Heikki Tahvanainen

OPC UA performance evaluation

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 14.3.2016

Thesis supervisor:

D.Sc. Ilkka Seilonen

Thesis advisor:

M.Sc. (Tech.) Jouni Aro

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Heikki Tahvanainen

Title: OPC UA performance evaluation

Date: 14.3.2016 Language: English Number of pages: 7+56

Department of Automation and Systems Technology

Professorship: Information and computer systems in automation

Supervisor: D.Sc. Ilkka Seilonen

Advisor: M.Sc. (Tech.) Jouni Aro

OPC Unified Architecture is an industrial communication specification, employing a client-server communication model. In this thesis, performance testing of OPC UA applications is explored.

First, performance testing of software applications is discussed on a general level. After this, details of OPC UA and previous studies of OPC UA performance are examined. Based on this information, example test cases are designed. The testing procedure and results of the tests are presented in detail. As an important part of the work, the design and implementation of a generic test client for read and subscription testing is presented.

The results provide information about overhead of OPC UA security modes in two different hardware platforms and comparison of two server applications in terms of read and subscription functionality. Substantial differences can be observed in terms of maximum number of requests as well as in the resulting resource usage. When overloaded, the servers again exhibit differences in their behaviour. The test results should be thought of as examples of features, limits and differences that can be observed by performance testing OPC UA applications. The results of this thesis are used in the future development of the Prosys OPC UA Java SDK.

Keywords: OPC UA, performance testing

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Heikki Tahvanainen		
Työn nimi: OPC UA suorituskyvyn arviointi		
Päivämäärä: 14.3.2016	Kieli: Englanti	Sivumäärä: 7+56
Automaatio- ja systeemitekniikan laitos		
Professuuri: Automaation tietotekniikka		
Työn valvoja: D.Sc. Ilkka Seilonen		
Työn ohjaaja: TkT Ilkka Seilonen		

OPC Unified Architecture on erityisesti teollisiin ympäristöihin suunniteltu tiedonsiirtomäärittely, joka käyttää asiakas-palvelin kommunikointimallia. Tässä työssä tutkitaan OPC UA -sovellusten suorituskykytestausta.

Ensin muodostetaan yleisellä tasolla käsitys tavoista mitata ohjelmistojen suorituskykyä. Tämän jälkeen OPC UA -määrittelyn yksityiskohtia käydään läpi tähän työhön liittyvin osin. Aiemmin aihepiiristä julkaistujen tutkimusten avulla muodostetaan kuva siitä, mitä ja miten oikeastaan kannattaa tutkia. Suoritettavat testit dokumentoidaan mahdollisimman hyvin ja testien tulokset raportoidaan yksityiskohtaisesti. Tärkeänä osana työtä esitellään yleiskäyttöisen testisovelluksen suunnittelu ja toteutus.

Tutkimuksen perusteella esitetään tietoa OPC UA -viestien tietoturvalliseen käsittelyyn kuluvasta resurssien kulutuksesta kahdella eri laitteistolla ja vertaillaan kahden OPC UA -palvelintoteutuksen ominaisuuksia. Toteutuksissa voidaan havaita selkeitä eroja käsiteltyjen pyyntöjen maksimimäärässä sekä palvelinohjelmistojen resurssienkäytössä. Kun palvelimiin tehdään enemmän pyyntöjä kuin ne ehtivät käsitellä, on tuloksena taas selkeitä eroja eri implementaatioiden välillä. Esiteltävät testitulokset pätevät tässä tietyssä konfiguraatiossa ja ne tulee käsittää esimerkkeinä siitä minkälaisia ominaisuuksia, rajoituksia ja eroja OPC UA -sovelluksissa on havaittavissa. Työn tuloksia tullaan käyttämään Prosys OPC UA Java SDK:n jatkokehityksessä.

Avainsanat: OPC UA, suorituskykytestaus

Preface

I would like to thank my instructor Jouni Aro for providing the thesis topic and for his support during the whole process. Another thanks goes to Ilkka Seilonen who operated as my supervisor. All the comments and improvements were appreciated. This thesis has been made at Prosys PMS Ltd, Espoo, Finland. Everyone at Prosys deserves a thanks for providing a good atmosphere to work in. Special thanks goes to my sister Hanna, who helped with the valuable job of proofreading.

Otaniemi, 3.3.2016

Heikki Tahvanainen

Contents

Abbreviations

Abbreviations

OPC	OLE for Process Control
OPC UA	OPC Unified Architecture
SCADA	Supervisory Control And Data Acquisition
SDK	Software Development Kit
GUI	Graphical User Interface
LAN	Local Area Network
XML	Extensible Markup Language
PLC	Programmable Logic Controller
CA	Certificate Authority
QA	Quality Assurance
JVM	Java Virtual Machine
JDK	Java Development Kit

1 Introduction

1.1 Background and motivation

Information transfer is an essential factor in today's automation systems. Currently, industrial networks are often characterized with automation pyramid approach. [1] OPC Unified Architecture (abbreviated OPC UA) is a software specification which creates a common infrastructure for sharing information between different systems. The most common use case is an OPC UA server acting as a gateway exposing data from underlying data source to OPC UA clients residing on higher level of this conceptual automation pyramid. OPC Classic, UA's predecessor, enabled data from automation instruments of different manufacturers to be accessed using one protocol. OPC UA is an evolution of this previous standard and an attempt to create an extensible specification to expose and access data regardless of the purpose or meaning of the data, enabling interoperability between any number of systems of different purposes [2]. OPC UA is an IEC standard IEC 62541 [3]. OPC UA specification addresses data communication in a client-server architecture and also defines information modeling in the form of an address space model, server discovery services, security modes and subscription mechanism [4]. Among other things, the actual data communication is an important feature of OPC UA specification.

OPC UA is already a mature specification used in industrial environments. Things like scalability and performance are growing in importance as more use cases are explored. When developing a product or service utilizing OPC UA data communications it is a matter of importance for the developer and user to know the possible performance constraints. The importance of performance studies can be seen by looking at recent studies where, for example, OPC UA performance has been studied in the context of an agricultural system [5] and in the context of smart electric grids [6]. It seems that performance studies are carried out at the same time when other study goals are the main objective. Thus, there is a need for a study focusing on OPC UA performance.

The evaluation of performance aspects provides valuable information for a system designer or for end-users. Mostly, end-users are not interested in detailed information. They want to know if a given service may be used for their application scenario, and if so, which parameters seem to have the biggest impact for communication. Using results from pre-tested platforms, system designers can quickly gauge the level of hardware performance they require and possibly eliminate the need to test systems themselves. [7] With historical benchmarking data, it is also possible to show for example an improvement in design and implementation when developing OPC UA products. Organizations that are currently migrating their existing production systems to OPC UA are interested in performance figures. [8] One aspect regarding this migration is that the new system must preserve or increase the quality of communication performance. For this reason some organizations must run their own performance tests when considering migration. These organizations themselves declare that they would benefit from performance tests that would help them choose between protocols.

1.2 Objectives and scope

This thesis explores performance testing of OPC UA applications considering especially server application testing. The work will be done considering specially a proprietary SDK developed for Java by Prosys PMS Ltd [9]. The objective of this thesis is to answer some of the performance related questions that system designers using OPC UA will face. To meet the goals of answering OPC UA performance evaluation questions, the thesis should provide information on how tests were developed as well as providing generic benchmarking results where possible. In the case of OPC UA, only a few ready made performance testing tools exist so it is necessary to device one during this thesis. Actual implementation of a publicly available generic performance testing tool is not in the scope of this thesis. However, the presented information can be used as a base knowledge for future development. The main research questions of this thesis are: 1) what kind of features and limits can be observed in OPC UA applications by performance testing and 2) what kind of issues surround performance testing of OPC UA applications.

1.3 Research methods

The main sources for information in this work have been previous studies of OPC UA performance and existing performance testing tools of OPC UA and other information systems. Relatively many publications exist on the topic of OPC UA performance. Previous studies provide information on OPC UA performance as well as help to understand which areas will be most fruitful for further studying. This thesis concentrates on experimental testing. Ready made performance testing tools exist for many information systems (for example web servers [10]). The existing performance testing tools are examined as examples.

1.4 Structure of the work

This thesis is organized into multiple sections. First, a look at performance testing of software systems in general is presented. Next, a brief introduction into OPC UA specification is provided. After this, a look at previous studies of OPC UA performance evaluation is made. Areas that need further studying in this topic are selected based on previous research. After the interesting topics are selected, practical aspects of developing a test client are documented in the chapter research material and methods. Then the results chapter documents the obtained results. At the end, conclusions chapter presents the most important results in a compact form and answers the research questions decided here in the beginning.

2 Performance evaluation of software systems

2.1 Measuring application performance

Performance Testing refers to testing done to analyze and improve the performance of an application. The focus here is on the optimization of resource consumption by analyzing data collected during testing. Performance Testing to a certain extent should be done by developers, but more elaborate, large scale testing may be conducted by a separate performance team. In some organizations, the performance team is a part of the QA function. Sometimes performance testing that is focused on understanding how an application scales is called Scalability Testing. As such scalability testing can be viewed as a kind of performance testing where the goal is to understand at what point the application stops scaling and identify the reasons for this. Scalability can be described in many ways, ranging from loose definitions referring to software's ability to extract the most performance out of the available resources, to more strict definitions like the ability to improve throughput or capacity when additional computing resources are added. **Load Testing** refers to the kind of testing usually done by QA organizations to ensure that the application can handle a certain load level. Criteria are set to ensure that releases of a product meet certain conditions, like the number of users they can support while delivering a certain response time. The best term for tests presented in this thesis is performance testing.

Performance measurement can be defined as the process of quantifying the effectiveness and efficiency of an action. Effectiveness is a measure of the level of performance and efficiency is a measure of how economically resources are utilized when providing a given level of performance. For example, if a server application handles 1000 requests per seconds and consumes 50% of the host machine CPU time, then the 1000 requests per second is a measurement of the effectiveness of the application and CPU usage is a measure of the efficiency. Performance is related to both time (speed of communication) and space (load and resources). These classifications are sometimes called speed and utilization indices [11]. The distinction between the two classes of measures is that the speed indices provide an absolute measure of the performance and utilization indices provide a basis to understand where the bottlenecks are. For example by measuring network bandwidth utilization it can be shown that bottleneck is not the network, but internal architecture of the server and utilization of other hardware resources [12]. Commonly measured effectiveness metrics are for example system throughput (number of requests handled per time period) and response time for a request. In addition to these, performance study might also report efficiency metrics, including measures such as CPU utilization and communication line utilization. Evaluated utilization metrics can also be memory or disk space, message size, queue lengths or even the energy consumed by different services [13]. This classification is illustrated in table 1.

Table 2 specifies a way to categorize performance tests based on their level of abstraction. Application software tests compare the performance or functionality of applications or application bundles. System software tests provide the basis for evaluation and comparison of software that serves applications and is not necessarily

Table 1: Classification of benchmark results [11].

speed indices (effectiveness)	utilization indices (efficiency)
System throughput	CPU usage
Response time for a request	Communication line usage
Time needed to activate a session	Length of system's queues
	Memory usage
	Message size
	Energy consumed

directly used by an end user. OPC UA communication SDKs, which are the basis for this thesis, belong to this system software category. Micro-benchmarks can be used to compare for example two different algorithms that accomplish the same thing. Same kind of classification is presented in the context of hardware testing.

Table 2: Categories for computer system performance testing [11].

Main category	Sub category	Description
Software	Application software	Compare performance and functionality of specific applications or application bundles, e.g. mail servers
	System software	Compare performance and functionality of software that serves applications, e.g. database systems
	Micro-benchmark	Measure the performance of a very small and specific piece of software
Hardware	System	Assessment of a system in its entirety
	Component	Assessment of specific parts of a computer system (for example graphical processing unit)
	Micro-benchmark	Measure the performance of a very small and specific piece of hardware

System software testing, the most suitable classification for tests done in this thesis, can further be divided into multiple categories. Some tests measure "how fast" a given unit of work can be processed or acknowledged; others measure "how much" work can be performed with a given quantity of computing resources [14]. A usual way to classify performance tests is to categorize them to **Elapsed time**, **Throughput** or **Response time** tests [15]. The simplest way of measuring performance is to select a certain task and see how long it takes to accomplish. This is called elapsed time or batch measurement. These measurements measure the amount of time (or other resources) that must be used to accomplish certain workload. Conversely, throughput measurements measure the amount of work that can be accomplished in a certain period of time. The most usual throughput test in client-server applications is that a client sends a request to the server. When it receives a response, it immediately sends a new request. That process continues; at the end of the test, the client reports the total number of operations that it achieved. [15, pp. 25] One defining factor is that in throughput tests, the workload is not fixed.

Contrary to throughput measurements, in response time tests, the effectiveness of the server is based on how quickly it responds to a fixed load. The most important factor to measure is the amount of time that elapses between the sending of a request from a client and the receipt of the response. The difference between a response time test and a throughput test is that clients in a response time test sleep for some period of time between operations. This is referred to as think time [15]. When think time is introduced into a test, throughput becomes fixed. At that point, the important measurement is the response time for the request. The effectiveness of the server is based on how quickly it responds to the fixed load. These different categorizations are illustrated in table 3.

Usually response time is reported as an average value or as a percentile. For example, if 90% of responses are less than 1.5 seconds and 10% of responses are greater than 1.5 seconds, then 1.5 seconds is the 90th percentile response time. If data contains huge outliers, then percentile values may give a more accurate view than average value. Huge outliers are rare in general, but smaller outliers may occur because of multiple reasons. In Java applications garbage collection introduces pause times which may easily cause outliers in measurements.

Table 3: Test categorization

	Elapsed time	Throughput	Response time
Workload	Constant	Variable	Constant
Throughput	Variable	Variable	Constant

2.2 Usefulness of performance testing

Best performance results would of course be obtained by profiling production applications, however this may be impossible in most cases. The other possibilities are empirical approach (making test application) or theoretical (using simulation to obtain results). There is a tradeoff here: differences observed in tests run with actual applications show results that are results of different implementations of certain protocol but may be platform dependent. With theoretical or simulated results there is no real guarantee of applicability of results. Also, obtaining reliable results by profiling an application is by no means trivial. Performance testing is a usual practice. However, there is always a risk of wrong performance testing results. For this reason, it is important to concentrate on getting useful results and to be careful in order to avoid potential errors. The following list contains aspects which can be used to assess the usefulness of performance testing [16].

- Relevancy – The results reflect important issues of the domain.
- Repeatability – The test is able to run multiple times yielding the same result.
- Fairness – All the compared systems are able to participate equally.
- Verifiable – There is confidence that the documented result is real.
- Economical – Users afford to run the test.

Not all of the criteria have to be fulfilled in perfection, but most successful benchmarks are strong in one or two of these aspects [16]. Often, in order to satisfy the last four of these items, a test developer must choose to give up on some of the first. This means that creating economical, verifiable, fair and repeatable tests often lowers its amount of relevancy in practice. Relevant tests have among other things understandable metric and a target audience that wants the information. Perhaps the most important requirement is appropriate use of software. The software features should be used in the way that a typical customer application would. Appropriate representation is an important factor also. A performance test result is not very useful if there is not a high degree of confidence that it represents the actual performance of the system under test.

Repeatability can become an issue when measuring performance [17]. For example, the amount of logging usually has a substantial effect on performance (this is also the reason why applications usually have different log levels). Managed runtime systems, such as Java, have their own quirks. Just-in-time compilation of Java causes the identical code to perform more effectively over time and this means that certain warm-up period must be taken into account in order to get reliable results. Java applications also build up garbage in the Java heap that must be cleaned out, which then introduces regular garbage collection pauses. Performance testing should try to ensure repeatability and consistency. There is a trade-off between repeatability and reality. One of these trade-offs is the creation of a steady-state period within the benchmark. Real applications are hardly steady in the way that they generate work on the system, but a benchmark where results will be compared requires either that the application and associated performance does not change over a period of time or that the exact same work flow runs for each iteration of the benchmark.

Testing client-server applications is cumbersome because it is not necessarily clear whether the client side or server side is the bottleneck. This can lead to results that are hard to verify. All client-server tests run the risk that the client cannot send data quickly enough to the server [15]. This may occur because there are not enough CPU cycles on the client machine to run the desired number of threads, or because the client has to spend a lot of time processing the request before it can send a new request. In those cases, the test is effectively measuring the client performance rather than the server performance, which is usually not the goal. The risk depends on the amount of work that each client thread performs. A throughput-oriented test is more likely to encounter this situation, since each client thread is performing a lot of work.

