message.

The experiments showed that the system is suitable for data acquisition from agricultural machinery. Using VPN over GSM connection, the OPC UA server was capable of transferring over 400 values per second, while the established requirements (in 1.4.2) were to transfer around 100 values per second. Furthermore, transfer rate is only limited by connection speed and does not put noticeable load on the gateway CPU, so it could be improved in future by replacing 3G modem with 4G modem.

The sampling rate of the local 169 MHz wireless network was found to satisfy requirements (set in 1.4.1.) for data acquisition from wireless sensors and farm automation tasks such as auto-irrigation or greenhouse climate control. The performance is good enough to exceed set requirements using AES encrypted packets.

The results of wireless link test showed that round trip delay time measured from the protocol adapter is fairly consistent with standard deviation of less than 0.15 ms, which was not expected in the system that is running OS.

REFERENCES

1. O.Vermesan, P.Friess, Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems, Aalborg: River Publishers, 2013, p. 348.
2. Warwick Ashford, IoT could be key to farming, says Beecham Research. *ComputerWeekly*, 2015 Feb, Internet access: http://www.computerweekly.com/news/2240239484/IoT-could-be-key-to-farming-says-Beecham-Research
3. J. Hadders, AGRI YIELD MANAGEMENT: The next revolution in agriculture, DACOM BV, Netherlands, 2009, p. 44.
4. Hewlett-Packard, Internet of Things Research Study Report, 2014. Internet access: http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA5-4759ENW.pdf
5. OPC Foundation, „OPC UA Specification Part 1: Overview and concepts ", OPC UA Specification release 1.02, 2012.
6. Crop, Livestock and Forests Integrated System for Intelligent Automation. Internet access: http://www.clafis-project.eu/
7. G. Fellidis,V. Garrick, S. Pocknee, et. al., How wireless will change agriculture. In: Stafford, J.V. (Ed.), Precision Agriculture '07 – Proceedings of the Sixth European Conference on Precision Agriculture (6ECPA), Skiathos, Greece, p. 57-67.
8. Ning Wang, Naiqian Zhang, Maohua Wang, Wireless sensors in agriculture and food industry—Recent development and future perspective, Computers and Electronics in Agriculture 50 (2006) p. 1–14.
9. Vijay N. Application of Sensor Networks in Agriculture // Third International Conference on Sustainable Energy and Intelligent Systems. – Tamilnadu, India, December 2012. p. 1-6.
10. Aqeel-ur-Rehman, Abu Zafar Abbasi, Noman Islam, Zubair Ahmed Shaikh. A review of wireless sensors and networks' applications in agriculture, Computer Standards & Interfaces 36 (2014) p. 263–270.
11. Mittal A., Chetan K.P., Jayaraman S., Jagyashi B.G. mKRISHI Wireless Sensor Network Platform for Precision Agriculture // Sixth International Conference on Sensing Technology. - Kolkata, 2012, p. 623 – 629.
12. L.Bencini, D.Di Palma, G. Collodi, G. Manes, Wireless Sensor Networks for On-field Agricultural management Process. Italy, University of Florence, December 14, 2010. Internet access: http://cdn.intechopen.com/pdfs-wm/12465.pdf
13. NetSense, Vinesense: more quality and less cost. Internet access: http://www.netsens.it/en/solutions-for-defence-against-pathogens-systems-1.php
14. Libelum, Waspmote Datasheet. Internet access: http://www.libelium.com/downloads/documentation/waspmote_datasheet.pdf

15. Libelium, Meshlium Datasheet. Internet access: http://www.libelium.com/downloads/documentation/meshlium_datasheet.pdf
16. ISO 11783-9:2012. Tractors and machinery for agriculture and forestry -- Serial control and communications data network -- Part 9: Tractor ECU
17. ISO 11783-7:2009. Tractors and machinery for agriculture and forestry -- Serial control and communications data network -- Part 7: Implement messages application layer
18. ISO 11783-6:2009 Tractors and machinery for agriculture and forestry -- Serial control and communications data network -- Part 6: Virtual terminal
19. SAE International, SAE J1939/71 Vehicle application layer.
20. ISO 11783-10:2012. Tractors and machinery for agriculture and forestry -- Serial control and communications data network -- Part 10: Task controller and management information system data interchange
21. A.K. Tripathya, J. Adinarayanaa, K. Vijayalakshmib, et. al.,Knowledge discovery and Leaf Spot dynamics of groundnut crop through wireless sensor network and data mining techniques // Computers and Electronics in Agriculture107, Sepy 2014, P. 104–114.
