Deliverable: 3.2

Assessment of Current Service Bus Technologies

Deliverable Responsible: Introsys, SA
Version: 1.1

29/09/2016

Dissemination level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (excluding the Commission Services)	

Project co-funded by the Horizon 2020
Programme of the European Union

Project Information

Acronym	openMOS
Name	Open dynamic manufacturing operating system for smart plug-and-produce automation components
Theme	FOF-11-2015: Flexible production systems based on integrated tools for rapid reconfiguration of machinery and robots
Grant agreement	680735
Start date	1, October 2015
Duration	36 months

Contact Information

Company Name	Introsys – Integration for Robotic Systems, SA
Address	Parkim – Parque Industrial da Moita Rua dos Girassóis, Nº1, Armazém A3 2860 – 274 Moita Portugal
E-Mail	info@introsys.eu
Phone	+351 212 951 499
Fax	+351 212 893 000

Version Control

Version	Date	Change
0.1	06/06/2016	Initial draft and document structure.
0.2	07/07/2016	Merged preliminary evaluation of protocols.
0.3	15/09/2016	Included chapter for OPC UA.
0.4	20/09/2016	Included chapter for CoAP.
0.5	21/09/2016	Overall additions and revision by Loughborough Univ. and fortiss.
0.6	27/09/2016	Included remaining chapters.
1.0	28/09/2016	First complete version.
1.1	29/09/2016	Overall revision.

List of Authors

Name	Role	Affiliation	Email
Ricardo Matos	Author	Introsys, SA	ricardo.matos@introsys.eu
Magno Guedes	Author	Introsys, SA	magno.guedes@introsys.eu
Ivo Pereira	Contributor and Reviewer	Loughborough University	I.Pereira@lboro.ac.uk
Kirill Dorofeev	Contributor and Reviewer	fortiss	dorofeev@fortiss.org
Luis Flores	Reviewer	Introsys, SA	luis.flores@introsys.eu
Stefan Profanter	Contributor and Reviewer	fortiss	profanter@fortiss.org

Table of Contents

Table of Figures

List of Acronyms

API: Application Programming Interface: Is a set of routines used to interface software and applications.

CoAP: Constrained Application Protocol. Is a software protocol for M2M communication.

COM: Component Object Model. Is a Microsoft standard to enable IPC and dynamic object creation.

DCOM: Distributed COM. Is a Microsoft standard for M2M communication.

DDS: Data Distribution Service. Is an OMG standard for M2M communication.

DNS: Domain Name System. Resolves full qualified domain names to IP addresses.

DTLS: Datagram Transport Layer Security. Is a protocol that provide security for UDP communications.

EMF: Eclipse Modelling Framework.

GDS: Global Discovery Server. Is a concept for network wide discovery functionality in OPC UA servers.

HMI: Human-Machine Interface.

HTTP: HyperText Transfer Protocol. Is an application protocol for information technologies.

HTTPS: HTTP Secure. Is a protocol that enables encrypted communications over HTTP.

IP: Internet Protocol. Is the principal communications protocol for the Internet.

IoT: Internet of Things. Is the nomenclature for a system of interconnected physical devices.

IIoT: Industrial IoT. Is the concept of IoT applied in industrial environments.

IPC: Inter-Process Communication. Is the act of sharing information between processes running either in the same or remote hosts.

JSON: JavaScript Object Notation. Defines a representation format for data objects.

LDAP: Lightweight Directory Access Protocol. Is a standard for accessing and maintaining distributed directory information services over IP networks.

LDS: Local Discovery Server. Offers server discovery functionality for locally hosted OPC UA servers.

LDS-ME: LDS with Multicast Extension. Uses mDNS for discovering other OPC UA servers.

M2M: Machine-to-Machine. Commonly used for referencing interactions between machines.

mDNS: Multicast DNS. Broadcasts DNS messages for discovery and service descriptions.

MQTT: Message Queuing Telemetry Transport. Is a standard for M2M messaging communication.

MSB: Manufacturing Service Bus. Is the openMOS communication and service stack.

OAuth: Open standard for Authorisation. Is a service for sharing login information across internet applications.

OPC: Open Platform Communications. Is a standard for industrial communications.

OPC A&E: OPC Alarms and Events. Is a set of standards for OPC.