The relevancy of results needs to be considered especially in the case of multiple measurements. It is common for tests that measure throughput also to report the average response time of its requests. Changes in that number are not indicative of a performance problem unless the reported throughput is the same. A server that can sustain 500 operations per second with a 0.5 second response time is performing better than a server that reports a 0.3 second response time but only 400 operations per second. [15] Throughput measurements are usually taken after a suitable warm-up period, particularly since what is being measured is not a fixed set of work.

2.3 Examples of other performance testing frameworks

2.3.1 httperf

When considering tools for performance measurement, examples could be taken from other information systems. For example database systems, computer networks, email servers and http servers all have performance measurement tools which are developed to measure values specific for the information system at hand. One example of long-lived web server testing software is httperf [10]. Httperf has been originally implemented in 1998 so as of writing this it has existed for 17 years. Considering that the web is a highly dynamic system and subject to relatively frequent and fundamental changes it is a major accomplishment for a tool to stay topical for this long. Surely there are also more modern tools that are more capable, but nevertheless httperf is still widely in use and can thus be used as an example design. To put it simply, httperf overloads a server beyond its normal capacity. Using the tool, you saturate the server with TCP connections. Each httperf test will consist of a session (or sessions) made up of calls spaced out at certain intervals.

This same kind of server performance testing tool could be useful in the case of OPC UA servers. One of the premises in the design of httperf has been the separation of performing HTTP calls from issues such as what kind of workload and measurements should be used [10]. In the same sense, it should be beneficial to have a common tool which could be used to test multiple different UA servers.

2.3.2 Faban

Faban is free and open-source, Java-based load generator. It can be used to create and execute workloads. [18] Faban comes with a fhb program, that can be used to measure the performance of a URL [15].

```
fhb –W 1000 –r 300/300/60 –c 25 http://host:port/testurl
```

This example measures 25 clients making requests to the server. Each request has a 1-second cycle time. The benchmark has a 5-minute (300-second) warm-up period, 5 minute measurement period and a 1-minute ramp-down period. Following the test, fhb reports the OPS and response times for the test. Because there is think time in this example, the response times will be the important metric and operations per second will be constant (unless the server is overloaded by this workload). In addition to the fhb program, Faban has a framework for defining benchmark load generators in Java.

3 OPC UA

3.1 Overview

OPC Unified Architecture is already a mature industrial communication specification and many good introductions to the subject exist [2] [19]. Another brief introduction is provided here in the context of this thesis. OPC UA is essentially an application-layer protocol which is designed to be used in automation and industrial use cases. The primary purpose of an application-layer protocol is to define how processes running on different end systems pass messages to each other. An application-layer protocol defines the types of messages (requests and responses), the syntax of the messages (fields), the meaning of information in these fields and rules determining when and how a process sends messages and responds to messages [20]. Applications operating in the control layer of the industrial computer systems are designed not only to perform real-time data exchange between each other, but also to communicate information to higher levels (SCADA, MES, ERP). OPC Unified Architecture is an example of a standard that is designed to handle this type of communication. The fundamental parts of OPC UA are data transport and information modeling [2, pp.19]. It leaves it up to other organizations to define what data is described in their information models.

OPC UA protocol is a classical example of the client-server communication model. All communication between OPC UA applications is based on the exchange of messages initiated by the client application. [4] Usually the OPC UA server resides on an automation device. One of the areas where OPC UA has really excelled is PLC interfaces. An OPC UA server encapsulates the source of process information like a PLC and makes the information available via its interface. An OPC UA client, for example SCADA system, connects to the OPC UA server and can access and consume the offered data. Applications consuming and providing data can be both client and server at the same time. In its current form, OPC UA applications mostly use binary encoding and TCP transport protocol called UA TCP to access automation data. Also, OPC UA communication is inherently tightly coupled and session-oriented. In many applications this connection oriented approach is advantageous, for example timely detection of communication failure and rapid recovery with no loss of data can be achieved with this approach. However keeping track of sessions must in theory introduce some boundaries on scalability of OPC UA applications.

OPC UA is an evolution of the previous OPC standard (currently referred to as OPC Classic). [2] To understand OPC UA, it is best to become familiar with reasons leading to the creation of the original OPC Classic protocol. The motivation for the original OPC standard was to provide access to intelligent field devices provided by multiple vendors and thus enable connecting industrial devices to control and supervision applications. The development of OPC standard began in 1995 as an answer to this connectivity problem. The standard became successful and in fact has become a de facto standard in factory automation for integrating different production automation related software in multivendor environments. This original version was denoted OPC Data Access (DA). Later on OPC Classic developed to contain several different protocol specifications, for example Alarms & Event (AE) and Historical Data Access (HDA) among others. These specifications broadened the scope of OPC Classic.

OPC classic was based on proprietary COM/DCOM technology which restricted its use practically to PC computers in LAN networks because of scalability and security issues. Also the robustness of OPC was a problem and the segmentation to different parts was considered negative. The motivation for the development of OPC UA was to offer modern equivalent to OPC without loss of functionality or performance. [2] The name Unified Architecture literally means unifying different OPC specifications under one umbrella. OPC UA was designed to be better than OPC in terms of information security, platform independence and enhanced information modeling. OPC Foundation, which is the organization developing OPC, introduced this new version in 2009.

3.2 Parts

OPC UA standard consists of 13 parts, which are listed in table 4. As of writing this, the latest OPC UA specification version 1.03 was released by OPC Foundation in 2015. The specification describes UA internal mechanisms, which get handled through the communication stack and are mostly only of interest for those that port a stack to a specific target or those that want to implement their own UA stack. The OPC UA application developers usually code using some commercial SDK and therefore mainly use API documentation. However, parts 3, 4, and 5 may also be of interest for application developers. These parts describe the address space model, services and the default information model of OPC UA.

All data in OPC UA communication is accessed via objects in address space. Specification part 3 describes the address space and its objects. [21] Part 3 is the OPC UA meta model on which OPC UA information models are based. The primary objective of the OPC UA address space is to provide a standard way for servers to represent objects to clients. Objects and their components are represented in the address space as a set of nodes described by attributes and interconnected by references. Specification part 5 defines the standardised nodes of the address space of an empty OPC UA Server. These nodes are standardised types as well as standardised instances used for diagnostics or as entry points to server-specific nodes. The default information model contains about 5000 predefined nodes [22]. However, it is not expected that all servers will provide all of these nodes [23].

OPC UA services are defined in specification part 4 and organized into nine service sets as illustrated in table 5. Out of these, Discovery, Secure channel and Session service sets are related to the underlying connection and the rest are used to view, modify and use the data that is available from the information model. [24] Each OPC UA Service described in Part 4 has a request and response message. The defined Services are considered abstract because no particular mechanism for implementation is defined in part 4. Part 6 specifies concrete mappings supported for implementation.

Table 4: OPC UA specification parts.

Part	Description
Part 1	Overview and concepts.
Part 2	Describes the security model.
Part 3	Address Space Model. Describes the OPC UA meta model on which OPC UA information models are based.
Part 4	Services. Defines collection of abstract Remote Procedure Calls (RPC) that are implemented by OPC UA Servers and called by OPC UA Clients.
Part 5	Information Model. Describes standardised Nodes of a Server's AddressSpace.
Part 6	Mappings. Specifies relation between parts 2, 4 and 5 and the physical network protocols.
Part 7	Profiles. Describes profiles which are used to segregate features of OPC UA products.
Part 8	Data Access. Defines how clients may read, write or monitor DataItems.
Part 9	Alarms and Conditions. Specifies the representation of Alarms and Conditions.
Part 10	Programs. Defines the information model and associated behaviour for programs.
Part 11	Historical Access. Defines the information model and associated behaviour for Historical Access (HA).
Part 12	Discovery. Specifies how Clients and Servers interact with DiscoveryServers.
Part 13	Aggregates. Defines the information model associated with Aggregates

3.3 Mappings

OPC UA standard is defined in such a way that same abstract features can be implemented with different protocols. This is done to increase the flexibility of the standard. [2] Building actual OPC UA applications still requires that the implementation details have been agreed upon. Specification part 6 describes mapping between the security model described in Part 2, the abstract service definitions, described in Part 4, the data structures defined in Part 5 and the physical network protocols that can be used to implement the OPC UA specification [25]. Mapping specifies how to implement an OPC UA feature with a specific technology. For example, the OPC UA binary encoding is a mapping that specifies how to serialize OPC UA data structures as sequences of bytes. Mappings can be organized into three groups: data encodings, security protocols and transport protocol as illustrated in table 6. Security Protocol ensures the integrity and privacy of UA Messages that are exchanged between OPC UA applications. Data encoding is a way to serialize OPC UA messages and data structures. Transport protocol represents a way to

Table 5: OPC UA service sets.

Service set	Use case
Discovery	Discover servers and their security settings.
Secure channel	Services related to the security model.
Session	Maintain the session between a client and a server.
Node management	Modify the address space.
View	Browse through the address space.
Attribute	Read and write attributes of nodes.
Method	Call methods.
Monitored item	Setup monitoring for attribute value changes or events.
Subscription	Subscribe for attribute value changes or events.

exchange serialized OPC UA messages between OPC UA applications.

Table 6: OPC UA mappings.

Group	Options
Data encodings	UA Binary
	UA XML
Security protocols	UA-SC
Transport protocols	UA-TCP
	HTTPS

OPC UA currently specifies two encodings: OPC UA Binary and XML. OPC UA Binary has been designed with performance and overhead on the wire in mind. This is the most efficient way for data exchange in OPC UA. In addition to this, the specification defines XML encoding. This structured format offers for example easier debugging of messages. The cost of any structured encoding format such as XML is the overhead introduced in the form of larger messages. Large majority of current OPC UA applications use UA Binary encoding.

There is currently one security protocol defined for OPC UA in order to map the abstract services defined in part 4 of the specification: UASecureConversation. This security protocol is based on a certificate-based connection establishment. UASecureConversation is a security protocol defined by the OPC UA working group. It is a combination of techniques and mechanisms of the standards TLS4 and WSSecureConversation [2, pp. 196]. Previously there was also a second optional security protocol called WS-SecureConversation. However, in the version 1.03 of the specification this is marked as deprecated because of not being widely adopted by industry.

For transporting data, OPC UA currently has two technology mappings supported, a TCP protocol based UA TCP and a HTTPS protocol. These protocols are used for establishing a connection between an OPC UA client and server at network level. UA TCP is by far the most widely used one of the transport protocols. UA TCP is a protocol defined by OPC Foundation on top of TCP. UA TCP contains application level configurable buffer sizes for sending and receiving data, the possibility to use

Table 7: OPC UA transport facets.

Transport protocol	Security protocol	Serialization
UA-TCP	UA-SC	UA Binary
HTTPS	SSL/TLS	UA Binary
HTTPS	SSL/TLS	UA XML

same IP-address and port for different endpoints of OPC UA server and the possibility to react on and recover from errors occurring at transport level [2, pp. 198]. HTTPS refers to HTTP Messages exchanged over an SSL/TLS connection. In this protocol, OPC UA messages are transferred in the body of HTTP messages. Previously there was also a third optional transport protocol called SOAP-HTTP. However, in the version 1.03 of the specification this is marked as deprecated because of not being widely adopted by industry. This protocol was based on SOAP web service technologies.

The combination of data encoding, security protocol and transport protocol is referred to as transport facet by the OPC UA specification. The standard currently defines three different transport facets (see table 7). In order for OPC UA communication to work between two applications, both must support at least one mutual transport facet. In practice, binary UA-TCP is the most common transport facet in use today.

3.4 Subscription model

A subscription is the context to exchange data changes and event notifications between server and client. Clients send publish requests to servers and receive notifications in the form of publish responses. The biggest benefit of the subscription model is efficiency compared to polling values periodically. A client can subscribe for three different types of information from an OPC UA server. These three types are variable value changes, event notifiers and calculated aggregate values. A subscription is used to group sources of information together. A monitored item is used to manage a source of information. A piece of information is called a notification. A subscription can contain all three different types of monitored items. Figure 1 illustrates association of session, subscription and monitored item.

Figure 2 illustrates different subscription-related settings. The sampling interval defines the rate the server checks variable values for changes or defines the time the aggregate value gets calculated. The monitoring mode defines whether the monitored item is active or inactive. The queue size defines how many notifications can be queued for delivery. The default value for data changes is one and infinite for events where the size of infinite depends on the resources available in the server. The filter settings are different for data changes, events, and aggregate calculation. The publish interval defines the interval when the server clears the queues and delivers the notifications to the client. The publish enabled setting defines whether the data gets delivered to the client. Services used to actually deliver the notifications in a notification message to the client are the publish service for transferring the

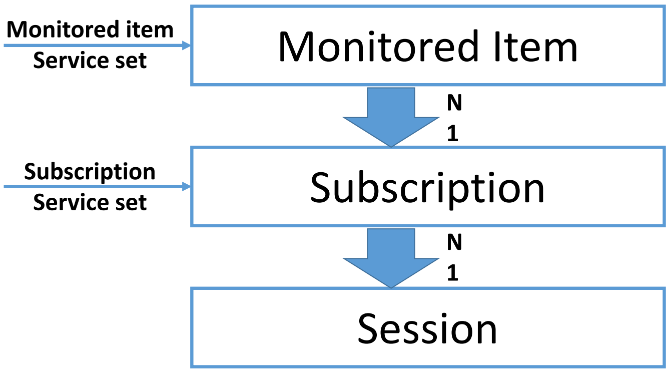


Figure 1: Subscriptions have a set of MonitoredItems assigned to them by the Client. MonitoredItems generate Notifications that are to be reported to the Client by the Subscription (modified from [2]).

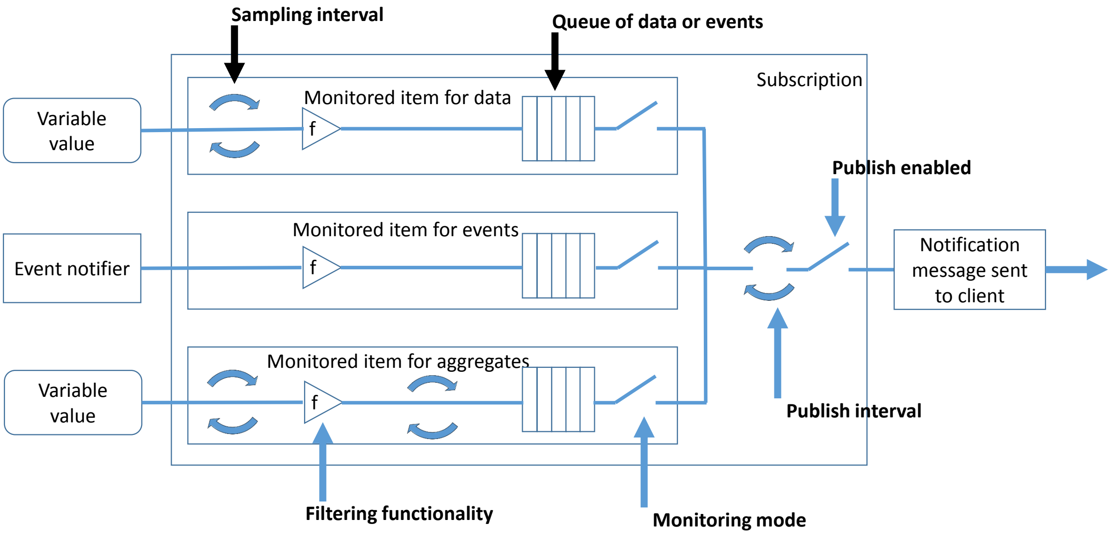


Figure 2: Subscription and monitored item settings (modified from [2]).

notification messages and the republish service to get lost notification messages from the server.

A subscription requires a session to transport the data to the client. The subscription can be transferred to another session, for example to be used in a session created by a redundant backup client if the client that created the subscription is no longer available. Therefore the subscription lifetime is independent of the session lifetime and a subscription has a timeout that gets reset every time data or keep-alive messages get sent to the client. [2]

3.5 Security

OPC UA specifies security capabilities, which make use of standard public key cryptography mechanisms. If HTTPS protocol is used, then TLS security is used to encrypt the traffic already in the Transport Layer. This section will cover the security policies employed with the UA-TCP transport protocol.