22. Yun Hwan Kim, Seong Joon Yoo, Yeong Hyeon Gu, et. al., Crop Pests Prediction Method using Regression and Machine Learning Technology: Survey // International Conference on Future Software Engineering and Multimedia Engineering, 2013. P. 52-56.
23. Naiqian Zhang, Maohua Wang, Ning Wang. Precision agriculture a worldwide overview // Computers and Electronics in Agriculture 36, 2002, p. 113-132.
24. Andy Beck, Hans Nissen (John Deere), ISOBUS Task Controller Workshop, 2008. Internet access: http://aem.org/Documents/NAIITF/Documents/NAIITF-ISOBUSWorkshop-TaskController.pdf
25. Moussa EL Jarroudia, Louis Kouadiob, et. al., Economics of a decision–support system for managing the main fungal diseases of winter wheat in the Grand-Duchy of Luxembourg // Field Crops Research 172, Feb 2015, p. 32–41.
26. Sherine M. Abd El-kader, Basma M. Mohammad El-Basioni, Precision farming solution in Egypt using the wireless sensor network technology // Egyptian Informatics Journal 13, November 2013, p 221-233.
27. Damas, M., Prados, A.M., G´omez, F., Olivares, G., 2001. HidroBus system: fieldbus for integrated management of extensive areas of irrigated land // Microprocessors and Microsystems 25, p. 177–184.
28. Zhou Yiming, Yang Xianglong, Guo Xishan, Zhou Mingang, Wang Liren, A Design of Greenhouse Monitoring & Control System Based on ZigBee Wireless Sensor Network // WiCom 2007, Shanghai, 2007, p. 2563 – 2567.
29. ISO 11783-10:2014. Tractors and machinery for agriculture and forestry -- Serial control and communications data network -- Part 12: Diagnostics services

30. OPC Foundation, „OPC UA Specification Part 11: Historical Access“, OPC UA Specification release 1.02, 2012.
31. OPC Foundation, „OPC UA Specification Part 8: Data Access“, OPC UA Specification release 1.02, 2012.
32. OPC Foundation, „OPC UA Specification Part 3: Address Space Model “, OPC UA Specification release 1.02, 2012.
33. OPC Foundation, „OPC UA Specification Part 5: Information Model “, OPC UA Specification release 1.01, 2009.
34. OPC Foundation, „OPC UA Specification Part 4: Services“, OPC UA Specification release 1.01, 2009.
35. OPC Foundation, „OPC UA Specification Part 9: Alarms and Conditions “, OPC UA Specification release 1.02, 2012
36. OPC Foundation, „OPC UA Specification Part 2: Security Model “, OPC UA Specification release 1.02, 2012.
37. OPC Foundation, „OPC UA Specification Part 6: Service Mappings “, OPC UA Specification release 1.02, 2012.
38. VARISCITE LTD., VAR-SOM-AM33v2.1 Datasheet :Texas Instruments Sitara AM335x-based System-on-Module. Internet access: http://www.variscite.com/images/stories/DataSheets/VAR-SOM-AM33_v2_1_datasheet_101.pdf
39. OpenVPN, Overview. Internet access: https://openvpn.net/index.php/open-source/overview.html
40. Linux man page: PPPD, Internet access: http://linux.die.net/man/8/pppd
41. Apache.org, Apache Web Server Project. Internet Access: http://httpd.apache.org/
42. LIGHTTPD project page: Internet access: http://www.lighttpd.net/
43. T. Stover, Demystifying Unix Domain Sockets. 2011. Internet access: http://www.thomasstover.com/uds.html
44. Kernel.org, SocketCAN, https://www.kernel.org/doc/Documentation/networking/can.txt
45. Harvard SEAS, stress project page, http://people.seas.harvard.edu/~apw/stress/

APPENDIX

Appendix 1. C header file of OPC-PA interface library

PAClient.h

```
/*
 * File:   PAclient.h
 * Author: Ignas
 *
 * Created on September 16, 2014, 11:25 AM
 */

#ifndef PACLIENT_H
#define PACLIENT_H

#ifdef  __cplusplus
extern "C" {
#endif

//Socket file definition
#define OPCSOCKET                "/tmp/OPCSOCKET"

#define PA_PACKET_SIZE             65000
//Packet types definitions
#define UPDATE_PACKET_TYPE       0
#define RELOAD_PACKET_TYPE       2
#define APPEND_PACKET_TYPE       3
#define GETID_PACKET_TYPE        4
#define DELETE_PACKET_TYPE       5

//Status code definitions
#define STATUS_GOOD              0x0000//Everything is as expected.