OPC DA: OPC Direct Access. Is the first group of specifications for OPC.

OPC HDA: OPC Historical Data Access. Is a set of standards for OPC.

OPC UA: OPC Unified Architecture. Is the new generation of OPC.

OPC XML-DA: OPC XML Data Access. Is a set of standards for OPC.

QoS: Quality of service.

REST: Representational state transfer. Is an architectural model for web applications and protocols.

SCADA: Supervisory Control and Data Acquisition. Is a system for remote monitoring and control.

SDK: Software Development Kit. Is typically a set of software development tools that allows the creation of applications.

SOAP: Simple Object Access Protocol. Is a protocol for exchanging structured data.

TCP: Transmission Control Protocol. Is a transport protocol for the Internet.

UDP: User Datagram Protocol. Is a transport protocol for the Internet.

UPnP: Universal Plug and Play. Is a set of protocols for service discovery over networked devices.

URI: Uniform Resource Identifier. Is an identifier for resources

URL: Uniform Resource Locator. Is an address to identify the location of networked resources.

WS: Web Service. Is a service offered by a device, over the Internet.

WS-SecureConversation: Is a protocol to enable security context for web services.

X.509: Is a standard for managing digital certificates.

XML: eXtensible Markup Language. Is a set of rules for encoding data.

1. Introduction

This document has been produced by the partners in the openMOS consortium with the aim of assessing current protocols and technologies prone to be implemented in the openMOS communication stack that will provide the manufacturing service bus (MSB). The following is an extract from the openMOS description of work [1] stating the objectives of the task that include the production of this deliverable:

> *"**Task 3.2: Implementation and test of open plug-and-produce manufacturing service bus [Introsys, M5-M21]***
>
> *Involved partners: IntRoSys*
>
> *This task is focused in the assessment of the different protocols supporting plug and produce. This includes the preliminary test of candidate technologies for implementation (OPC-UA, DDS, UPnP, DPWS) in respect to security, data synchronization and data persistence. This analysis should be the base for the implementation of the service bus that will enable the virtualization of the manufacturing components and support a cloud based modular deployment of the cyber representation of the manufacturing components.*

This task has the ultimate goal of implementing a communication medium that can be used to facilitate registering and interactions for openMOS devices and components. The delivering of this document finalises the first phase of the task, by providing a view on what current technologies and standards may better adapt to the openMOS technical and architectural requirements. The conclusions offered in this document should not impose a final decision on the content of the MSB communication stack, neither its architecture, but rather describe the pros and cons, as well as provide the expected functionality and performance given by each considered solution.

1.1. Methodology

This assessment was carried out by first studying current protocols used for machine-to-machine communication and pinpointing those that better met the functionality needed for openMOS [2] [3]. Subsequently, for each protocol, the most promising SDK solutions were analysed and tested in terms of performance. The results of these tests are offered in this document.

1.2. Document structure

The remaining of this document is structured as follows: Section 2 provides a detailed analysis on the requirements, pinpointed by the openMOS consortium, regarding the communication stack; Section 0 offers a brief overview on current methods and protocols commonly used in network communications; Section 4 provides an evaluation of candidate service bus technology; Section 5 shows a comparison on the evaluated technology; and Section 6 offers some conclusions.

2. Analysis of openMOS requirements for the Manufacturing Service Bus

In the early stage of the openMOS project, a detailed set of requirements was specified and reported [2]. This list of requirements had the purpose of defining a consensual framework that could meet the expectations of the project stakeholders, without disregarding physical and technical limitations inherently present in the target deployment environments. Here, we recall this list to give a brief overview of the specific requirements that affect directly the work related with the implementation and functionality of the communication stack (manufacturing service bus, from now on mentioned as MSB), for openMOS devices and components.

2.1. Functionality

The functional requirements reflect the essential behaviour of the MSB according to the demand of the devices and components that need to interact with each other, in the openMOS system. The main required functionality for the MSB is listed below:

- Support all interactions: the MSB must be able to communicate and interpret data from all different openMOS enabled devices and components, which means that exchanged information must be mapped or bind to a common semantics.

- Support legacy systems: the cost-effectiveness of new manufacturing paradigms is influenced by the amount of investment needed for changing current methods. Thus, the ability for the new systems to communicate, directly or through adapters, with the legacy ones, should be considered.