Figure 3 illustrates the concepts of the transport layer, secure channel and session. Each OPC UA application has an Application Instance Certificate, which is a standard X.509v3 certificate with some extra fields for additional OPC UA validation. This enables authentication of the applications which may communicate with each other. The respective RSA public and private keys are used to perform a secure hand-shake, when applications create the Secure Channel between them. Both applications will perform the authentication of the other party in the hand-shake, which in practice is an OPC UA OpenSecureChannel service message. During the hand-shake the applications also exchange a symmetric encryption key, which is then used to secure all forthcoming messages through the Secure Channel. When a secure communication policy is selected, Secure Channel ensures the integrity or confidentiality and integrity of the messages that are sent. Once the Secure Channel is in place, the client application will create a Session in the server, over which all other service messages are sent and validated. Session is used to authenticate and authorize users. The session is always bound to a secure channel, which is also renewed frequently. [2]

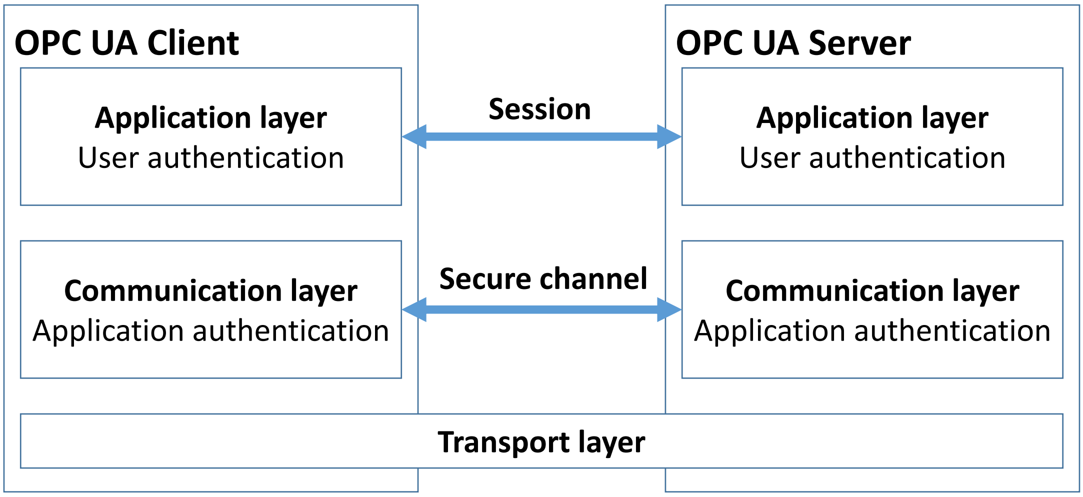


Figure 3: Transport Layer, Secure Channel and Session (modified from[26, pp. 12])

OPC UA enables a flexible selection of the used security policy and security mode between each connection. Specification defines several alternative Security Policies: None, Basic128Rsa15, Basic256 and Basic256Sha256 as of writing this. New policies can be defined and old ones made obsolete as security requirements increase in future. These define the suite of algorithms used to sign or encrypt messages. For example, if the security policy Basic256Sha256 is used, then asymmetric signature algorithm is Rsa_Sha256, symmetric signature algorithm is Hmac_Sha256, asymmetric encryption algorithm Rsa_Oaep and symmetric encryption algorithm Aes256_CBC. Three Message Security Modes are available: None, Sign and SignAndEncrypt. These define the level of security applied to each message. If "None" is selected, then the messages will not be secured. Of course also no overhead is imposed because of signing or encrypting. When "Sign" is chosen then the message is signed. Signing messages allows detecting whether a received message has been manipulated by an untrusted third party. If "SignAndEncrypt" is used, then the message is additionally encrypted. Encrypting messages prevents or at least makes it very difficult for untrusted third parties to read the content of messages. The server administrator may configure which policies and modes are available. Of course, both the client and server need to support the same Security Policy and Mode in order to communicate with each other.

User authentication is performed on the session level. OPC UA defines alternative authentication methods: Anonymous, User name and Password combination, X.509 certificates and also ways to use external user authentication systems, such as Kerberos, via External Tokens. [2]

4 Previous studies of OPC UA performance

Previous studies could be classified in multiple different ways. The first factor is of course what OPC UA service set the test is measuring. Most previous studies concentrate on attribute and subscription service set (see table 5) which is reasonable as these are the services which are used all the time during an OPC UA session. The second factor that might be used to classify previous studies is which transport mappings or security modes (none,sign,sign and encrypt) they use. Some studies start by defining the needed performance levels and then performing measurements [5] [27] as others are just interested in absolute performance. One separating factor is whether the test measures speed indices or utilization indices (see table 1) and which indices are considered. For example speed indices might be either throughput or response time measurements. In this chapter the previous tests are divided into speed indices (effectiveness) and utilization indices (efficiency) as that provides a logical separation of test cases.

4.1 Speed indices

Industrial applications generally feature two different kinds of data exchange. The first is asynchronous, and occurs when a client (e.g. SCADA) requires the read or write access to one or more variables at unforeseeable time instants. The second kind of data exchange is cyclic or periodic and occurs when a client accesses values of variables according to a periodic signals; an example is a client that needs to access a temperature value produced by a sampling algorithm. In the case of asynchronous data exchange, round-trip delay seems a suitable parameter for the performance measurements; considering the read service issued by a client, round-trip delay may be defined as the total response time between the instant at which a request to read one or a set of variables is issued by a client and the instant at which the relevant values are delivered to the client. Considering the periodic data exchange and taking into account a particular variable whose values are periodically produced on the server-side, the delay between the instant at which each value has been produced and the instant at which the client receives that value, seems to be of interest during performance evaluation. In the following, this time interval will be simply called delay. The average delay for each variable gives a measurement of the efficiency of the data exchange. [28] In the subscription case, also the overall throughput is interesting. This is because it is much more important that all values are communicated reliably than it is that values are communicated quickly but some values would be dropped.

Multiple studies of OPC UA communication performance have concentrated on elapsed time tests and in particular the read service. Measured round-trip times without network effect for small message sizes vary between 0.2 milliseconds [2] and 50 milliseconds [29]. When message size grows and real network is used, then round-trip times grow. Tests have been carried out with both scalar values and data blocks. Roundtrip times for scalar values depend on the number of variables per call and the network speed, while data type, such as Boolean, integer or double, is not a significant factor. Roundtrip times for a single variable call depends on the block size and the network speed. For example reading 2.5 megabyte data block from an OPC UA server takes about 200 milliseconds [7]. The network that is used has an effect on the absolute performance. When read calls on localhost are in the range of average 1.4 milliseconds, same read calls can be almost as fast (1.5 milliseconds) when executed in fast LAN network. However same read calls over rural 3G network takes on average 96 milliseconds. Ping time to some common servers in this same network is in the range of 30 - 40 milliseconds. [5]

The read service call uses request-response pattern, and delays are present in both steps. In contrast, notification messages are transferred without a specific request, making them more responsive. For this reason, the latency of subscriptions can be assumed to be smaller than periodic reading of variable values discussed above. Sometimes it is more important to achieve the biggest throughput possible rather than concentrate on fast responses [30]. However, previous studies do not seem to present comprehensive throughput results. Fojcik and Folkert [12] present result of 250 kB/s throughput. However, that result is not the maximum possible throughput as the test case is defined to test what is the average network traffic generated with certain use case.

Signing and encrypting data has been of interest in many studies. Signing and encryption adds overhead to the communication [31]. The conventional wisdom is that signing and encrypting data is a time demanding task and potentially unusable in resource limited devices. The previous results show somewhat ambiguous results: some studies report that encrypting messages seems to increase round-trip time by a factor of ten, whereas signing seems to have little or no effect [32]. On the other hand, some studies show that with small message sizes encryption doubles the amount of round-trip delay and signing and encryption operations take almost identical times [33]. These studies also differ so that the first one reads variable with static size 102400 bytes whereas the second one shows results with multiple variable sizes and concludes that the impact of signing and encrypting gets smaller as message sizes increase. These two results are illustrated in figures 4 and 5. In figure 5 the results are presented normalizing the values of the round-trip time to those achieved considering scenario no security and UA TCP. This study presents different results for security modes "no security" and "secure channel with security mode none". It is not completely clear what the difference between these is. The conclusion is however that there is a clear difference between these results.

In addition to the attribute and subscription service sets, the session service set could also be evaluated as an important part during an OPC UA session. In some use cases where the goal is to gather data from multiple sources, the connect and disconnect times may become important. Results obtained with OMNeT++ simulator show that creating an OPC UA session takes from 0.054 seconds to 14 seconds depending on what kind of CA verification is used. [33]

In conclusion, OPC UA communication effectiveness depends on the way that the data in the OPC UA server address space is updated, the network speed, number of nodes per call and data block size. Here, updating the server address space is considered to include possible subscription settings such as sampling interval and publish interval in case of subscription. To some extent the selected security mode

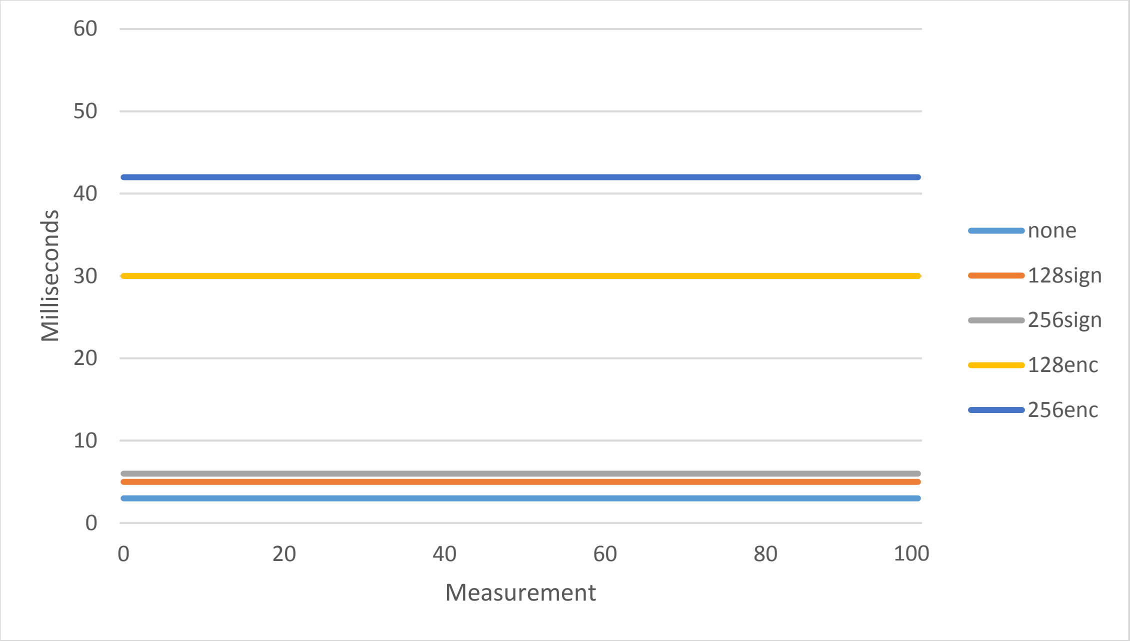


Figure 4: Signing and encrypting overhead according to Post, Seppälä, Koivisto (modified from [32]).

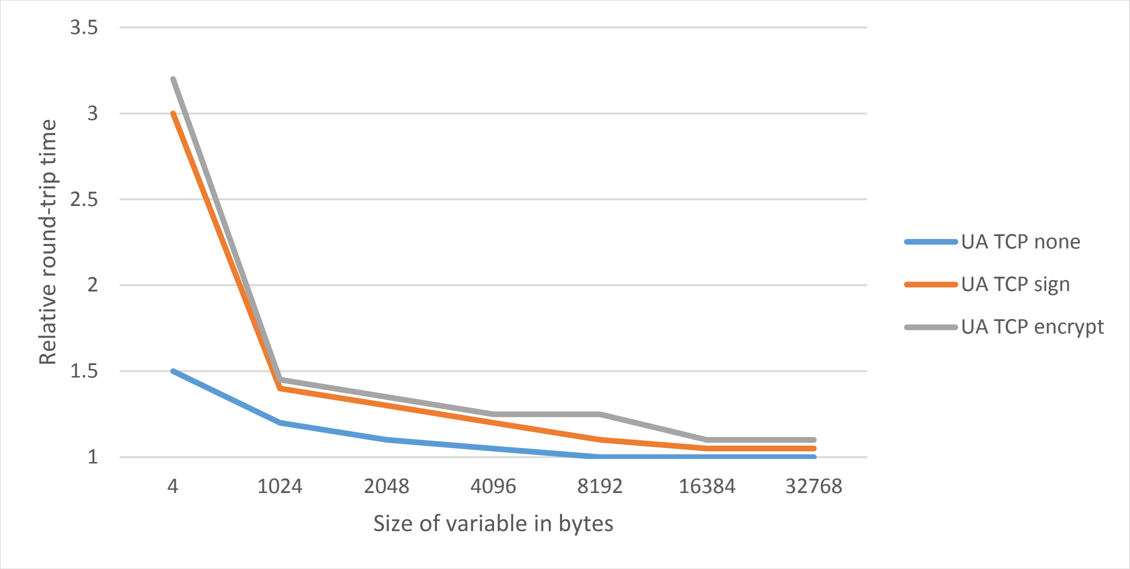


Figure 5: Signing and encrypting overhead according to Cavalieri et al. (modified from [33]).

(whether or not to use signing and encrypting) has an effect. The programming language and operating system may have an impact on performance, but the impact is probably much smaller than with the before mentioned features. Different data types do not influence read and write performance.

4.2 Utilization indices

Performance does not only mean the speed of communication, but also the load and resource requirements imposed on the target system by the application. In theory, server load should depend on the number of clients and amount of data to process.

Processor load in an application that produces constant workload has been investigated in a case study from year 2009 [7]. The hardware used in this specific test is Intel Atom Z530 and two different profiles were used. The embedded UA server profile consisted of 500 variables across 5 subscriptions. The standard UA server profile in turn included 37500 variables across 75 subscriptions. When the embedded profile was sampled at 100 milliseconds sampling rate, the processor load was 2%. When the standard server profile and sampling rate of 1000 milliseconds were used, the processor load rose to 20%. When sampling was done with the same profile but with sampling rate of 500 ms, the processor load was 40%. According to this, sampling had a huge linear effect on the processor load, as doubling the amount of sampling also doubled the processor load. The amount of subscriptions and handled variables of course also has an effect on the processor load.

It is possible that a large amount of monitored items can inhibit OPC UA server from working correctly [34]. A study by Fojcik and Folker from year 2012 suggests that even less than 1000 monitored items may prevent acceptable performance level [12]. Figure 6 illustrates this result. It is interesting that the reported level is so low as the test uses PC-class computers with quad-code processors and 4 gigabytes RAM memory. This finding is contrary to previous where 37500 monitored items caused CPU load of 20%. Of course, there is difference in the OPC UA software stack used and a different hardware than the previously mentioned. The same study also illustrates how server application architecture can make a huge difference. Two versions of server applications were tested. In the first case the server was updating tag's values directly after acquiring new data from data source and in the second case there were two separate threads: one for gathering data from the data source in dedicated queue and second only for updating associated monitored items. In the first case, where tags were updated without buffering, the CPU load for 20 clients was on the level of 60% and the CPU was not able to handle all of the processes. In the second case the CPU load for 50 clients was on the level of 15%.