#define STATUS_GOOD_ASYNC        0x002E//The processing will complete asynchronously
#define STATUS_GOOD_OVERLOAD     0x002F//Sampling has slowed down due to resource limitations
#define STATUS_GOOD_CALL_AGAIN   0x00A9//The operation is not finished and needs to be called again
#define STATUS_NON_CRIT_TIMEOUT 0x00AA//A non-critical timeout occurred
#define STATUS_UNCERTAIN         0x4000//Something unexpected occurred but the results still maybe usable
#define STATUS_LASTVAL_NOCOMM    0x408F//communication to the data source has failed. The variable value is the last value
that had a good quality.
#define STATUS_LASTVAL           0x4090//Whatever was updating this value has stopped doing so.
#define STATUS_SUBSTITUTE_VAL    0x4091//The value is an operational value that was manually overwritten.
#define STATUS_INIT_VAL          0x4092//The value is an initial value for a variable that normally receives its value
from another variable
#define STATUS_NOT_ACCURATE      0x4093//The value is at one of the sensor limits.
#define STATUS_SUBNORMAL         0x4095//The value is derived from multiple sources and has less than the required number
of Good sources.
#define STATUS_DATA_SUBNORMAL    0x40A4//The value is derived from multiple values and has less than the required number
of Good values.
#define STATUS_BAD               0x8000//An error occurred.
#define STATUS_UNEXPECTED_ERR    0x8001//An unexpected error occurred.
#define STATUS_INTERNAL_ERR      0x8002//An internal error occurred as a result of a programming or configuration error.
#define STATUS_MEMORY            0x8003//Not enough memory to complete the operation.
#define STATUS_NO_RESOURCE       0x8004//An operating system resource is not available.
#define STATUS_COMM_ERR          0x8005//A low level communication error occurred.
#define STATUS_TIMEOUT           0x800A//The operation timed out.
#define STATUS_PERMISION         0x801F//User does not have permission to perform the requested operation.
#define STATUS_NO_COMM           0x8031//Communication with the data source is defined, but not established, and there is
no last known value available.
#define STATUS_WAIT_INIT_DATA    0x8032//Waiting for the server to obtain values from the underlying data source.
#define STATUS_INV_DATA          0x8038//The data encoding is invalid.
#define STATUS_INV_ID            0x8034//The syntax of the node id is not valid.
#define SATUS_UNKNOWN_ID         0x8034//The node id refers to a node that does not exist in the ddress space.
#define STATUS_NOT_WRITABLE      0x803B//The access level does not allow writing to the Node.
#define STATUS_BAD_VALUE         0x803C//The value was out of range.
#define STATUS_NOT_SUPPORTED     0x803D//The requested operation is not supported.
#define STATUS_NO_DELETE         0x8069//The server will not allow the node to be deleted.
#define STATUS_NO_ARGUMENT       0x8076//The client did not specify all of the input arguments for the method.
#define STATUS_CONFIG_ERR        0x8089//There is a problem with the configuration that affects the usefulness of the
value.
#define STATUS_DEV_FAIL          0x808B//There has been a failure in the device/data source that generates the value that
has affected the value.
#define STATUS_SENSOR_FAIL       0x808C//There has been a failure in the sensor from which the value is derived by the
device/data source.
#define STATUS_NO_SERVICE        0x808D//The source of the data is not operational.
#define STATUS_FILE_NOT_FOUND    0x8109//Invalid file name specified.

#ifndef SUCCESS
#define SUCCESS 0
#endif

//Type definitions for size sensitive variables
```

```c
typedef unsigned char UINT8;
typedef unsigned short UINT16;
typedef unsigned int UINT32;
typedef unsigned long long UINT64;

typedef signed char INT8;
typedef short INT16;
typedef int INT32;
typedef long long INT64;

/**
 * @brief Gets the current time in FILETIME format which is based on 100 nanosecond ticks
 * since January 1, 1601 00:00:00.
 * @return - 100 ns ticks since January 1, 1601 00:00:00.
 */
UINT64 GetFiletimeTime();

/**
 * @brief Converts Unix time stored in struct timespec(in second and nanoseconds)
 * to FILETIME format
 * @param tv - struct timespec containing Unix time
 * @return - time in FILETIME format
 */
UINT64 UnixToFiletime(struct timespec tv);

/**
 * @brief Initializes PA interface to OPC UA server and sets a function that will be
 * called when data is received from OPC UA. Information received from OPC UA will be
 * passed as arguments to this function.
 * @param ns_name - unique namespace name used to identify PA.
 * @param function - function that must take node id (indicating a parameter or action),
 * pointer to value data and data length as arguments.