- Real-time or, at least, fast communication: the necessity for real-time communication within the openMOS system is still an open discussion. This is due to the fact that the consortium already decided that the MSB is not responsible for transporting time critical information, such as safety or control signals. Nonetheless, communications should be fast enough to enable prompt intervention on the manufacturing process as, and when needed.

- Service discovery: the MSB should enable newly added devices to discover the localisation (IP and port) of services available on the network, through a commonly known discovery mechanism.

- Data persistence: the MSB should allow the persistence, to a certain extent, of data exchanged on the network.

- Single or multiple standards: due to the heterogeneous nature of the target industries addressed by the openMOS project, the MSB is required to be flexible enough to support multiple communication standards (either directly or through adapters), and synchronize different transport protocols.

- Synchronous or asynchronous communication: the MSB should allow synchronous communication (e.g. for re-configuration or other action that requires the interruption of a process), as well as asynchronous communication (e.g. for streaming sensor data).

- Scalability: as the level of granularity used for representing openMOS devices and components may vary between complete workstations to single sensors and actuators, the number of correspondent openMOS enabled components linked with the MSB can easily scale to the order of several hundreds of individuals. This aspect should be handled by the MSB without notorious loss of performance.

- Offline functionality: besides offering Internet/cloud services, the MSB should also be able to handle offline (i.e., without Internet connection) functionality.

2.2. Security

- Authentication: by offering an Internet wise system for manufacturing processes, the authentication of devices and personnel is critical for the openMOS project and should be considered in all interactions with the MSB.

- Authorisation: being a common medium for communication across several levels of functionality and information, the MSB should control who have access to what. Consequently, mechanisms for authorising access for devices and users should be considered.

- Encryption: some of the information exchanged through the MSB may contain confidential data that must be prevented from being accessed by intrusion, such as "man-in-the-middle" attacks to the network. Thus, the MSB should be able to secure and encrypt such data.

2.3. Constraints

- Constrained network environments: the MSB should be able to support being deployed in constrained network environments, which can be the case of some industrial sites. This means that bandwidth can be limited, packets may frequently get lost during transport and data can become corrupt because of electromagnetic interference.

- Conflicting requirements: some identified functional requirements may cause conflicts during runtime. For instance, the capability to handle different communication standards into a single communication medium may hamper real-time capabilities. Thus, the MSB implementation must be able to handle prioritization and automatic activation/deactivation of standards and/or functionality, based on the configuration parameters defined during the MSB deployment.

3. Machine-to-Machine communication breakdown

A substantial part of the openMOS project relies on the characteristics of the machine-to-machine communication channels that allow the different components, devices and software to interlock and exchange data in a common network. In order to comprehensively assess the current solutions that can better adapt to, or offer the ability to be extended and meet the requirements for the openMOS service bus, it is necessary to have a general look on the technology present at the different layers composing a complete end-to-end communication medium.

This section provides a brief overview on current methods and protocols, used at different layers and/or for different purposes, either individually or in conjunction, to provide an infrastructure for machine-to-machine communication.

3.1. Architecture options

This section provides an overview of the possible architecture options for M2M communication systems, from the perspectives of access, deployment and interaction.

3.1.1. Access to services

The access to MSB services may follow two different architectures: a) indirectly through API's, or b) directly, as show in the following figure:

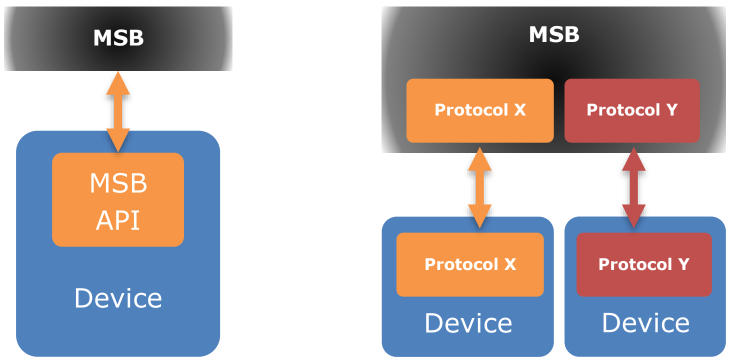


Figure 1 - Access to MSB services. Left: indirectly, through API. Right: directly.