In conclusion, resource requirements imposed on the target system by an OPC UA application are more application specific than communication effectiveness discussed in previous section. There are less publications regarding hardware utilization than there are publications measuring communication speed. A typical thing is that round-trip values are presented in the case of read service tests and resource usage is reported in the case of subscription testing. This is logical because in a normal application scenario the important thing about subscriptions is that all value

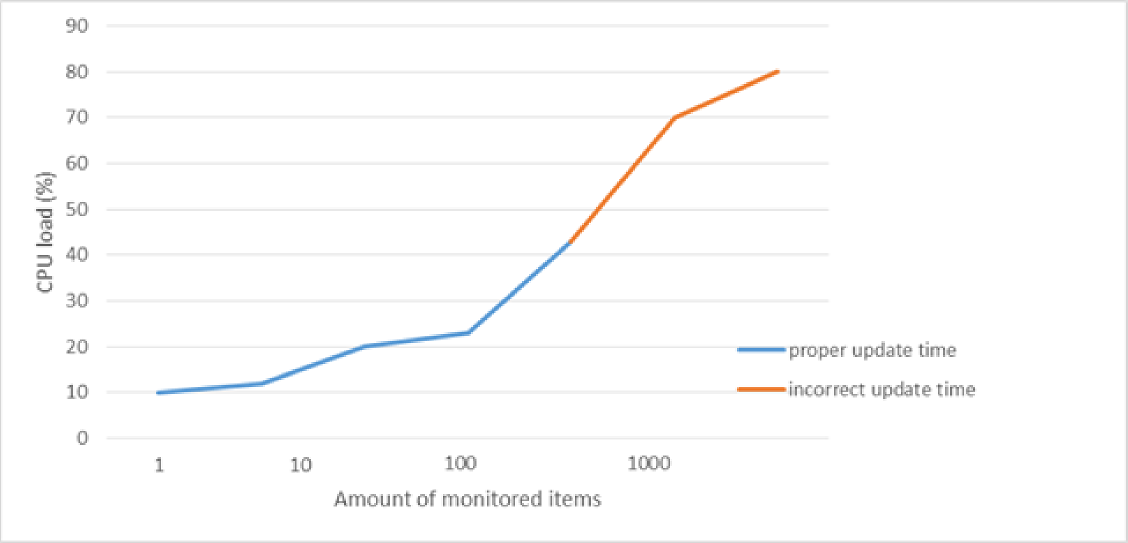


Figure 6: CPU load as the function of monitored items according to Fojcik and Folkert (modified from [12]).

changes are reported reliably and the speed of actual data communication is not that important. The resource usage of other services besides subscriptions was not found in previous studies.

4.3 Protocol comparison

OPC UA does not exist in a vacuum. Knowing how OPC UA implementations compare against competitors is valuable information. The older OPC standard is still widely deployed in legacy applications and in some sense could be considered as a competitor to OPC UA. Thus, the first comparison is naturally between OPC classic and OPC UA [2]. As said before, OPC UA supports multiple transport protocols. Differences between these transport protocols inside OPC UA specification are of course interesting. Lastly, differences between completely different protocols like OPC UA and web service technologies would be interesting.

Historically, the COM and DCOM technologies used in OPC Classic were high performance binary protocols. One of the requirements for OPC UA was to maintain or even enhance the performance of Classic OPC [2]. Tests comparing these two specifications confirm the hypothesis that OPC DA and OPC UA data transferring capabilities are roughly the same [2]. A study made at Cern reports update times as function of subscribed elements [35]. The time that it takes for the whole process is rather similar with a low number of items, but as the number of items increases in the server's address space, the OPC UA Client/Server remains having a very low update time (similar as having a low number of items), while OPC DA Client/Server has an increasingly higher update time. For 5000 elements the update times are 1000 milliseconds for UA application and 2000 milliseconds for OPC DA application. For 30000 elements, UA application still updates at a rate of 1000 milliseconds whereas OPC DA application is taking almost ten times as long (10000 milliseconds). Thus, it appears that systems based on the OPC UA scale better in terms of address space size than analogous systems based on OPC-DA. Using UA TCP and binary encoding, OPC UA is able to keep the performance of Classic OPC, while at the same time providing new features, such as security of the communication [36, pp. 179].

Historically XML encoding has been seen as resource intensive [2, pp. 308]. The OPC UA book presents results of measuring round-trip times with different transport protocols and data encodings. The numbers indicate that there is only a small overhead for using SOAP/HTTP protocol with binary encoding instead of the UA TCP protocol. There is a much bigger overhead when using SOAP/HTTP protocol with XML encoding instead of the UA TCP protocol. SOAP/HTTP protocol with XML encoding is 1.8 times slower for small messages and 18 times slower for large messages [2]. Some publications even deem the XML format prohibiting high performance real-time systems [37]. However, XML or other structured data encodings have also positive sides such as human-understandable format and flexibility. Usually using structured data such as XML messages results in much bigger message sizes, 16 to 25 times larger than a conventional binary message [13]. This illustrates the overhead that is associated with using a structured versus a custom data format. However, Efficient XML Interchange (EXI) data encoding of XML messages may increase the efficiency of structured data formats. In an example case, message size is 10 kB for UA Binary, 270 kB for XML and 15 kB for EXI encoded XML message [38]. According to this example, the message size for UA Binary coding (3.75% of the original XML message size) is nearly the same as for the default EXI coding (5.15% of the original XML message size). However, this study does not report results of round-trip times or resource usage when using EXI encoded XML-messages and thus it is not clear that the smaller message size leads to better communication performance automatically.

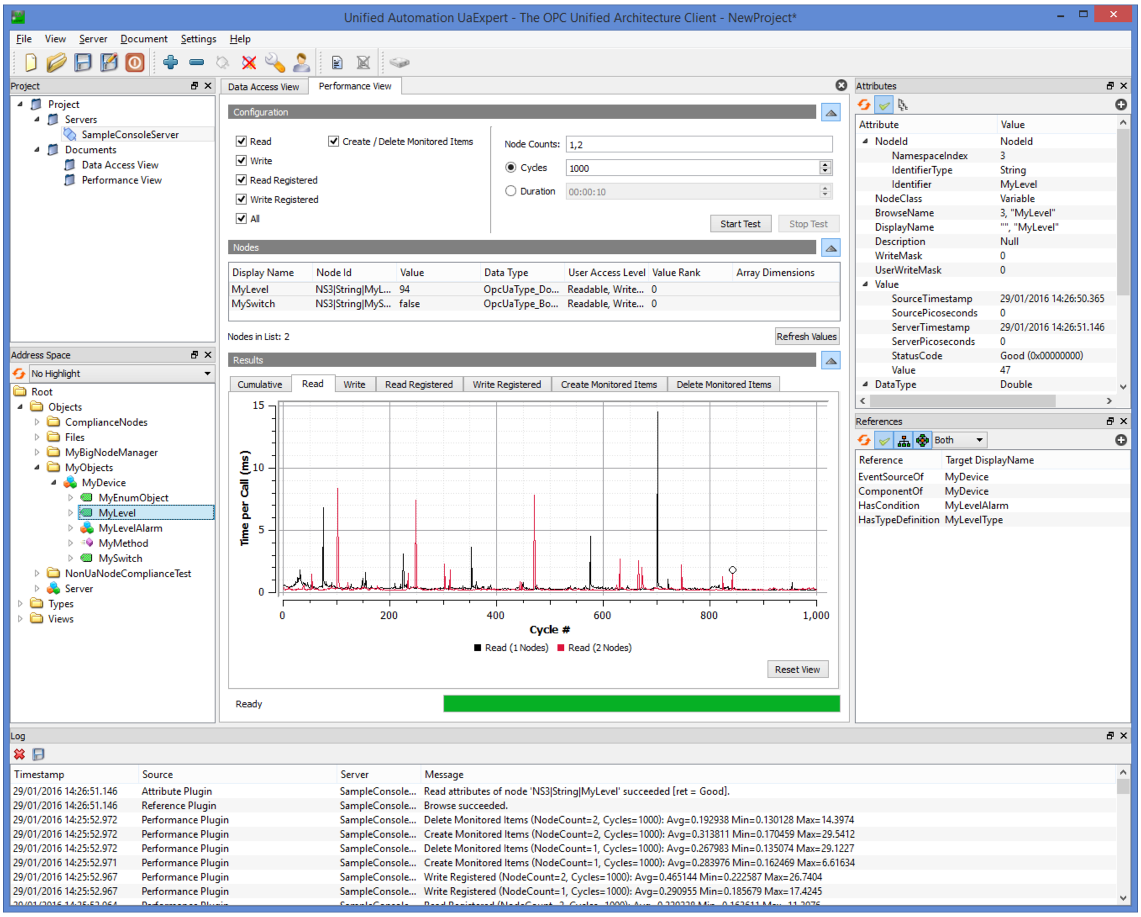
Some protocol comparisons are more of a qualitative kind. One interesting study from year 2011 compares multiple IP-based protocols in terms of suitability for automation gateway role. SNMP, LDAP, SQL, Web servers, OPC XML-DA, OPC UA and a proprietary protocol are compared. [1] One interesting conclusion is that web servers presenting HTML are well suited for presenting fieldbus and gateway data to a human operator but introduces overheads with periodic data exchange. Using SQL as gateway is also an interesting premise. The study criticizes support for generic clients and support for different data types when using SQL.

In conclusion, it can be stated that binary data encoding should be more efficient than XML or other structured formatting. OPC UA binary and classic OPC should exhibit same kind of performance. OPC UA standard has been developed to use cases in automation gateway role and contains features needed in that application area.

4.4 Tools

Unified Automation's UaExpert software is shipped with a Performance View feature. This performance view can be used to measure the performance of certain UA service calls to a UA server. It can be configured to call a service for a defined number of times or to call a service as often as possible in a specified time span. Figure 7 illustrates the user interface of this performance view feature. Current version of UaExpert as of writing this is 1.4.1. The supported services of the tool are read, write, read registered, write registered, create monitored items and delete monitored items. The tool can be used to measure round trip times of these service calls or the amount of service calls made in a specified period of time. In round-trip mode, minimum, average and maximum round-trip times are reported and in addition to these, all individual round-trip times are available. In duration mode, the total number of calls and the average time per call are reported. The amount of service calls per time period is done with synchronous calls from client application. This means that for example new read call is made as soon as previous read call returns. This approach is different from the approach taken for example in the httperf application presented before where the aim is to saturate server with asynchronous service calls.

In duration mode, tests produced by this tool are perfect examples of the throughput test as defined in the chapter 2.1. In the round-trip mode the definition of these tests is a bit more unclear. In principle, the test case sounds like a response time test, because what is measured is a response time. However, there is no concept of think time in these tests, because next request is sent when previous is gotten from the server. In other words; throughput demanded by the client is not fixed. Still, the amount of work is fixed, because the amount of service requests is decided before starting the test. This leads to classifying this test case as batch measurement. Actually, the amount of service calls can be thought of as a "batch", which explains the naming quite well. This tool is not designed to be used with third type of tests, response time tests. In response time tests, some think time between requests would be introduced.



☑ Read
☑ Write
☑ Read Registered
☑ Write Registered
☑ All
☑ Create / Delete Monitored Items

Node Counts: 1,2
● Cycles 1000
○ Duration 00:00:10

Display Name	Node Id	Value	Data Type	User Access Level	Value Rank	Array Dimensions
MyLevel	NS3|String|MyL...	94	OpcUaType_Do...	Readable, Write...	0	
MySwitch	NS3|String|MyS...	false	OpcUaType_Bo...	Readable, Write...	0	

Nodes in List: 2

Ready

Attribute	Value
NodeId	NodeId
NamespaceIndex	3
IdentifierType	String
Identifier	MyLevel
NodeClass	Variable
BrowseName	3, "MyLevel"
DisplayName	"", "MyLevel"
Description	Null
WriteMask	0
UserWriteMask	0
Value	
SourceTimestamp	29/01/2016 14:26:50.365
SourcePicoseconds	0
ServerTimestamp	29/01/2016 14:26:51.146
ServerPicoseconds	0
StatusCode	Good (0x00000000)
Value	47
DataType	Double

Reference	Target DisplayName
EventSourceOf	MyDevice
ComponentOf	MyDevice
HasCondition	MyLevelAlarm
HasTypeDefinition	MyLevelType

```
Timestamp                Source              Server            Message
29/01/2016 14:26:51.146  Attribute Plugin    SampleConsole...  Read attributes of node 'NS3|String|MyLevel' succeeded [ret = Good].
29/01/2016 14:26:51.146  Reference Plugin    SampleConsole...  Browse succeeded.
29/01/2016 14:25:52.972  Performance Plugin  SampleConsole...  Delete Monitored Items (NodeCount=2, Cycles=1000): Avg=0.192938 Min=0.130128 Max=14.3974
29/01/2016 14:25:52.972  Performance Plugin  SampleConsole...  Create Monitored Items (NodeCount=2, Cycles=1000): Avg=0.313811 Min=0.170459 Max=29.5412
29/01/2016 14:25:52.972  Performance Plugin  SampleConsole...  Delete Monitored Items (NodeCount=1, Cycles=1000): Avg=0.267983 Min=0.135074 Max=29.1227
29/01/2016 14:25:52.971  Performance Plugin  SampleConsole...  Create Monitored Items (NodeCount=1, Cycles=1000): Avg=0.283976 Min=0.162469 Max=6.61634
29/01/2016 14:25:52.967  Performance Plugin  SampleConsole...  Write Registered (NodeCount=2, Cycles=1000): Avg=0.465144 Min=0.222587 Max=26.7404
29/01/2016 14:25:52.967  Performance Plugin  SampleConsole...  Write Registered (NodeCount=1, Cycles=1000): Avg=0.290955 Min=0.185679 Max=17.4245
```

Figure 7: UaExpert Performance View

5 Test requirements

It would not be feasible to discuss all the possible aspects of OPC UA performance more deeply in this thesis. In this section OPC UA functionality is assessed based on previous studies. Usually users are in the first place interested in data transferring capabilities of OPC UA applications [39]. Performance and possible limitations of OPC UA subscription model are interesting, especially because the subscription model has previously been the limiting factor of performance in some cases [34]. When analyzing the data transferring capabilities of an OPC UA application it would be obvious to expect that the network imposes restrictions on data throughput. However, previous research has concluded that indeed this is not the case with fast LAN network but actually the internal architecture of the server and utilization of hardware resources can be a bottleneck [12]. OPC UA supports multiple technology mappings. Using HTTPS as the transport or XML as the encoding is not that popular. In the following tests only UA TCP and UA binary encoding are considered. Based on previous studies, three main research topics can be formulated:

- Effect of signing and encryption
- Overloading the server with asynchronous reads
- Subscription performance testing

To provide some logical separation, the selected test cases are logically divided into elapsed time (batch) measurements, throughput measurements and response time measurements. Based on previous studies, executing response time tests is the most interesting part, since that kind of studies have not been done previously with OPC UA, whereas with other information systems response time testing is usual. In response time tests, workload and throughput are constant.

For the elapsed time tests, the server application needed to contain variables of selected size. In these tests, client and server application resided on the same host. Round-trip times without network effect were measured as well as relative effect of signing and encrypting. Synchronic (blocking) read calls were used. Only the round-trip time was measured and in this case the client functionality was really straightforward. For the response time tests, a custom client application was implemented and the idea was that the response time client could be used with any OPC UA server. The client functionality as well as collected metrics were more complicated. When running the response time tests, client and server applications were always on different host machines. In response time tests the main idea was to measure the response times until the server application becomes saturated with more requests than it can handle.

5.1 Elapsed time (batch) measurements

In these tests, the workload is pre-determined and the interesting thing to measure is how fast the given workload is executed.

5.1.1 Impact of signing and encrypting

As previously mentioned, signing and encrypting data has been studied many times previously but with somewhat ambiguous results. This is one aspect of interest where more results and documentation is needed. Besides the test results documenting round-trip times and resource usage, it is important to provide detailed information of the software and hardware platform used in this test.

5.2 Response time measurements

Based on previous publications, read and subscription services are considered the most interesting ones for response time measurement testing. It is important to ensure that the client side application is not the limiting factor during these tests. If however the client side is the limiting factor, then documenting the resulting resource usage in the given hardware and software platform is essential. In response time tests, a suitable think time should be introduced between requests and this leads the throughput to be fixed unless the server application is overloaded with more requests than it can handle.

5.2.1 Read mechanism

The read service's request and response format are pretty straightforward. One thing to note is that every read request in OPC UA can contain a list of nodes and their attributes to read [24, pp. 48]. This means that single read service call may request attribute values from multiple nodes and one read response may contain multiple attribute values for multiple nodes. This list size is referred to as "batch size" in the following text. In order to determine how OPC UA servers react to growing number of read requests, the following measurements should be made.

1) Read one value with growing frequency
2) Grow batch size, while keeping the amount of sessions and frequency constant. The session amount should be one and frequency should be decided based on previous test case.
3) Grow the amount of sessions while keeping other parameters constant. Batch size and frequency of reads should be decided based on results of previous test case.

Response times (arithmetic mean, maximum, 90th percentile time) as well as amount of responses per second are recorded in these test cases in order to determine the achieved response times and throughput. These tests can be classified as response time tests measuring speed indices. Resource usage on server and client shall be monitored manually. All read requests in this test should be made asynchronously to allow client to keep sending messages even if the server has not responded to all previous requests. This approach makes it possible to overload the server.