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PAInitInterface(const char* ns_name,void (*function)(UINT32 id,void* data, unsigned data_len));

/**
 * @brief Attempts to reconnect to OPC-UA server socket.
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PAReconnect();

/**
 * Closes PA to OPC UA interface
 */
void PACloseInterface();

/**
 * @brief Sends "Update message" to update parameter value in OPC UA server.
 * Automatically adds timestamp containing current time and STATUS_GOOD status code.
 * @param id - id of parameter
 * @param data - pointer to value to be sent
 * @param data_len - length of value
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PASendData(UINT32 id, const void* data, unsigned data_len);

/**
 * @brief Sends "Update message" to update parameter value and status in OPC UA server.
 * @param id - id of parameter
 * @param data - pointer to value to be sent
 * @param data_len - length of value
 * @param status - status code indicating success or cause of error
 * @param timestamp - value time stamp
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PASendDataEx(UINT32 id, const void* data, unsigned data_len,UINT16 status, UINT64 timestamp);

/**
 * @brief Sends "Update message" to report that an error occurred.
 * Automatically adds timestamp containing current time.
 * @param id - id of parameter or method
 * @param status - status code indicating cause of error
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PASendStatus(UINT32 id, UINT16 status);

/**
 * @brief Send request tp OPC UA to delete all nodes in namespace managed by PA and load
 * new nodes from specified file.
 * @param filename - OPC UA Nodeset 2.0 XML file containing description of new nodes
 * @return - status code.
 */
int PASendReloadNamespace(const char* filename);
```

```
/**
 * @brief Send request tp OPC to add nodes (described in XML file) to namespace managed
 * by PA
 * @param filename - OPC UA Nodeset 2.0 XML file containing description of new nodes
 * @return - status code.
 */
int PASendAppendNamespace(const char* filename);


/**
 * @brief Sets log file for PA
 * @param fname - path of log file
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PAOpenLogFile(const char* fname);

/**
 * @brief Writes message to log file
 * @param text - message to
 * @return - 0 (SUCCESS) on success, -1 on failure.
 */
int PALogToFile(const char* text);

/**
 * @brief removes node(s) from opc UA address space
 * @param ns_name - namespace name of the node
 * @param id - numeric id of the node
 * @return - status code.
 */
int PADeleteNode(const char* ns_name,UINT32 id);

#ifdef  __cplusplus
}
#endif

#endif  /* PANODECLIENT_H */
```

Appendix 2. NodeSet 2.0 XML file example: Weather station PA

```
<UANodeSet xmlns="http://opcfoundation.org/UA/2011/03/UANodeSet.xsd">
    <NamespaceUris>
        <Uri>WeatherStation</Uri>
    </NamespaceUris>
  <Aliases>
    <Alias Alias="SByte">i=2</Alias>
    <Alias Alias="Byte">i=3</Alias>
    <Alias Alias="Int16">i=4</Alias>
    <Alias Alias="UInt16">i=5</Alias>
    <Alias Alias="Int32">i=6</Alias>
    <Alias Alias="UInt32">i=7</Alias>
    <Alias Alias="Int64">i=8</Alias>
    <Alias Alias="Float">i=10</Alias>
    <Alias Alias="Double">i=11</Alias>
    <Alias Alias="String">i=12</Alias>
    <Alias Alias="Organizes">i=35</Alias>
    <Alias Alias="HasTypeDefinition">i=40</Alias>
    <Alias Alias="HasSubtype">i=45</Alias>
  </Aliases>
  <UAObject NodeId="ns=1;i=0" BrowseName="1:WS">
        <DisplayName>Weather Station</DisplayName>
        <References>
        <Reference ReferenceType="Organizes">ns=1;i=1</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=2</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=3</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=4</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=5</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=6</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=7</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=8</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=9</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=10</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=20</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=30</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=40</Reference>
        <Reference ReferenceType="HasTypeDefinition">i=61</Reference>
        <Reference ReferenceType="Organizes" IsForward="false">i=1000</Reference>
        </References>
    </UAObject>
    <UAVariable DataType="Float" NodeId="ns=1;i=1" BrowseName="1:Temperature" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Air Temperature, Â°C</DisplayName>
```

```xml
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Byte" NodeId="ns=1;i=2" BrowseName="1:Leafs" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Leaf weatness</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Byte" NodeId="ns=1;i=4" BrowseName="1:Humidity" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Humidity, %</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Float" NodeId="ns=1;i=3" BrowseName="1:Precipitation" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Accumulated precipitation, mm</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="UInt16" NodeId="ns=1;i=5" BrowseName="1:Pressure" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Air pressure, mBar</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Float" NodeId="ns=1;i=6" BrowseName="1:WindSpeed" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Wind Speed, m/s</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="UInt16" NodeId="ns=1;i=7" BrowseName="1:WindDeg" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Wind Direction, Â°</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=8" BrowseName="1:longitude" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Longitute</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=9" BrowseName="1:latitude" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Latitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
  <UAObject NodeId="ns=1;i=10" BrowseName="1:SoilSense1">