Indirect approach: an API is provided as a library or source code to be included in interaction devices, giving function calls to connect and communicate with the MSB. This API may implement current or new standards and protocols, assuming that all devices follow the same methodology. This approach has the advantage of giving an enclosed, self-sustained kit that eases the integration of the communication stack within a custom application. On the other side, the advantages relate to a bigger implementation effort, less flexibility and less compatibility with current systems.

Direct approach: the MSB implements direct access to the current standards and protocols used by devices. This approach has the advantage of complying with current systems, thus relieving the effort of customisation and integration of new solutions. Also, it is more easily scalable. However, might be a more complex solution if it is required for different protocols and standards to interlock and translate between themselves.

3.1.2. Deployment

Typically, being a multi-functional middleware framework, rather than a single communication bus, the MSB may also be decentralised, regarding its deployment on the network, as shown below:

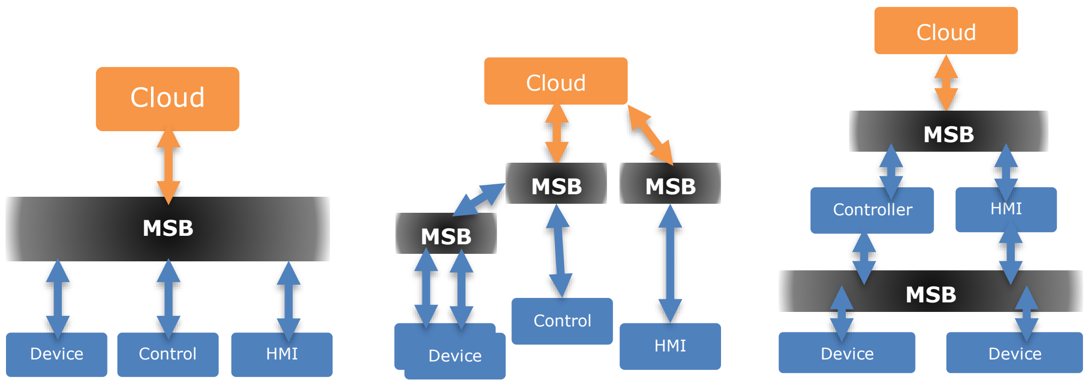


**Figure 2 - Deployment of MSB services over a network.
Left: centralised. Middle: distributed. Right: Hierarchical.**

Centralised approach: in a centralised approach, all services, parameters, functions, namespaces, etc. are accessed through a single instance, or endpoint of the MSB, even if different communication standards or protocols are used simultaneously. This is the less complex approach but also the less flexible.

Distributed approach: sometimes, in order to distress the network and reduce communication overhead, endpoints may be fragmented and deployed only with the strict functionality needed for each application or system. This approach can be more flexible and scalable at the cost of additional interaction, synchronization and concurrency issues that need to be taken into consideration[1].

Hierarchical approach: in a particular case of the above, MSB instances can be organised hierarchically, that is having endpoints dedicated for low-level devices and others for high level components. This approach may resolve some of the issues mentioned on the above, but may add more communication latency for data flowing between the low and the high levels.

[1] The UPnP protocol (see section 3.6.3) is an example of a distributed communication architecture.

3.1.3. Interaction

Regarding interaction, there are two major approaches that can be considered: client-server and publish-subscribe.

Client-server: this interaction model represents strict point-to-point communication where, on one side, there are data/resource providers (clients), and on the other, there are data/resource requesters (clients). In this approach, clients can connect and disconnect to servers dynamically, but servers should always be available. This also provides better (more reliable) communication with less lost packets, the communication overhead can be higher due to and handshake procedures. Multicast communication is possible only by keeping several open connections, which reduces the network's throughput.

Publish-subscribe: in this paradigm there is not an explicit link between data senders and receivers. Here, information providers (publishers) are responsible for exposing data and resources in a given location on the network (broker or event bus), which can then be reached by information consumers (subscribers). Publish-subscribe systems are typically more scalable and more adequate for multicast communication.