5.2.2 Subscription mechanism

The purpose of this test case is to grow the number of sessions, subscriptions and monitored items and measure the delays as well as achieved subscription throughput. This test should be run with enough client machines to ensure that the client side application is not the limiting factor. The intended usage of subscriptions is that one subscription contains multiple monitored items. Usually, one session has 1-10 subscriptions, each of which may contain even thousands of monitored items. One important detail is that the monitored items should be added in one complete set instead of repetitive requests [24, pp. 62]. In order to determine the available subscription mechanism performance, the following measurements should be made.

1) Test response times and throughput with one session, one subscription and variable amount of monitored items.

2) Test response times and throughput with one session, variable amount of subscriptions and constant amount of monitored items. The constant amount is decided based on the previous test.

3) Test response times and throughput with variable amount of sessions and constant amount of subscriptions and monitored items. The constant amount is again decided based on the previous test.

6 Research material and methods

Because of the client-server architecture, where the server is passive and only responds to requests made by a client, it is possible to make a generic test client application which is usable with any server application.

6.1 Design of response time test client

In order to stress the server side application, a custom client application was developed. This makes it easier to reproduce tests and potentially enables people to run tests with their own applications if the test client is published. Sample client application provided with the Prosys OPC UA Java SDK (from now on referred to also as the SDK) was used as a base for this test application. The most important aspects were the read and subscription testing and test functionality for these services was developed with highest priority. As the output of this process, a command line application named UaPerfClient was made. An example of command line options to start the application is shown here:

```
−n "ns=2;s=MyLevel" −clients 1 −batchsize 100 −rate 100 −
    requests 9000 −duration 300 −percentile 0.9 opc.tcp
    ://10.50.100.156:52520/OPCUA/SampleConsoleServer
```

This example connects to the specified OPC UA server residing on a machine with ip address 10.50.100.156 and starts reading the value attribute of node "ns=2;s=MyLevel" with a rate of 100 read cycles per second. In this example, one read cycle consists of one session (-clients 1) sending one read request to the server. The 'batchsize' parameter defines the size of the nodesToRead list in a read request. In this example 100 values are read in every request. Parameter 'requests' defines the total amount of read requests to make. With these settings, the test will run for 90 seconds when everything goes normally, because 100 requests per second * 90 seconds = 9000 requests. If the server cannot keep up with the requests that the client supplies, the test run will take a longer time. After 'duration' amount of seconds the client execution is stopped in any case. After the test is finished, the UaPerfClient will print statistical information about the run. In addition to the parameters introduced here, table 8 lists all the command line parameters of the UaPerfClient application.

6.1.1 Read testing

In OPC UA, the read requests are always performed asynchronously. Ordinarily, the OPC UA SDKs offer both asynchronous and synchronous read functions. The synchronous reads (or blocking reads) explicitly wait until a response arrives from the server. This provides convenience for the client application developer. The Prosys OPC UA Java SDK also supports reading with synchronous or asynchronous methods. As stated before, the client side application should be able to overload the server side application with more requests than it can handle. For this purpose, the obvious choice was to use the asynchronous read methods.

Table 8: Command line parameters of UaPerfClient application.

Parameter	Declaration
clients	number of client sessions to establish.
batchsize	defines the number of ReadValueIds in nodesToRead list in batch mode and number of read requests per client connection in single mode.
duration	the maximum time to wait for asynchronous responses or the duration of subscription test.
subscriptions	number of subscriptions to establish per client connection.
items	number of monitored items established per client connection.
single	only request one value per read request. Default operation is not to use this mode.
rate	define the rate (requests/second) at which read requests are issued during read test.
requests	amount of read requests. This defines the duration of read test.
percentile	Which percentile value of round-trip or delay times is calculated. Default 0.9

Periodic reads from the client application was developed by defining a Runnable object which handled the asynchronous read calls and scheduling this Runnable with ScheduledExecutorService. ScheduledExecutorService is a fixed-size thread pool that supports delayed and periodic task execution, similar to Timer [14, pp. 120]. Scheduled thread pools allow multiple threads for executing periodic tasks. ScheduledThreadPoolExecutor also deals properly with ill-behaved tasks and continues the execution even if some individual tasks ended up in error. At the end of the executions, statistical information including information about round-trip times per single request and total throughput statistics will be printed. Parameter percentile can be used to print certain percentile value of all arrived requests. The default option is to print 90th percentile response time of all request-response round-trip times. An example of statistics printed out by the program is shown below.

```
Number of read requests: 9000
Number of read responses: 9000
Total duration: 89998.97914699999 milliseconds
Average response time: 1.745715
Max response time: 10.323435
0.9 percentile response time: 1.845754
Number of results: 900000
Results per second: 10000.11342939772
```

In the default mode, the Prosys OPC UA Java SDK contains functionality to monitor server status. This server status monitoring is done by reading the server status as any other data. With large amounts of read requests, the client may falsely think that the server is not responding because of long wait times before the server responds to status reads. Clients can set an interval for this status check as well as a timeout value. By adjusting the timeout, the user can configure for how long the client continues to operate without knowing what the status of the server is. In the client application, this status check timeout is set in the following manner.

```
client.setStatusCheckTimeout(<suitable value>);
```

However, during testing there is really no point to monitor the status at all. Either the client stays connected to the server or, if not, the test run can be repeated. Also, the server status reading actually interferes with the test run because it also involves issuing read requests. Because of this, it is better to disable the status check altogether by setting the status check interval to zero.

```
client.setStatusCheckInterval(0);
```

6.1.2 Subscription testing

The main architecture of UaPerfClient with subscriptions is that sessions, subscriptions and monitored items are initialized. After that,a client side wait is introduced before measurement phase begins. During measurement, the number of publish requests, publish responses, monitored items, test duration, delays and subscriptionIDs in publish responses are tracked. After the time specified by the parameter 'duration' has passed, logging of publish requests, responses and monitored items is stopped and subscriptions are unsubscribed. The intention is to ensure that only the time when all subscriptions are fully initialized is taken into account in the measurement. An example of statistics printed out by the program is shown below.

```
Number Of Publish Requests: 90
Number Of Publish Responses: 90
Number of Monitored Items : 9000
Duration: 18.018479137 seconds
Monitored items per second: 499.4872170714438
Average delay time (ms): 7
Max delay time (ms): 15
0.9 percentile delay time (ms): 13
Subscription min freq: 18
Subscription max freq: 18
```

Command line options to obtain this result were

```
-n "ns=2;s=MyLevel" -clients 1 -subscriptions 5 -items 100 -
    duration 18 opc.tcp://10.50.100.156:52520/OPCUA/
    SampleConsoleServer
```

In the case of read requests, round-trip time was a parameter of interest. In the case of subscriptions, there is no similar concept of round-trip time. Instead, the time when new value is obtained by the server and the time when client application gets notified can be monitored. This is called just 'delay' in this text. As in read requests, in delay measurement, the average, maximum and percentile values were obtained. The delay could be measured through number of ways: 1) based on the timestamp field of response header in publish responses, 2) from the publish time field of notification message (there is one notification message per publish response) or 3) from individual data value's server timestamp. In practice, the difference in all these approaches is probably not that big but the most correct way would be to use the timestamp of each individual data value. The specification states that the server timestamp is used to reflect the time that the server received a variable value or knew it to be accurate [24, pp. 121]. During testing, it was however noticed that calculating the time difference from every received value update consumes surprisingly much resources and client side easily becomes a bottleneck because of this. The next best thing in delay measurement is the publish time field of notification message, which OPC UA specification defines as the time that the message was sent to the client [24, pp. 136]. This was deemed very suitable for these measurements as there is considerably less notification messages than individual data values.

The tests were made with server and client applications residing on different hosts. In read service testing the round-trip time is measured with the client system clock. In subscription delay measurement however, server machine sets the publish time, and client application calculates delay from this time. Differences in these clocks are reflected in the measured delay. The first delay time measurements pointed out that internal clocks actually drift a considerable amount if no clock synchronization protocol is used. Adjusting clocks before every test would be daunting and would also introduce huge differences in measured delay values. Because of this, it is important to use clock synchronization protocol. Despite the use of clock synchronization protocol, some difference of clocks was visible between different client hosts. Two client hosts were used, and immediately after synchronizing clocks, the other one reported average delay time of 7 milliseconds. In similar test run on the other host, the delay was in the range of 60 milliseconds. In absolute values, this may seem like a small difference, but it is still an increase of almost ten times. The delay presented in the subscription tests should be anyway thought of in relative terms. This means that the most interesting thing in practice is to see how the delay time grows when more data notifications are handled.

With subscriptions, an important aspect of testing was to try to evaluate whether or not all notification messages actually arrive at the client application. This is very important in the case of subscription mechanism, where one missing data notification could mean that the client never gets the newest value of some variable. Missing some value update altogether might potentially be very harmful. Delivering all notifications reliably is more important than delay of notifications. In OPC UA, one publish response always contains data notifications for only one subscription. In UaPerfClient, incoming data notifications are counted and in addition to this, the publish responses per subscription are monitored. This is by no means a perfect way of telling if all notification messages arrived as they should have but it does not consume much resources and it still provides some information about how fairly subscriptions have been treated. Measuring all individual notification messages and analysing that they arrived in order would be better, but when designing the system it was deemed too resource intensive because it is possible that high amount of sessions, subscriptions and monitored items may be measured. As in the case of delay measurement, the client side easily becomes the bottleneck and thus resource usage in the client side was kept to a minimum.

The intention is to ensure that only the time when all subscriptions are fully initialized is taken into account in the measurement. To make sure that all sessions, subscriptions and monitored items have been initialized correctly, a wait was introduced before measurement phase begins. This wait was decided to be 60 seconds as that seemed long enough without making test cases unnecessarily long.

In subscription testing, it is of course necessary that the tested server application has some node with deterministic value changes. In the Prosys OPC UA Java SDK SampleConsoleServer the sample node "ns=2;s=MyLevel" provides this service as its value is changing once every second. Correspondingly, in the UA Demo Server node "ns=4;s=Counter1" changes at intervals. This value changing could perhaps also be done by another client program continuously writing values to a selected server node. This approach would offer more flexibility as basically any server could be tested with any data change interval desired. However, it would also mean that not only subscription service performance would be tested because the server would be handling both the write calls and subscriptions at the same time. This is why this approach was not studied further during this thesis.

In Prosys OPC UA Java SDK, the UaClient class represents a client connection interface to an OPC UA server. The most straightforward way to add custom functionality to the UaClient is to define a custom UaClientListener. SDK sample implementation MyUaClientListener was extended with a class called UaPerfClientListener. The class UaPerfClientListener keeps track of publish requests, publish responses and data change notifications. One instance of this class is shared by multiple UaClients. The variables were defined as AtomicIntegers and updated via incrementAndGet() methods to avoid multithreading issues.

Compared to reading and writing variables, the subscription mechanism is more complicated. There are multiple parameters that affect the communication. The most important ones of these communication parameters are listed here:

- Client side subscription settings:
    - **NotificationBufferSize** defines the maximum number of NotificationData packets to keep in the client internal buffer. The buffer is used to handle the incoming notification packets in the client. Setting the value too low causes the client to ask for republish frequently.
    - **MaxNotificationsPerPublish** is the maximum number of notifications that the client wishes to receive in a single publish response.
    - **MaxMonitoredItemsPerCall** defines the maximum number of items in one service call.
- Client settable server side MonitoredItem parameters
    - **SamplingInterval** is the interval that defines the fastest rate at which the MonitoredItem(s) should be accessed and evaluated. This interval is

  defined in milliseconds.
  - **QueueSize** is the requested size of the MonitoredItem queue.

- Server side subscription settings:
  - **RetransmissionQueueSize** defines the amount of messages to keep stored for retransmission. This effects how easily subscription starts to report "Message not available" errors. Clients cannot set this variable. The default value is 10. Specification part 4 states that the session shall maintain a retransmission queue size of at least two times the number of publish requests per session the server supports [24, pp. 69].

A couple of the most influential parameters to subscription performance were detected when instantiating a large amount of subscriptions. The first case was that about 10000 monitored items can be added in one request with normal settings. This happens because the message just grows to be such a big one. When trying to subscribe to more monitored items the following error is shown.

```
com.prosysopc.ua.ServiceException: Bad_EncodingLimitsExceeded
com.prosysopc.ua.ServiceException: ServiceFault: Bad_TooManyOperations
```

Monitored items must be thus added in smaller batches. The Prosys OPC UA Java SDK contains functionality of OperationLimits to work with this situation. When creating subscription, the subscription can be limited to add a maximum of 10000 monitored items per time.

```
subscription.setMaxMonitoredItemsPerCall(10000);
```

Another case was that with about 100000 monitored items, the client started reporting missing notification messages from republish.

```
com.prosysopc.ua.client.Subscription - Requesting missing NotificationMessage
    (SequenceNumber=261) from the server using Republish: Target
    SequenceNumber=265
com.prosysopc.ua.client.Subscription - Message not available
```

This message was shown because the client side notification buffer was too small for this particular test case. The client had received the mentioned message and acknowledged it. After this, another thread is responsible for actually handling the publish response. If notification buffer overflows, then messages are lost in the client side application before they can be handled. Setting client side notification buffer size to larger value should help in this situation.

```
subscription.setNotificationBufferSize(5000);
```

7 Results

7.1 Elapsed time results

Modern PC hardware usually contains hardware-accelerated encryption. However, not all devices support such features. The purpose of this test case was to find out what kind of performance could be expected from different hardware platforms. We compared the following computers:

- Dell laptop, OS Windows 8.1 64-bit, Intel Core i7 @2.70GHz, 8 GB RAM memory and SSD drive.
- Raspberry Pi, OS Raspbian Linux, 700 MHz single-core ARM, 512 MB RAM memory and SD card as a storage.

The Dell laptop contains hardware-accelerated encryption whereas the Raspberry Pi does not have this feature. The average results of profiling the time used to encrypt a single OPC UA chunk at a given platform are shown in table 9. The most interesting thing in practice is not the absolute time but the relative difference between the devices. We see that on the Raspberry Pi platform the encryption takes approximately 20 times more time than on normal modern PC hardware.

Table 9: Average time to encrypt a single request value.

Platform	Milliseconds
Laptop	0.19
Raspberry Pi	3.87

There is considerable difference in these measurements which is also just what was expected. The User of OPC UA SDK is however not interested in the time taken to encrypt one value but more in the round-trip time of service calls. It is again expected that adding signing and encryption will increase the round-trip time. The interesting thing is how much overhead the security modes will induce. Figures 8 and 9 show the time required to make a complete read service call, including the request and response messages. The read consists of a single variable of byte array with 10 elements.

The measurements were made with both the server and the client on the same machine using the sample applications included in Prosys OPC UA Java SDK. The communication protocol was UA Binary and the security policy was Basic128RSA15. The times were recorded by profiling synchronic read service calls and averaging it for 1000 calls. As a conclusion we can see that encryption adds overhead to the communication as expected. Round trips with encryption take 1.5 – 2 times as long as without encryption. However, in most application areas this could still be deemed negligible because the absolute time is in the range of 0.5 ms to 10 ms, only. Figures 10 and 11 illustrate the same measurements but now with a byte array variable containing 10 000 elements. It can be seen that when the amount of transferred data grows, also the overhead of encryption seems to grow. In the case of

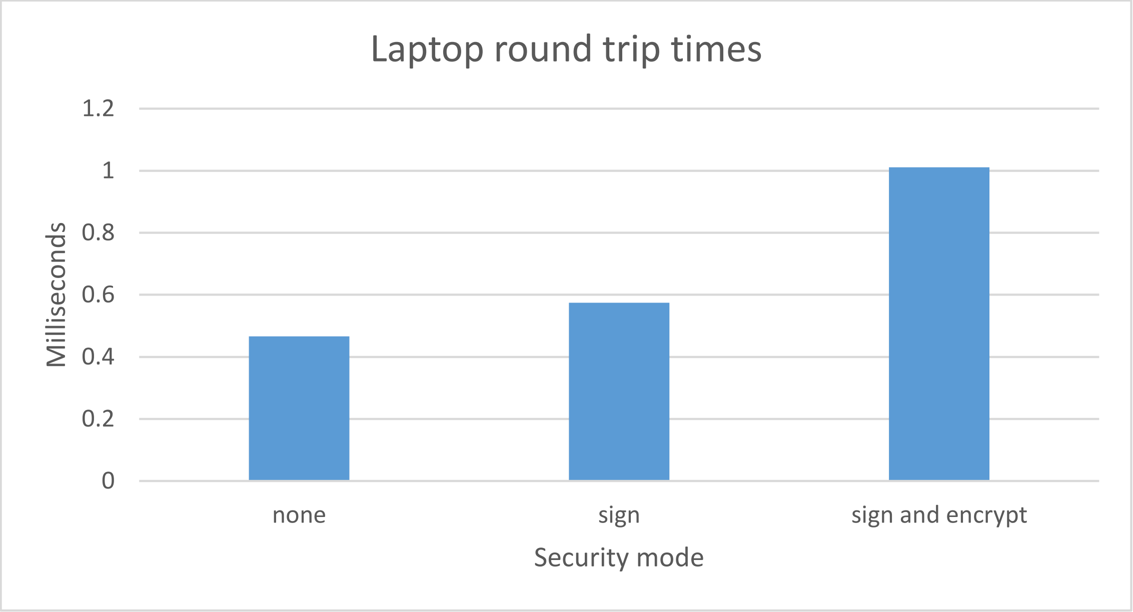


Figure 8: Average round-trip time with laptop and byte array of length 10.