        <DisplayName>Soil sensor 1</DisplayName>
        <References>
        <Reference ReferenceType="Organizes">ns=1;i=11</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=12</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=17</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=18</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=19</Reference>
        <Reference ReferenceType="HasTypeDefinition">i=61</Reference>
        </References>
   </UAObject>
    <UAVariable DataType="Float" NodeId="ns=1;i=11" BrowseName="1:SoilT1" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil temperature, Â°C</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Byte" NodeId="ns=1;i=12" BrowseName="1:SoilM1" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil moisture, %</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=18" BrowseName="1:SoilLong1" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Longitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=19" BrowseName="1:SoilLat1" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Latitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Float" NodeId="ns=1;i=17" BrowseName="1:SoilD1" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Depth, cm</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAObject NodeId="ns=1;i=20" BrowseName="1:SoilSense2">
        <DisplayName>Soil sensor 2</DisplayName>
        <References>
        <Reference ReferenceType="Organizes">ns=1;i=21</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=22</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=27</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=28</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=29</Reference>
        <Reference ReferenceType="HasTypeDefinition">i=61</Reference>
        </References>
   </UAObject>
    <UAVariable DataType="Float" NodeId="ns=1;i=21" BrowseName="1:SoilT2" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil temperature, Â°C</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Byte" NodeId="ns=1;i=22" BrowseName="1:SoilM2" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil moisture, %</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=28" BrowseName="1:SoilLong2" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Longitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=29" BrowseName="1:SoilLat2" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Latitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Float" NodeId="ns=1;i=27" BrowseName="1:SoilD2" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Depth, cm</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
  <UAObject NodeId="ns=1;i=30" BrowseName="1:SoilSense3">
        <DisplayName>Soil sensor 3</DisplayName>
        <References>
        <Reference ReferenceType="Organizes">ns=1;i=31</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=32</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=37</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=38</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=39</Reference>
        <Reference ReferenceType="HasTypeDefinition">i=61</Reference>
        </References>
   </UAObject>
    <UAVariable DataType="Float" NodeId="ns=1;i=31" BrowseName="1:SoilT3" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil temperature, Â°C</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Byte" NodeId="ns=1;i=32" BrowseName="1:SoilM3" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil moisture, %</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=38" BrowseName="1:SoilLong3" UserAccessLevel="3" AccessLevel="3">
```

```
        <DisplayName>Longitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=39" BrowseName="1:SoilLat3" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Latitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Float" NodeId="ns=1;i=37" BrowseName="1:SoilD3" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Depth, cm</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
  <UAObject NodeId="ns=1;i=40" BrowseName="1:SoilSense4">
        <DisplayName>Soil sensor 4</DisplayName>
        <References>
        <Reference ReferenceType="Organizes">ns=1;i=41</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=42</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=47</Reference>
        <Reference ReferenceType="Organizes">ns=1;i=48</Reference>
    <Reference ReferenceType="Organizes">ns=1;i=49</Reference>
        <Reference ReferenceType="HasTypeDefinition">i=61</Reference>
        </References>
   </UAObject>
    <UAVariable DataType="Float" NodeId="ns=1;i=41" BrowseName="1:SoilT4" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil temperature, Â°C</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Byte" NodeId="ns=1;i=42" BrowseName="1:SoilM4" UserAccessLevel="5" AccessLevel="5"
Historizing="true">
        <DisplayName>Soil moisture, %</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=48" BrowseName="1:SoilLong4" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Longitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Double" NodeId="ns=1;i=49" BrowseName="1:SoilLat4" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Latitude</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>
    <UAVariable DataType="Float" NodeId="ns=1;i=47" BrowseName="1:SoilD4" UserAccessLevel="3" AccessLevel="3">
        <DisplayName>Depth, cm</DisplayName>
        <References>
        <Reference ReferenceType="HasTypeDefinition">i=63</Reference>
        </References>
    </UAVariable>

</UANodeSet>
```

Appendix 3. PA source code example: CAN PA

```
#include <time.h>
#include <libgen.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <linux/can.h>
#include <linux/can/bcm.h>
#include <sys/un.h>
#include "PAclient.h"
```

```c
#include "can_config.h"

#define U64_DATA(p) (*(unsigned long long*)(p)->data)

unsigned running = 1;
unsigned active = 0; //status indicating whether we are receiving messages from EBS
int s;              //CAN socket file descriptor
float pressure=0.0f;

/*this function is used to start transmitting message to wabco device
and later update content of tranmitted messages.