3.2. Transport

The transport layer is a critical topic for communications and deserves to be well studied and understood in order to better meet the requirements of the applications. In the Internet Protocol (IP) suite, considered for the openMOS's communication stack[2], there are two opposing protocols to address: TCP and UDP.

3.2.1. TCP

The Transmission Control Protocol (TCP) offers reliable, ordered, and error-checked delivery of data packets between applications. TCP is connection oriented, thus, is typically used in a client-server model and peer-to-peer communications. The protocol is optimised for accurate, rather than fast delivery. Thus, is not well suited for real-time communications or data streaming. However, it already offers flow/congestion control, error detection and retransmission of lost packets. It is also able to detect lost connections, which is useful to infer issues on devices.

3.2.2. UDP

The user datagram protocol (UDP) is an alternative to TCP for a faster and more lightweight transmission. UDP does not require an active connection, just sending messages unreliably (i.e., without knowing if they reached their destination) over the network. There is no congestion control and packages may arrive unordered, or don't even arrive. Also, the UDP packets are smaller that TCP ones, mainly because of a

[2] The IP suite is widely used and offers obvious advantages for communicating over the Internet, rather than using very specialised solutions.

reduced header size. This protocol is more adequate for broadcasting and fast data streaming where losing some data packets is not a major concern. When using UDP, applications may require the addition of quality of service mechanisms to improve reliability of communications. However, this is specific for applications and not supported directly by the transport protocol.

3.3. Messaging

Regarding the messaging perspective, there are several different approaches, which basically relate with the structure/format of data and the delivery process.

3.3.1. HTTP

The hypertext transfer protocol (HTTP) is a widely used application protocol for transmitting data over the Internet. HTTP messages consist of four different fields: request line, request header, empty line and an optional body. The request line and header specify the type of message (GET, POST, PUT, DELETE, etc.) and options, while the body contains the data to be exchanged. HTTP accepts any type of byte encoded data. A special field in the header allows to specify the type of encoding used to encode the body. HTTP is commonly used over TCP transport, however can be adapted for UDP. Considering machine-to-machine communication, HTTP may be too complex and inefficient, having currently more adequate alternatives.

3.3.2. RTP

The real-time transport protocol (RTP) is mainly used for delivering audio and video over the Internet. Typically, it runs over UDP and aggregates a dedicated control protocol (RTCP) for guaranteeing quality of service. RTP uses sessions, rather than connections. A session works like a tunnel for data streams, thus, by adding session information to messages, receivers can relate data transported over unreliable channels. Being designed for multimedia, RTP only supports audio and video/image formats, hampering its adoption for other M2M communications.

3.3.3. SOAP

The simple object access protocol (SOAP) offers a specification for exchanging structured data through Web Services. It utilises XML to harmonise the way different applications may be able to understand data. Also, it is suitable for use with any transport protocol. However, the protocol itself, lacks a standardised interaction model on its own which is typically application specific.

3.3.4. WebSocket

WebSocket is a protocol designed for web applications (similar to HTTP), but offering full-duplex communication channels over a single TCP connection (unlike HTTP). The target scenarios are web browsers and web servers, but it can be adopted by any client-server oriented network. The protocol facilitates real-time data transfer, over

TCP, by implementing a custom (yet standardized) way for sending data. WebSocket is a great solution for sending content over the Internet. Its major drawback is that it does not offer a featured service set capable of meeting the needs of industrial M2M communications.

3.3.5. XMPP

The extensible messaging and presence protocol (XMPP) is a M2M communication standard for near-real-time exchange of structured (XML) data between two or more entities. It follows an instant messaging paradigm, similar to email, and uses also a similar addressing nomenclature. The architecture is client-server but, by design, decentralised across several servers. Thus, it offers sufficient flexibility to be adapted to different topologies (e.g. publish-subscribe). The XMPP open standard provides security (authentication and encryption), however it does not provide quality of service. The original transport protocol used with XMPP is TCP. Because TCP ports, used by XMPP, may be blocked by firewalls, some recent implementations have been using XMPP over HTTP or WebSocket.

3.4. Security

In computer networks, the security field is vast and complex. In the scope of this survey, only a brief overview on the major topics, such as authentication, authorisation and encryption, is given.