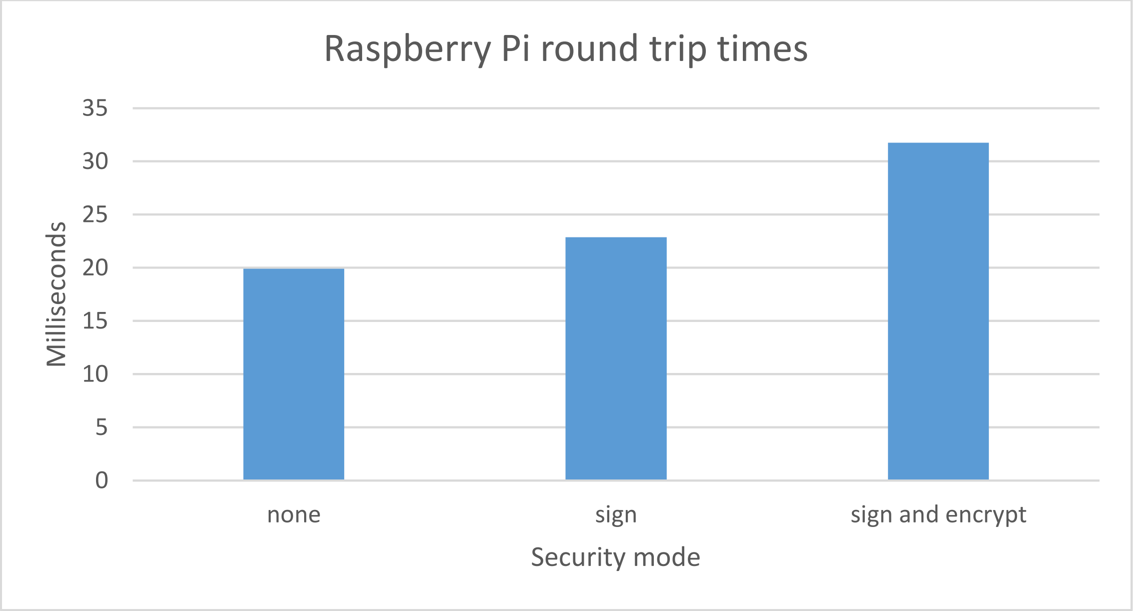


Figure 9: Average round-trip time with Raspberry Pi and byte array of length 10.

Table 10: Relative average round-trip values compared to security mode none with different security modes and the security policy Basic128RSA15.

Platform	Variable size	None	Sign	Sign and encrypt
Laptop	10	1	1.2	2.2
Laptop	10000	1	1.5	2.7
Raspberry Pi	10	1	1.1	1.6
Raspberry Pi	10000	1	1.3	3.8

the laptop computer, round trip times with signing take 1.5 times as long as the time without security. Adding encryption adds more overhead and round trip times with encryption take 2.7 times as long as without security. In the case of the Raspberry Pi, the corresponding values are 1.3 and 3.8. These values are illustrated in table 10. Of course in absolute measurements the laptop is always faster in round trip times. Looking at the relative numbers however shows that with small message sizes, adding signing or encryption actually adds relatively more overhead to the laptop version. When the message size is increased this relative difference is no longer true. Signing messages adds much less relative overhead than encrypting messages.

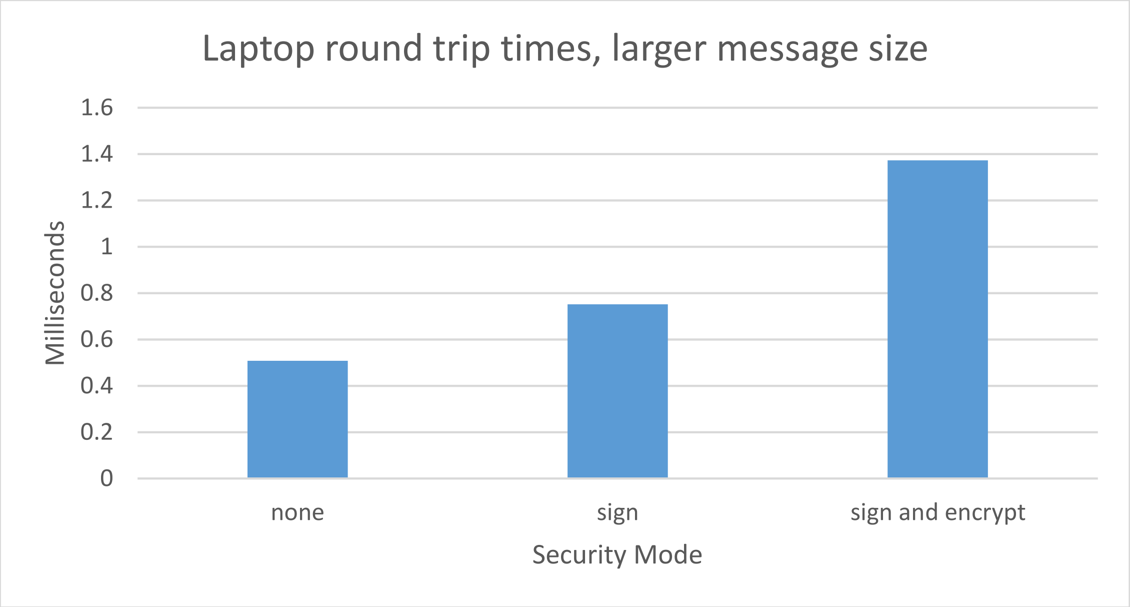


Figure 10: Average round-trip time with laptop and byte array of length 10 000.

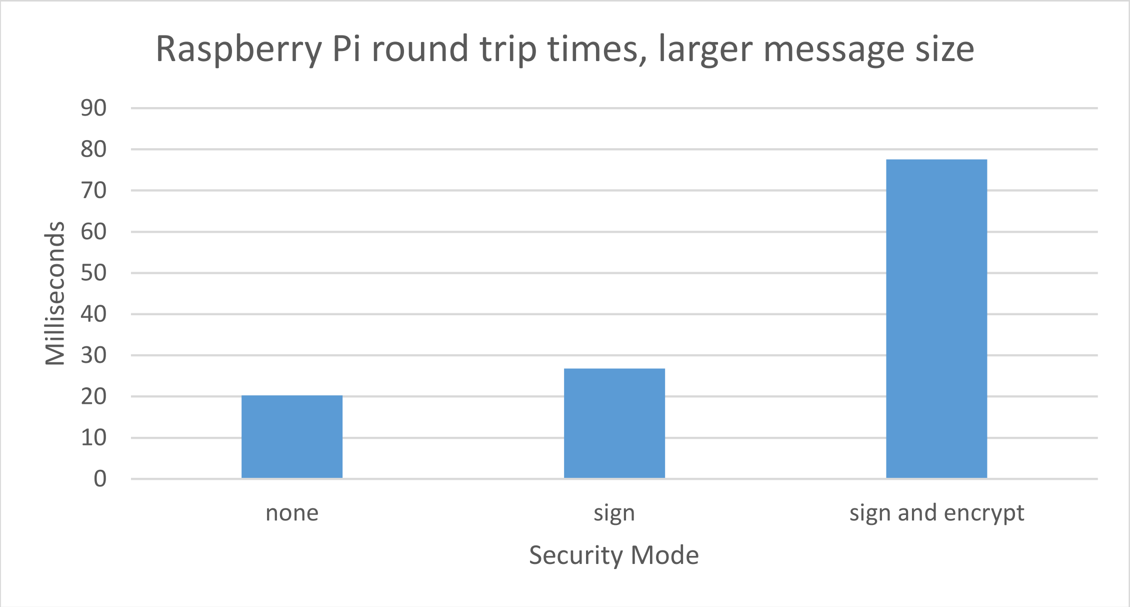


Figure 11: Average round-trip time with Raspberry Pi and byte array of length 10 000.

7.2 Response time results

7.2.1 Test setup

It is always advisable to have a separate performance testing environment resembling the production environment as much as possible. The following test setup was created for response time tests:

- Server machine:
    - OS: Windows 7 Enterprise 64-bit
    - CPU: Intel Core 2 Quad Q6600 @ 2.40GHz
    - RAM: 4,00GB Dual-Channel DDR2
    - Storage: SATA-II 3.0Gb/s Hard drive
- Client1:
    - OS: Windows 7 Enterprise 64-bit
    - CPU: Intel Core 2 Quad Q9300 @ 2.50GHz
    - RAM: 4,00GB Dual-Channel DDR2
    - Storage: SATA-II 3.0Gb/s Hard drive
- Client2:
    - OS: Windows 7 Enterprise 64-bit
    - CPU: Intel Xeon W3520 @ 2.67GHz
    - RAM: 4,00GB Triple-Channel DDR3
    - Storage: SATA-II 3.0Gb/s Hard drive

The default server application was SampleConsoleServer application which is shipped in connection with Prosys OPC UA Java SDK as a sample application. The version of SDK used in these tests was 2.2.0. An important part of OPC UA communication is the software library called stack, which is shipped by OPC Foundation. The version of OPC UA Java stack used in these tests was 1.02.337.5. Oracle Java SE Development Kit was used, specifically JDK 8u05. Additionally, in some tests UaDemo server was used. The version shipped in conjunction with Unified Automation UA SDK C++ Bundle 1.3.3 (Evaluation Edition) was used. The client side application was the before mentioned UaPerfClient. The computers were connected together by a fast LAN network.

7.2.2 Read mechanism

The first read test was run with one client application reading one value attribute from the server and the rate of read requests was varied. The maximum rate of reads was decided to be 1000 reads per second (one read per millisecond). The minimum

Table 11: One client reading one value from server

rate (reads per second)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
100	10.3	203.1	16.0	99
200	10.0	218.5	16.0	199.7
300	11.6	218.7	16.1	299.6
400	11.9	218.6	16.2	399.5
500	11.9	203.0	16.3	499.4
...	...	...	...	...
1000	12.9	218.6	16.9	998.7

amount worth of testing was 100 requests per second. The duration of the test was 180 seconds.

Table 11 shows that on average it takes 25 percent more time to read one value when the rate increases to 10 times the original. Considering the 90th percentile value, the difference is even smaller. It is noticeable that there is a good reason why the percentile values are widely used. The 90th percentile value is growing logically with more reads and not fluctuating. Average values are also pretty reliable. Some random variations can be seen in the maximum values from run to run, which is expected. When interpreting these results, it also has to be considered that the client side handling of read requests and responses is mostly counted in this time also. To test whether or not the response times change when two clients connect to the server at the same time, a test with a rate 1000 was run also with the other test client. After this, the measurement was done with both test clients started at the same time with rate being 1000. If the server is the bottleneck, connecting two simultaneous client applications should add further delay. The tests were started manually and it was made sure that they were started and finished during the same second time interval. Based on table 12, the conclusion can be drawn that the OPC UA server is not a limiting factor when client reads one attribute value per millisecond.

Table 12: Measurement with rate 1000

Client	Concurrent	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
1	no	12.9	218.6	16.9	998.6
2	no	14.0	203.1	16.6	998.7
1	yes	13.0	203.9	16.9	998.8
2	yes	14.0	219.4	16.6	998.7

The next test is to increase the amount of batch size per read request and keep the session count still at one. In practical terms, the read rate of previous tests (one value per millisecond) is really high. Therefore a rate of 100 reads per second was considered. This is still an unrealistically high value considering any real world application. It is used here because this way the test can overload the server and show rise in round-trip times. In previous test, the duration of 180 seconds was used. This was deemed needlessly long and following read tests were executed with the duration of 90 seconds as this seemed to provide good balance between the test duration and reliability and consistency of results.

In the following tests, SampleConsoleServer and UaPerfClient applications were run with explicitly defined JVM options in order to make sure that the applications would not run out of memory. These command line options were:

–Xmx2048m –Xms2048m

In table 13 it can be seen that 3000 value requests 100 times per second is clearly out of reach for the tested server. The total duration should be 90 but it is prolonged to 120 seconds. More clearly, the round-trip values have become unsustainable averaging at over one minute. At the level of 2500 value requests 100 times per second, the server is already pretty stressed and round-trip times have grown noticeably. Still at this level of batchsize 2500, the throughput remains good.

Table 13: Measurement with client 1 and variable amount of nodes to read

nodesToRead	Total duration (s)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
100	90	1.8	36.4	1.9	9998.3
500	90	6.6	143.1	6.5	49988.5
1000	90	16.4	813.4	13.4	99982.7
1500	90	28.2	1108.3	17.0	149971.8
2000	90	53.8	2511.7	26.6	199928.9
2500	90	103.8	3060.6	62.5	249922.7
3000	120	64030.1	112318.4	92214.8	223641.6

Many OPC UA servers have a setting that can be used to limit the number of simultaneous sessions. There must be some background to this setting, so testing the possible number of sessions is an interesting part of the read functionality. In order to reliably overload the server and be able to observe differences between OPC UA server applications, suitable values for batchsize and rate were selected. Based on previous tests, the number of batchsize 100 and rate 100 were deemed suitable. Amount of sessions was varied while the batchsize and rate were kept at these values.

The test was run with SampleConsoleServer on the server machine and UaPerfClient application on the Client1 machine. Server application was restarted after each test run. Results are shown in table 14. It is visible that the response times start to degrade rapidly with rising amount of sessions. However, the user might not notice this because only at the level of 25 sessions does the total duration start to rise and even at this level the throughput remains pretty much at the desired level. After 30 sessions, it is clear that the server application is not working correctly. Here the drop in throughput is clearly visible.

Table 14: Measurement with variable amount of sessions.

Sessions	Total duration (s)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
1	90	1.8	36.4	1.9	9998.3
5	90	2.8	36.5	3.7	49994.5
10	90	6.6	282.3	6.9	99984.4
15	90	11.0	512.1	10.2	149972.5
20	90	109.7	2336.5	16.6	199980.2
25	92	4164.3	7408.9	4797.1	244199.9
30	124	30204.8	54770.2	49640.6	216922.8

As in client-server application testing in general, also in this case it is important to assess whether we are actually measuring the server performance or client performance. If the 30 session test case from one physical machine seems to be the limit for throughput, then the naive solution would be to try the same test with two physical machines and 15 sessions each. Table 15 illustrates these results. It is clear that this tested server setup can deliver about 300,000 read responses to client applications per second. When cases with 15 and 16 clients are compared, the 16 client case experiences actually no additional throughput, but instead faces prolonged duration and greatly prolonged round-trip times of requests.

Based on these results it is also clear that two physical machines both hosting 15 sessions is not the same as one single application hosting 30 sessions. A part of this difference can be attributed to the fact that the read requests are distributed more evenly on really fine-grained level. Another reason is that the two client applications were started manually and thus there is a small difference in their starting times. However, there is also the chance that actually the client side application is the bottleneck in the 30-sessions test case. Table 16 lists a result when UA Demo Server was tested on the same hardware and using the same kind of configuration. Based on this result it can be reasoned that the client application is not the limiting factor in the situation depicted in table 14. Also, this result shows substantial performance differences between OPC UA products. For UA Demo Server, the response times are on a pretty much normal level for the 30 sessions test case.

Table 15: Measurement with two client machines and 15 sessions each

Client	Sessions	Duration (s)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
1	15	90	13.3	377.5	14.9	149967.7
2	15	90	64.3	2265.5	24.8	149986.8
1	16	95	2531.5	13275.5	8959.0	151527.3
2	16	99	2192.5	13337.5	8413.6	145111.6

Table 16: Measurement with UA Demo Server.