It is irrelevant if we only want to listen to messages sent by device*/
void CallbackFunction(UINT32 id, void* val, unsigned len)
{

    //check if node id is 100, which is defined as brake pressure deamand value in XML
    //also check if status indicating that we are receiving messages from EBS is set
    if ((id==100)&&(active))
    {
     //frame for CAN BCM
     struct {
          struct bcm_msg_head msg_head;
          struct can_frame frame[1];
       } msg;

    pressure = *((float*)val);       //data type is defined as float in XML file

    if (pressure > 500.0f)           //just limiting value range
    {
     PASendStatus(100,STATUS_BAD_VALUE);
     pressure = 500.0f;
     PASendData(100,&pressure,sizeof(pressure)); //change value in OPC UA
     }
    else if (pressure < 0)
        {
          PASendStatus(100,STATUS_BAD_VALUE);
          pressure = 0.0f;
          PASendData(100,&pressure,sizeof(pressure)); //change value in OPC UA
          }

    long brakePreas = pressure*256.0f/5.0f+0.5; //bit value = 5/256 kPA

    //setup CAN TX
    memset(&msg,0,sizeof(msg));
    msg.msg_head.opcode  = TX_SETUP;
    msg.msg_head.can_id  = 0xC02C820|CAN_EFF_FLAG;
    msg.msg_head.nframes = 1;
    msg.msg_head.ival2.tv_sec = 0;
    msg.msg_head.ival2.tv_usec = 10000; //interval 10 ms
    msg.frame[0].can_dlc   = 8;
    msg.frame[0].can_id  = 0xC02C820|CAN_EFF_FLAG; //PGN 00200,PR=3,DP=0,PF=2,DS=0xC8,SA=0x20
    msg.frame[0].data[0]=0x00;
    msg.frame[0].data[4]=0;
    msg.frame[0].data[5]=125;
    msg.frame[0].data[6]=0xFF;
    msg.frame[0].data[7]=0xFF;

    if (active == 2)  //update TX messages
    {
       //change transmitted message data
       msg.msg_head.flags   = TX_CP_CAN_ID;
       msg.frame[0].data[1]= brakePreas == 0 ? 0xF0 :0xF1; //brake light switch
       msg.frame[0].data[2]= brakePreas&0xFF;          //new brake pressure demand value
       msg.frame[0].data[3]= (brakePreas>>8)&0xFF;
       write(s, &msg, sizeof(msg));
       }
       else           //first time TX setup, done only once
       {
          //Setup TX frames
          msg.msg_head.flags   = SETTIMER|STARTTIMER|TX_CP_CAN_ID;
          msg.frame[0].data[1]= 0xF0;
          msg.frame[0].data[2]= 0;
          msg.frame[0].data[3]= 0;
          write(s, &msg, sizeof(msg));

          //setup second CAN message, EBS doesn't react if we dont send this
          msg.msg_head.can_id  = 0x18FEC920|CAN_EFF_FLAG;
          msg.frame[0].can_id=0x18FEC920|CAN_EFF_FLAG;//PGN 0FEC9,PR=6,DP=0,PF=254,PS=201,SA=0x20
          msg.msg_head.ival2.tv_usec = 100000; //interval 100 ms
          msg.frame[0].data[0]=0xFC;
          msg.frame[0].data[1]= 0xFF;
          msg.frame[0].data[2]= 0xC0;
          msg.frame[0].data[3]= 0xFF;
          msg.frame[0].data[4]=0xFF;
          msg.frame[0].data[5]=0xFF;
```

```
            write(s, &msg, sizeof(msg));
            active = 2;  //dont do this again
        }
    }
    else PASendStatus(id,STATUS_INV_ID); //there are no other writable nodes defined in XML
}


int main(int argc, char **argv)
{
    struct ifreq ifr;
    struct sockaddr_can addr;
    float fval = 0;
    UINT32 ival;
    UINT16 sval;
    char bval=0;
    UINT16 status;
    UINT64 tt;
    int error = 0;
    int last_t = 0;

    PAOpenLogFile("/tmp/PAwabco.log"); //init log file, comment this to print messages to stdout

    system("canconfig.sh");           //to make sure that can0 interface is up

    /*this adapter has namespace name "Wabco"
    CallbackFunction is only used to send data to wabco EBS. We can replace it with NULL in case
    only listening of messages sent by device is required*/
    if (PAInitInterface("Wabco",CallbackFunction)==-1) //init PA to OPC UA interface
    {
        PALogToFile("Failed to initialize PA interface");
        return -1;
    }

    //open CAN socket, use CAN BCM for filtering and automatic TX
    if ((s = socket(PF_CAN, SOCK_DGRAM, CAN_BCM))<0)
    {
        PALogToFile("Error opening CAN socket");
        return -1;
    }

    //connect to CAN BCM
    addr.can_family = AF_CAN;
    strcpy(ifr.ifr_name, "can0");
    if (ioctl(s, SIOCGIFINDEX, &ifr))
    {
        PALogToFile("Error: CAN ioctl");
        return 1;