Sessions	Total duration (s)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
30	90	9.2	259.2	15.7	299951.3

Next logical test is to test when the UA Demo Server reaches its throughput limit. To assess this, the amount of sessions was grown from the 30. Results are shown in table 17. Up to the level of 35 sessions from one client, the UA Demo Server works quite correctly. At this level, again, total duration and round-trip times start to grow. Also, the throughput level gets lower, even though only a little. Only one session more, and on the level of 36 sessions, each reading 100 values 100 times a second, this particular server setup will not be able to handle all requests. It is interesting that the drop in server performance happens so fast in this situation. It seems that about 350,000 reads per second can be handled by the server in this case and the maximum is something below 360,000 reads per second.

Table 17: Measurement with variable amount of sessions and Ua Demo Server.

Sessions	Total duration (s)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
30	90	9.2	259.2	15.7	299951.3
31	90	9.8	325.4	16.7	309933.8
32	90	10.5	212.0	17.4	319945.6
33	90	63.5	337.4	124.9	329602.8
34	90	14.1	170.2	22.8	339949.6
35	92	181.8	401.1	238.5	348981.7
36	100	351.0	2789.0	634.4	22341.0

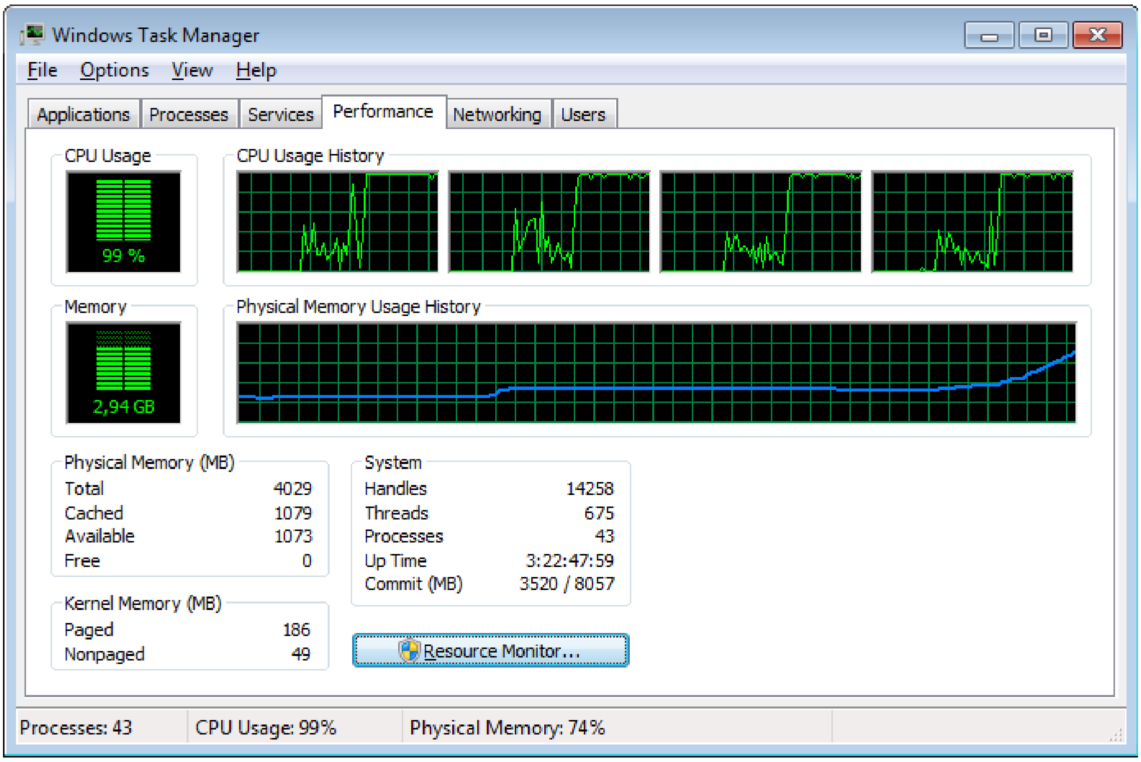
To test this hypothesis that about 360,000 values can be read from this particular server per second, two physical client machines were used. Here, the two test applications were started at the same time. Results are shown in table 18. In the case of two clients both initiating 17 sessions, the server is not able to respond to the load. This test shows that server throughput is something between 320,000 and 360,000 reads per second. It is noticeable that in the case of SampleConsoleServer, two physical machines noticed larger throughput than one physical machine only. In the case of UA Demo Server, the inverse is true. This is something that seems to be implementation specific.

During these measurements, the CPU and memory consumption of the server and client hosts was monitored manually. An interesting feature was noticed. The two OPC UA servers had clearly a different kind of implementation which was visible in the way they used CPU on the server machine. Figure 12 shows how the

Table 18: Measurement with variable amount of sessions and UA Demo Server. Client 1 and 2 were used the same time.

Sessions	Total duration (s)	Average (ms)	Max (ms)	90th percentile (ms)	Throughput (responses per second)
13	90	8.3	97.7	13.3	129978.9
13	90	6.3	28.9	12.1	129986.8
14	90	5.3	66.3	8.6	139983.6
14	90	5.4	25.5	9.5	139986.1
15	90	8.7	61.6	15.9	149963.3
15	90	9.2	48.2	14.7	149981.2
16	90	184.2	255.1	236.0	159971.4
16	90	182.2	254.7	234.8	159613.7
17	100	273.1	2977.6	424.8	7894.8
17	100	410.8	3552.3	1863.0	13787.6

SampleConsoleServer used CPU. This shows that all CPU cores are working quite symmetrically. On the other hand, figure 13 shows that the UA Demo Server was working in a different fashion. One of the CPU cores is fully utilized (first from the left in the picture), other is about 1/3 occupied (rightmost) and two cores are almost not used at all (two in the middle). It is also evident that the memory usage differs significantly between the servers. In figure 13 the memory usage is practically fixed whereas in figure 12 the memory grows rapidly. These kinds of reproducible test cases can be used to empirically quantify the benefit achieved from changes to the server side application. One important continuation is to look at why the memory is consumed and whether the working of the server can be changed. This however is outside the scope of this thesis.



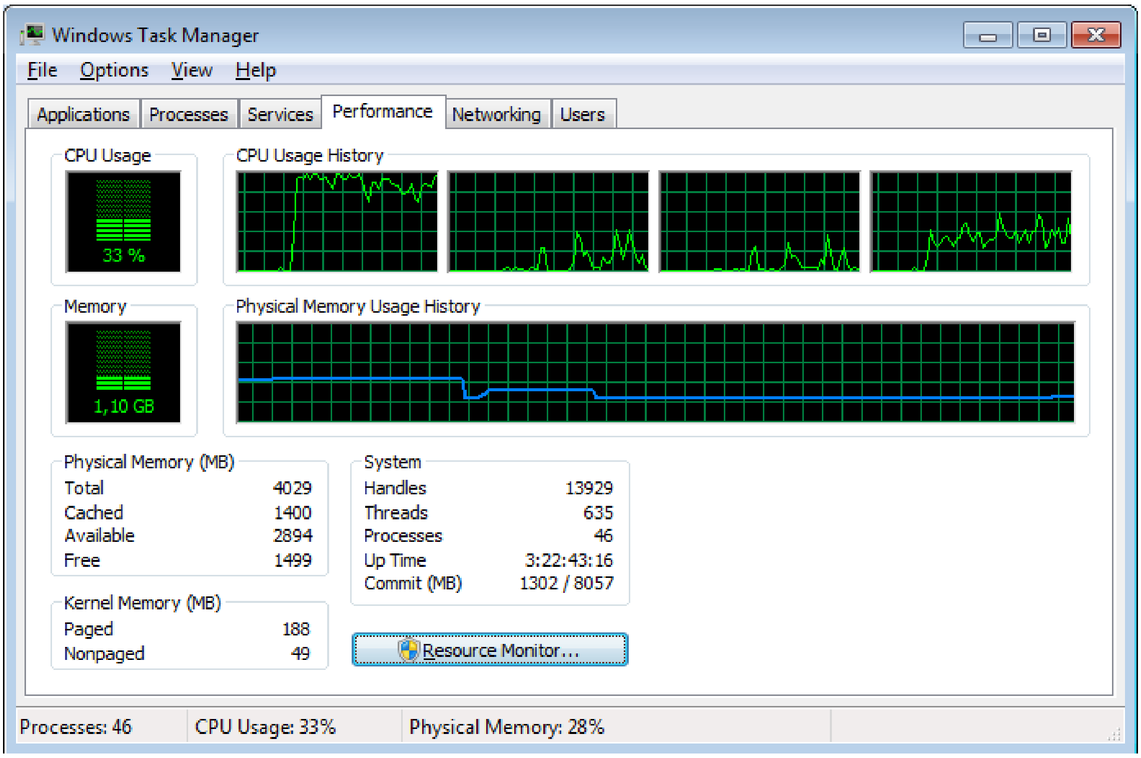
Physical Memory (MB)
Total 4029
Cached 1079
Available 1073
Free 0

Kernel Memory (MB)
Paged 186
Nonpaged 49

System
Handles 14258
Threads 675
Processes 43
Up Time 3:22:47:59
Commit (MB) 3520 / 8057

Processes: 43
CPU Usage: 99%
Physical Memory: 74%

Figure 12: SampleConsoleServer CPU usage with 300000 value attribute reads per second (30 session reading 100 values 100 times a second)



Physical Memory (MB)
Total 4029
Cached 1400
Available 2894
Free 1499

Kernel Memory (MB)
Paged 188
Nonpaged 49

System
Handles 13929
Threads 635
Processes 46
Up Time 3:22:43:16
Commit (MB) 1302 / 8057

Processes: 46
CPU Usage: 33%
Physical Memory: 28%

Figure 13: Ua Demo Server CPU usage with same settings as in figure 12

7.2.3 Subscription mechanism

In these subscription tests, the aim was to make as little changes as possible to the server side. This would allow the tests to be easily run with many different servers. However, usually there are some settings that directly prevent testing the server. Two of these most usual settings are the maximum number for sessions and subscriptions. In these tests, SampleConsoleServer and UA Demo Server were used. In SampleConsoleServer, these settings are configured with the following methods:

```
server.getSubscriptionManager().setMaxSubscriptionCount(50);
server.getSessionManager().setMaxSessionCount(500);
```

The UA Demo Server is configured with an XML file called ServerConfig.xml. In ServerConfig.xml, settings MaxSessionCount and MaxSubscriptionCount are used to set these limits. The comment on MaxSessionCount says "maximum number of sessions the server allows to create". The MaxSubscriptionCount is respectively "maximum number of subscriptions the server allows to create". In both options, value 0 is considered unlimited.

Subscribing to items and keeping track of subscriptions can be resource demanding on the client side. To make sure not to run out of memory, JVM options Xmx and Xms were used. Xmx option sets the maximum Java heap size and Xms option sets the initial and minimum Java heap size. Setting these to the same value causes the JVM to only use that one value for the whole test run. In all tests presented here, the client side was run with explicit JVM memory options.

```
-Xmx2048m -Xms2048m
```

In the read tests previously, the original test duration was 180 seconds. This 3 minute time interval seemed appropriate also with the subscription tests. In all these tests, if nothing different is stated, server application resides always on the "server" machine and client application on "client1" machine. In some tests the "client2" machine was used in addition to these. See chapter 7.2.1 for more information on the physical machines. Because the delay time measurement depends on server and client clocks, then the absolute value is not interesting but rather the changes when the amount of monitored items grows. Server and client applications were restarted between measurements.

The first test is to test response times with one session and subscription. Example command line options used are presented below.

```
-n "ns=2;s=MyLevel" -clients 1 -subscriptions 1 -items 1 -
    duration 180 opc.tcp://10.50.100.156:52520/OPCUA/
    SampleConsoleServer
```

Based on table 19 it can be seen that one client initiating 100000 monitored items is still normally handled by the server. The delay time grows somewhat, but no other sign of lesser throughput is visible. The amount of publish requests and responses grows, and this is probably because all data notifications do not fit into one publish response and notifications need to be divided into multiple responses. Publish requests are sent based on the amount of publish responses which explain the

Table 19: Measurement with variable amount of monitored items. Number of sessions and subscriptions is always one.

Monitored Items	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
1	180	180	180	180	0.99	16	25	23	180	180
1,000	180	180	180000	180	999	16	24	22	180	180
10,000	181	181	1810000	180	10051	22	32	29	181	181
100,000	361	361	18050000	180	100248	38	124	44	361	361

Table 20: Measurement with two client machines and 100,000 monitored items.

Concurrent test	Client	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
No	1	360	360	17949909	180	99718	48	162	56	360	360
No	2	366	366	18000000	180	99949	100	142	108	366	366
Yes	1	460	460	18000000	180	99962	52	134	66	460	460
Yes	2	540	540	18000000	180	99962	113	166	127	540	540

grown amount of publish requests. The "Min. updates" and "Max. updates" fields indicate the minimum and maximum amount of publish responses per subscription.

In table 19 it is clear that the delay is the only thing that really grows as the number of monitored items grows. Based on table 20, it can be seen that throughput remains the same even when two clients are subscribed to 100000 items at the same time. The delay grows, but throughput is on the same level as previously. Between these measurements the clocks were synced, so the absolute delay times are different. What we are trying to find is whether or not the delay grows in the server side application. The server side is not the bottleneck and it can be concluded that the server is capable of handling 200000 monitored items simultaneously. The server CPU usage level was 20% when 200000 monitored items were requested once per second. The number of publish requests and publish responses is different based on whether the tests are run concurrently. This seems weird and the reason for this needs to be studied. However, examining this detail is outside of the scope of this thesis.

100000 monitored items each changing per second can be considered a large amount and next we should consider the case where the number of subscriptions is other than 1. Let us start with 1000 monitored items per subscription and start growing the number of subscriptions. With growing number of subscriptions it quickly becomes clear that actually creating subscriptions can be time consuming. When creating 100 subscriptions takes 19 seconds, creating 500 subscriptions takes already 472 seconds in this setup.

Results in table 21 show that "items per second" value is not keeping up with the actual amount of data changes when amount of subscriptions and monitored items is growing. For example, in the case of 300 subscriptions, there should be 300000 items changing per second, and only 292314 changes are reported to the client. From the average and 0.9 percentile delay not much can be stated other than they do not seem to change much. From the max delay it can be stated that it clearly gets bigger with more subscriptions. The minimum and maximum subscription ID measurements come in handy here and show clear difference. In the case of 300 subscriptions, at least one subscription got publish response through to the client only 147 times, although the number should be 180 for all subscriptions. The first test case where the drop in subscription frequency is seen is the test case with 1 sessions, 20 subscriptions and 1000 monitored items. One thing that can be reasoned from this table is that creating subscriptions requires more resources and is more heavy than using monitored items. While the same server and client combination handled 100,000 monitored items easily, the combination seems to be incapable of reliably handling 100 subscriptions with 1000 monitored items each.

Table 21: Measurement with variable amount of subscriptions

Subscriptions	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
1	180	180	180000	180	999	44	53	51	180	180
10	1800	1800	1800000	180	9996	41	60	48	180	180
20	3565	3565	3569040	180	19820	52	75	60	173	181
30	5375	5374	5377778	180	29865	55	89	65	171	181
40	7059	7059	7102858	180	39445	62	102	69	171	181
50	8859	8858	8888344	180	49360	65	278	74	171	180
100	17586	17585	17710480	180	98344	44	77	51	172	180
200	35099	35099	35358603	180	196432	46	250	53	159	179
300	52404	52404	52632823	180	292314	54	955	59	147	180

Next the case where different amount of sessions is used is considered. In a normal use case of OPC UA, the client would probably make many monitored items but only small amount of subscriptions. Also, in previous test case it was confirmed that actually establishing subscriptions is more resource intensive than making monitored items. For these reasons, 10 subscriptions and 1000 monitored items per sessions were used.

Results of growing the amount of sessions are shown in table 22. The first thing that is visible from the results is that the delay times grow pretty much logically as the number of sessions is grown. The minimum number of updates for individual subscription ID that client application logs are getting lower as session count grows. With sessions 10, subscriptions 10 and monitored items 1000, the server still sends about the correct amount of updates. When the sessions count still grows bigger, some subscriptions just do not get all their updates through to the client. The exact reason for this is not clear, but the symptom is clear. This seems weird and is something that should be looked into.

Table 22: Measurement with variable amount of sessions. Amount of subscriptions is always 10 and amount of monitored items 1000.

Sessions	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
1	1800	1800	1800998	180	10001	0	19	6	180	180
2	3601	3601	3601998	180	20003	3	22	10	179	181
3	5402	5402	5402000	180	29999	6	21	13	180	181
4	7204	7202	7201465	180	39993	8	40	15	179	181
5	8994	8991	8991000	180	49931	9	37	16	179	181
10	18004	18003	17998662	180	99935	9	54	16	178	181
20	35847	35845	35889120	180	199308	11	94	18	163	181
30	53213	53201	53315790	180	296088	14	237	22	143	186
40	68439	68431	69027462	180	383340	25	599	42	135	182
50	83273	83255	84257888	180	467908	44	744	82	142	178

To test whether or not the client side application was the bottleneck, a test case with two physical machines hosting a test application with 50 sessions was made, see table 23. If this test shows similar results as the test in table 22, then it can be argued that the client side test application is the bottleneck. If on the other hand, the throughput is lower, then server side is most probably the bottleneck. Results show that the client side test application probably is not the bottleneck. From these results, no definite limit for the subscription performance can be decided. At least the test case depicted in table 23 is clearly out of the reach for the SampleConsoleServer in the depicted hardware platform. Also, it could be argued that the test case for 20 sessions in table 22 is not sustainable as some subscriptions only get 163 responses through whereas they should get 180 responses through to the client. In a normal production system, the amount of data notification would probably fluctuate and it seems that the server could probably sustain momentary overload for some time without crashing.

Table 23: Measurements with SampleConsoleServer and two physical machines subscribing to 50 sessions each.

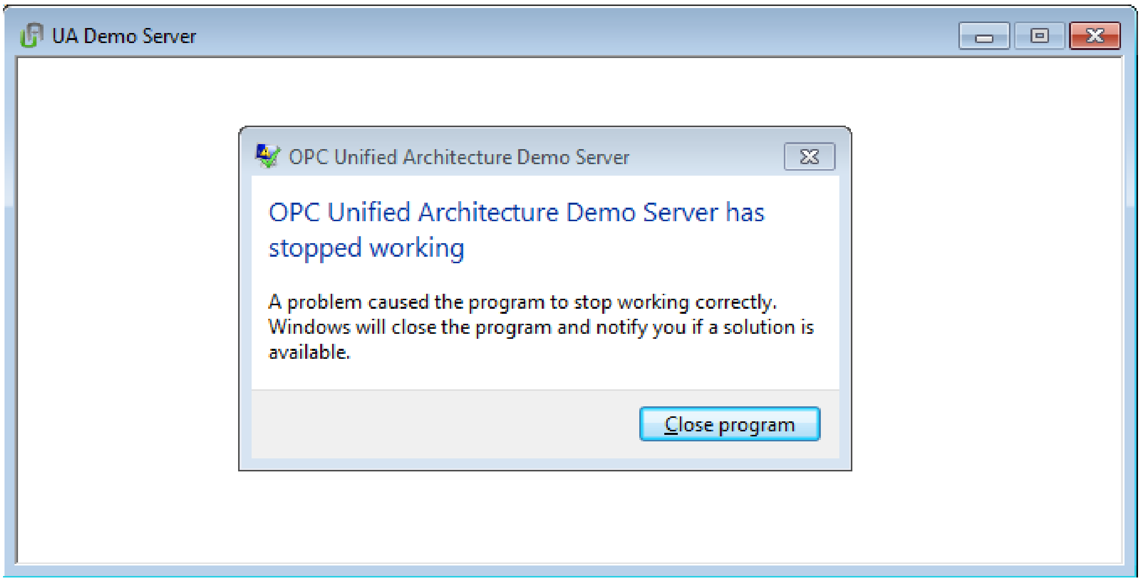
Client machine	Sessions	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
1	50	72584	72562	66950570	180	371901	679	2057	706	137	166
2	50	74013	73993	69228285	180	384436	303	4359	317	141	175

As in the case of read testing, a different OPC UA server was tested to see whether or not some differences between OPC UA implementations exist. The same settings were used as in previous testing with growing amount of sessions. The updates per subscription show different values than previously but this is normal as the variable in question might not be actually changing every second. These tests were started for example with the following command line options:

```
-n "ns=4;s=Counter1" -clients 1 -subscriptions 10 -items
    1000 -duration 180 opc.tcp://<ip-address>:4841
```

Table 24: Measurements with UA Demo Server. See table 22 for comparison.

Sessions	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
1	1452	1452	1441000	180	8002	57	58	58	144	155
2	2871	2871	2871000	180	15943	59	76	61	143	145
3	4291	4291	4290000	180	23824	62	64	63	143	144
4	5711	5711	5705000	180	31682	49	66	50	142	145
5	7122	7121	7113000	180	39498	55	71	55	141	146
10	13855	13852	13843000	180	76896	65	97	65	137	141
20	20221	20200	20200000	180	112179	64	111	65	101	101
30	24976	24952	24952000	180	138617	70	163	70	83	84
40	...	...	...	...	...	...	...	...	...	...
50	...	...	...	...	...	...	...	...	...	...



OPC Unified Architecture Demo Server has stopped working

A problem caused the program to stop working correctly. Windows will close the program and notify you if a solution is available.

Figure 14: UA Demo Server error when 30 sessions make 10 subscriptions each containing 1000 monitored items.

Results for UA Demo Server in the table 24 show similar results as in SampleConsoleServer test up to the case of 10 sessions. The subscriptions show somewhat different results but overall everything seems to go properly. After this, with test cases 20 sessions and 30 sessions, the drop in throughput can be observed in the minimum and maximum subscriptions IDs (min. updates and max. updates). Finally, with test case of 30 sessions, the server stops working correctly. One interesting thing is that the measured delay does not grow very much. Thus, it can be said that the server is not able to handle all subscriptions correctly, but those publish responses that it sends arrive at the client pretty quickly. Another difference between SampleConsoleServer and UA Demo Server can be observed in the minimum and maximum subscriptions IDs. Taking the 30 sessions test case as an example, the SampleConsoleServer provides figures of minimum 143 and maximum of 186 publish responses per subscriptions. In this same case, UA Demo Server provides corresponding figures of 83 and 84. Even though UA Demo Server clearly is not sending all publish responses, it is sending an equal amount of responses per subscription. In SampleConsoleServer, some subscriptions send much less publish responses than others.

Because of the error shown in figure 14, measurements with more sessions than 30 were not made. This however raises the question whether the crashing was somehow caused by the client. To test whether or not this really was the limit of the server, a test case was prepared where two physical client machines started 15 sessions each. In this case the server application did not crash, which was a little surprising. This again shows the importance of using actual physically different machines in performance tests. Results of this test are shown in table 25. Based on this it seems that this really was the limit that the server could handle. This is because the combined 'items per second' measurement shows similar value as in the test with one physical machine and 30 sessions. This is also true for the 'data changes' measurement. Also, minimum and maximum subscription ID frequencies are the same as in the test case with one physical machine. This validates that the test client works and can overload the server.

Table 25: Measurements with two physical machines subscribing to 15 sessions each.

Client machine	Sessions	Publish requests	Publish responses	Data changes	Duration (s)	Items per second	Avg. delay (ms)	Max delay (ms)	0.9 percentile (ms)	Min. updates	Max. updates
1	15	12598	12593	12593000	180	69934	156	234	157	83	84
2	15	12542	12532	12532000	180	69596	502	549	502	83	84

8 Conclusions

OPC UA allows the user to configure security mode settings flexibly. Overhead of security modes were assessed with elapsed time measurements. Adding signing or encryption imposes overhead to OPC UA communication. With larger message sizes, the overhead is proportionally larger. With less powerful hardware, the overhead is larger in absolute measures but not necessarily in relative measures. General statements in performance studies are dangerous. However it is fairly safe to say that with desktop grade hardware the overhead of encryption is negligible. With embedded hardware some consideration should be applied, but in a normal case there should be no problem. Signing messages adds considerably less overhead than encrypting messages.

With read and subscription testing, an example test client was presented. This serves as a proof-of-concept for the idea of a generic response time test client for OPC UA servers. The client was developed with the purpose that it could overload server applications. Multiple differences between the tested servers were observed. Read service testing offered an insight into how much read requests the servers are able to handle. The tested servers proved to be able to handle hundreds of thousands of value reads per second. In this particular setup, the Prosys OPC UA Java SDK based SampleConsoleServer was able to handle 250,000 - 300,000 reads per second and the Unified Automation C++ UA SDK based UA Demo Server was able to handle 320,000 - 350,000 reads per second. The testing also showed that there are differences between different server implementations in terms of resource usage.

In subscription testing, the limit of sessions, subscriptions and monitored items of the two example server applications were studied. Both servers were able to handle hundreds of thousands of monitored items changing once per second. For both servers, the test case with 300,000 monitored items allocated to 300 subscriptions and 30 sessions can be considered as an upper limit in this particular setup. For UA Demo Server, this test case resulted in error which stopped the server application. For SampleConsoleServer, the limit was observed in that some subscriptions did not send publish responses with the desired frequency. Testing confirmed the fact that normally multiple monitored items should be made instead of multiple subscriptions. During subscription testing and in high load situations, the tested servers handled subscriptions differently which was clearly visible in the minimum and maximum number of publish responses per subscription. When an application is saturated with more requests than it can handle, the outcome can be graceful degradation or abrupt ending of all application functionality. The tested servers exhibited different behaviour also in this area.

One of the research questions was to figure out what kind of issues surround performance testing of OPC UA applications. Testing that all data notifications arrive reliably from subscriptions under heavy load proved to be difficult. This reliability is also perhaps the most central feature of subscriptions and thus, this aspect should be figured out. However, with small and moderate load the tested OPC UA servers handled all value updates reliably.

The test client developed during this thesis proved to be practical when analysing

OPC UA server applications. Reproducible test cases that are easy to run proved to be useful in product development work. During performance testing, parameters with most effect on read and subscription performance were uncovered. Thus, it can be argued that this thesis provides information that is valuable for product development of OPC UA SDKs as well as to users of OPC UA applications. There are many features that could be added to the test client. For example, support for more services could be added. Write and browse services are the most obvious ones. Also, more configuration options for existing services could be added. For example, now the reads and subscriptions were always made to only one node. This thesis concentrated on server side testing. A possible future agenda could be to broaden the scope to client side testing.

References

[1] T. Sauter and M. Lobashov, "How to access factory floor information using internet technologies and gateways," *Industrial Informatics, IEEE Transactions on*, vol. 7, pp. 699–712, Nov 2011.

[2] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC Unified Architecture*. Springer, 2009.

[3] "Iec webstore: Iec/tr 62541-1:2010, opc unified architecture - part 1." `http://webstore.iec.ch/webstore/webstore.nsf/Artnum_PK/43769`. Accessed: 2015-04-01.

[4] OPC Foundation. OPC Unified Architecture Specification, Part 1: Overview and Concepts. Release 1.03. 2015.

[5] P. Piirainen, "Opc ua based remote access to agricultural field machines," master's thesis, Aalto University, 2014.

[6] A. Claassen, S. Rohjans, and S. Lehnhoff, "Application of the opc ua for the smart grid," in *Innovative Smart Grid Technologies (ISGT Europe), 2011 2nd IEEE PES International Conference and Exhibition on*, pp. 1–8, IEEE, 2011.

[7] Intel, Ascolab, and Unified Automation, "Reducing product development effort for opc unified architecture." white paper: http://www.ascolab.com/en/company-media/company-press-releases/150-intel-case-study.html, 2009.

[8] "Opc-ua end user considerations." `https://wikis.web.cern.ch/wikis/display/EN/OPC-UA+End+User+Considerations`. Accessed: 2015-03-30.

[9] Prosys PMS Ltd. OPC UA Client Java SDK & OPC UA Server Java SDK. 2015. `http://www.prosysopc.com/`.

[10] D. Mosberger and T. Jin, "httperf—a tool for measuring web server performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[11] A. Bog, *Benchmarking transaction and analytical processing systems : the creation of a mixed workload benchmark and its application*. Springer, 2014.

[12] M. Fojcik and K. Folkert, "Introduction to opc ua performance," in *Computer Networks* (A. Kwiecień, P. Gaj, and P. Stera, eds.), vol. 291 of *Communications in Computer and Information Science*, pp. 261–270, Springer Berlin Heidelberg, 2012.

[13] C. Groba and S. Clarke, "Web services on embedded systems - a performance study," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pp. 726–731, March 2010.

[14] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes, *Java Concurrency in Practice*. Pearson Education, 2006.

[15] S. Oaks, *Java Performance: The Definitive Guide*. O'Reilly Media, 2014.

[16] K. Huppler, "The art of building a good benchmark," in *Performance Evaluation and Benchmarking* (R. Nambiar and M. Poess, eds.), vol. 5895 of *Lecture Notes in Computer Science*, pp. 18–30, Springer Berlin Heidelberg, 2009.

[17] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, pp. 57–76, Oct. 2007.

[18] "Performance workload creation and execution framework faban." `http://faban.org/`. Accessed: 2016-02-29.

[19] O. Palonen, "Object-oriented implementation of opc ua information models in java," master's thesis, Aalto University, 2010.

[20] J. Kurose and K. Ross, *Computer Networking: A Top-down Approach*. Pearson, 2013.

[21] OPC Foundation. OPC Unified Architecture Specification, Part 3: Address Space Model. Release 1.03. 2015.

[22] "Opc ua default information model nodes.." `http://www.opcfoundation.org/UA/schemas/1.02/NodeIds.csv`. Accessed: 2015-04-20.

[23] OPC Foundation. OPC Unified Architecture Specification, Part 5: Information Model. Release 1.03. 2015.

[24] OPC Foundation. OPC Unified Architecture Specification, Part 4: Services. Release 1.03. 2015.

[25] OPC Foundation. OPC Unified Architecture Specification, Part 6: Mappings. Release 1.03. 2015.

[26] OPC Foundation. OPC Unified Architecture Specification, Part 2: Security Model. Release 1.03. 2015.

[27] K. Folkert, M. Fojcik, and R. Cupek, "Efficiency of opc ua communication in java-based implementations," in *Computer Networks* (A. Kwiecień, P. Gaj, and P. Stera, eds.), vol. 160 of *Communications in Computer and Information Science*, pp. 348–357, Springer Berlin Heidelberg, 2011.

[28] S. Cavalieri and G. Cutuli, "Performance evaluation of opc ua," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pp. 1–8, IEEE, Sept 2010.

[29] M. Freund, C. Martin, A. Braune, and U. Steinkrauss, "Jsua—an opc ua javascript framework," in *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pp. 1–4, IEEE, Sept 2013.

[30] T. Burke, “The performance and throughput of opc,” tech. rep., Rockwell Software Inc., 1998. White paper, available: `http://www.ascolab.com/images/stories/ascolab/doc/opc_performance.pdf` last accessed 27.02.2015.

[31] T. Miettinen, “Synchronized cooperative simulation: Opc ua based approach,” master’s thesis, Aalto University, 2012.

[32] O. Post, J. Seppälä, and H. Koivisto, “The performance of opc-ua security model at field device level.,” in *ICINCO-RA*, pp. 337–341, 2009.

[33] S. Cavalieri, G. Cutuli, and S. Monteleone, “Evaluating impact of security on opc ua performance,” in *Human System Interactions (HSI), 2010 3rd Conference on*, pp. 687–694, May 2010.

[34] “Subscribing to a very large number of nodes in a prosys opc ua java sdk based server.” `http://prosysopc.com/blog/subscribing-to-a-very-large-number-of-nodes-in-a-prosys-opc-ua-java-sdk-based-server/`. Accessed: 2015-04-01.

[35] “Opc ua vs opc da performance comparative.” `https://wikis.web.cern.ch/wikis/display/EN/OPC+UA+vs+OPC+DA+performance+comparative`. Accessed: 2015-03-30.

[36] S. Rohjans, *Semantic Service Integration for Smart Grids*, vol. 14 of *Studies on the Semantic Web*. IOS Press, 2012.

[37] S. Chilingaryan and W. Eppler, “High speed data exchange protocol for modern distributed data acquisition systems based on opc xml-da,” in *Real Time Conference, 2005. 14th IEEE-NPSS*, pp. 5 pp.–, June 2005.

[38] S. Sucic, “Optimizing opc ua middleware performance for energy automation applications,” in *Energy Conference (ENERGYCON), 2014 IEEE International*, pp. 1570–1575, IEEE, May 2014.

[39] “Opc ua performance benchmarks forum discussion.” `http://forum.prosysopc.com/forum/opc-ua-java-sdk/opc-ua-performance-benchmarks/`. Accessed: 2015-05-14.