    }
    addr.can_ifindex = ifr.ifr_ifindex;

    if (connect(s, (struct sockaddr *)&addr, sizeof(addr))<0)
    {
        PALogToFile("Error: CAN connect");
        return 1;
    }

    //prepare structure for BCM packets
    struct {
      struct bcm_msg_head msg_head;
      struct can_frame frame[1];
    } msg;

    memset(&msg,0,sizeof(msg));
    //Setup RX frames (total 3 CAN messages)
    //notifications will only be received only when data change in packet is detected

    //1st message
    msg.msg_head.opcode  = RX_SETUP;
    msg.msg_head.can_id  = 0xC0320C8|CAN_EFF_FLAG; //PGN 0x00300,PR=3,DP=0,PF=3,DA=0x20,SA=0xC8
    msg.msg_head.flags   = SETTIMER|STARTTIMER|RX_NO_AUTOTIMER;
    msg.msg_head.nframes = 1;
    msg.msg_head.count = 0;
    msg.msg_head.ival1.tv_sec = 0;
    msg.msg_head.ival1.tv_usec = 0;
    msg.msg_head.ival2.tv_sec = 0;
    msg.msg_head.ival2.tv_usec = 0;
    U64_DATA(&msg.frame[0]) = 0xFFFFFFFFFFFFFFFFULL; /* data change mask */
    write(s, &msg, sizeof(msg));

    //2nd message
    msg.msg_head.can_id  = 0x18FEC4C8|CAN_EFF_FLAG;//PGN 0x0FEC4,PR=6,DP=0,PF=254,PS=196,SA=0xC8
    write(s, &msg, sizeof(msg));

    //3rd message
    msg.msg_head.can_id  = 0x18FEC6C8|CAN_EFF_FLAG;//PGN 0x0FEC6,PR=6,DP=0,PF=254,PS=198,SA=0xC8
    write(s, &msg, sizeof(msg));

    while (running) //loop
    {

      while (error != 0) //reconnect to PA interface
      {
          PALogToFile("PA interface:send error, reconnecting");
          sleep(1);
          error = PAReconnect();

       }

    //refresh pressure value in OPC server once in a while
     if (last_t + 600 <= time(NULL))
     {
      last_t = time(NULL);
      if ((error = PASendData(100,&pressure,sizeof(pressure)))<0)
         continue;
     }

    //Reading CAN messages and decoding data
    //Error or not available status is indicated by values 3,4 for bit signals
    //254,255 for bytes, etc (as defined in ISO 11992)
    //Node ids and data types of parameters are as defined in XML file
     if (read(s,&msg,sizeof(msg))>0)
     {
      if (msg.msg_head.opcode == RX_CHANGED)
      {
       if (msg.msg_head.can_id  == (0xC0320C8|CAN_EFF_FLAG)) //EBS 21 (PGN 0x00300)
       {
           tt = GetFiletimeTime();
           bval = msg.frame[0].data[0]&3; //vehicle ABS active/pasive
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(3,&bval,sizeof(bval),status,tt))<0)
               continue;
           bval = (msg.frame[0].data[0]>>2)&3; //vehicle retarder control active/pasive
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(4,&bval,sizeof(bval),status,tt))<0)
               continue;
           bval = (msg.frame[0].data[0]>>4)&3; //vehicle service brake active/pasive
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(5,&bval,sizeof(bval),status,tt))<0)
                   continue;
           bval = (msg.frame[0].data[0]>>6)&3; //automatic towed vehicle braking active/pasive
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(6,&bval,sizeof(bval),status,tt))<0)
                   continue;
           bval = msg.frame[0].data[1]&3; //VDC active
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(7,&bval,sizeof(bval),status,tt))<0)
               continue;
           fval = (float)msg.frame[0].data[2]/256+msg.frame[0].data[3]; //wheel based vehicle speed
           status = msg.frame[0].data[3] < 254 ? STATUS_GOOD : bval == 254 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(22,&fval,sizeof(fval),status,tt))<0)
               continue;
           fval = (float)msg.frame[0].data[4]*0.4; //actual percentage of retarder peak torque
            status = msg.frame[0].data[4] < 254 ? STATUS_GOOD : bval == 254 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(23,&fval,sizeof(fval),status,tt))<0)
               continue;
           fval = -125.0+(float)msg.frame[0].data[5]/256+msg.frame[0].data[6]; //wheel speed difference main axle
            status = msg.frame[0].data[6] < 254 ? STATUS_GOOD : bval == 254 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(24,&fval,sizeof(fval),status,tt))<0)
               continue;
       }
       else if (msg.msg_head.can_id  == (0x18FEC4C8|CAN_EFF_FLAG))//EBS 22 (PGN 0x0FEC4)
       {
           if (active ==0) //status indicating that we are receiving medssages from EBS
               active = 1;
           tt = GetFiletimeTime();
           bval = msg.frame[0].data[1]&3; //Vehicle EBS elsectrical supply sufficient/insuficient
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(8,&bval,sizeof(bval),status,tt)) < 0)
               continue;
           bval = (msg.frame[0].data[1]>>2)&3;//red warning signal request
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(9,&bval,sizeof(bval),status,tt))<0)
               continue;
           bval = (msg.frame[0].data[1]>>4)&3;//amber warning signal request
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
           if ((error = PASendDataEx(10,&bval,sizeof(bval),status,tt))<0)
               continue;
           bval = (msg.frame[0].data[1]>>6)&3;//electrical supply of non-breaking systems
           status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
```

```
            if ((error = PASendDataEx(11,&bval,sizeof(bval),status,tt))<0)
                continue;
            bval = msg.frame[0].data[2]&3; //spring brake installed
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(12,&bval,sizeof(bval),status,tt))< 0)
                continue;
            bval = (msg.frame[0].data[2]>>2)&3; //electric load proportion function
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(13,&bval,sizeof(bval),status,tt))<0)
                continue;
            bval = (msg.frame[0].data[2]>>4)&3;//vehicle type
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(14,&bval,sizeof(bval),status,tt)) < 0)
                continue;
            bval = (msg.frame[0].data[2]>>6)&3;//spring brake engaged
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(15,&bval,sizeof(bval),status,tt))<0)
                continue;
            bval = msg.frame[0].data[3]&3;//Loading ramp approach assistance
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(16,&bval,sizeof(bval),status,tt))<0)
                continue;
            bval = (msg.frame[0].data[3]>>2)&3;//supply line braking request
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(17,&bval,sizeof(bval),status,tt))<0)
                continue;
            ival = 2*msg.frame[0].data[4]+512*msg.frame[0].data[5];//axle load sum
             status = msg.frame[0].data[5] < 254 ? STATUS_GOOD : bval == 254 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
             if ((error = PASendDataEx(25,&ival,sizeof(ival),status,tt))<0)
                continue;
             //reference retarder torque
             status = msg.frame[0].data[7] < 254 ? STATUS_GOOD : bval == 254 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
             if ((error = PASendDataEx(26,&msg.frame[0].data[6],sizeof(sval),status,tt)) <0 )
                continue;
        }
        else //EBS 23 (PGN 0x0FEC6)
        {
            tt = GetFiletimeTime();
            bval = msg.frame[0].data[0]&3; //tyre pressure sufficient/insufficient
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(18,&bval,sizeof(bval),status,tt))<0)
                continue;
            bval = (msg.frame[0].data[0]>>2)&3;//brake lining sufficeint/insufficient
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(19,&bval,sizeof(bval),status,tt))<0)
                continue;
            bval = (msg.frame[0].data[0]>>4)&3;//brake temperature status
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(20,&bval,sizeof(bval),status,tt)) < 0)
                continue;
            bval = (msg.frame[0].data[0]>>6)&3;//vehicle pneumatic supply sufficient/insufficient
            status = bval < 2 ? STATUS_GOOD : bval == 2 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
            if ((error = PASendDataEx(21,&bval,sizeof(bval),status,tt))<0)
                continue;
            sval = (UINT16)msg.frame[0].data[7]*5;//pneumatic supply pressure
             status = msg.frame[0].data[7] < 254 ? STATUS_GOOD : bval == 254 ? STATUS_DEV_FAIL : STATUS_NO_SERVICE;
             if ((error = PASendDataEx(27,&sval,sizeof(sval),status,tt))<0)
                continue;
        }
        }
      }
     }

    close(s);
    PACloseInterface();
    return 0;
}
```