3.4.1. Authentication

Authentication allows entities to confirm their identity in order to gain access to a network resource. This is the basic form of securing data over the network and can be implemented in several manners, such as usernames and passwords, public-private key pairs, certificates, between others.

3.4.2. Authorisation

Authorisation complements authentication by specifying what resources are accessible by which entities. Typically, the host of the resource keeps a list of authorised users, and manages accesses according to their authentication information.

3.4.3. Encryption

Encryption is the process of encoding information in a way that only authorised entities can read it. Encryption uses cryptographic algorithms to scramble data to be unreadable by unintended entities. A secret passcode must then be used to decrypt the original information.

3.5. Quality of Service

Quality of service (QoS) refers to the overall performance of a communication channel. Different application may consider different performance metrics, such as low latency, high throughput or low error. However, much of these metrics have interdependencies, making it difficult to obtain the absolute best of them all, at the same time. Consequently, QoS mechanisms typically offer a way to configure and prioritise such metrics to better adapt to the real necessities of the applications and respective environments. Some of these metrics are listed below:

3.5.1. Throughput

Throughput measures the rate at which data can be delivered to a destination in a network. This is directly affected by the number of entities exchanging data simultaneously and the capacity of the network for caching data. A low throughput can hamper scalability. QoS may impose mechanisms to schedule and prioritise messages in order to improve throughput.

3.5.2. Dropped packets

In low quality or constrained networks, the number of undelivered packets may be too high, resulting in large data loss. QoS mechanisms may compensate this loss by requesting retransmission of packets, at the cost of higher transmission delays.

3.5.3. Errors

Noise and interference in the communication medium may cause the corruption of data. QoS mechanisms may incorporate error detection by applying checksum calculations, and request retransmission of corrupted packets.

3.5.4. Latency

Latency measures the time used by a packet to reach its destination. High latency values are often undesirable and may be a result of different causes, such as packet collisions, retransmissions, long queues, etc.

3.5.5. Jitter

Jitter measures the difference between the delays observed for packets reaching a given destination. This measure is more important when fragmented data have to be regrouped, at a fixed rate, on the destination, such as audio/video streams. QoS may balance jitter by managing the way packets are queue along the network.

3.5.6. Order of delivery

In connectionless communications, data packets may often reach the destination out of order. QoS can offer additional protocols for rearranging packets in the most appropriate way for the application at hand.

3.6. Service Discovery

In M2M communication, sometimes it is desirable that devices (especially small ones) do not have to keep information about the location (address) of services present on the network. For this, there are several protocols for discovering services over a given network.

3.6.1. DNS-SD

DNS Service Discovery (DNS-SD) is used when the network provides a DNS service. This protocol allows clients to execute queries to the DNS server in order to retrieve a list of additional services offered. However, in this situation, the location of the DNS itself must be known. In order to circumvent this fact, DNS-SD may be used with multicast, that is, sending requests to a well-known multicast address, so service hosts can capture such requests and reply with their service addresses.

3.6.2. DHCP

The dynamic host configuration protocol (DHCP) is a standard for IP networks that uses a centralised server (DHCP server) to control and dynamically distribute network configuration parameters. DHCP allows for automatically configuration of devices when they connect with the network, including addresses and the locations of network services. The discovery of the DHCP server is made by broadcasting messages using the destination address 255.255.255.255 (IP broadcast address).

3.6.3. UPnP

The Universal Plug and Play protocol (UPnP) permits devices to seamlessly discover themselves. However, UPnP assumes that the network runs HTTP, SOAP and XML on top of IP. Thereafter, services are advertised (broadcasted), accordingly to a known description model, to specific control points that are listening for such advertisements.

4. Evaluation of current service bus technology

This section offers a more detailed view on current service bus technology that provides complete solutions for machine-to-machine communication. For this survey, the following standards were addressed:

- CoAP (RFC 7252) – Constrained Application Protocol;
- DDS^{TM} – Data Distribution Service;
- MQTT (ISO/IEC PRF 20922) – MQ Telemetry Transport;
- OPC UA (IEC 62541) – OPC Unified Architecture.

The choice of these solutions resulted from a preliminary study of the standards that better relate with the requirements and necessities of the MSB. This study can be consulted in [3].

4.1. CoAP

The Constrained Application Protocol (CoAP) is a collection of open-standard protocols that forms a complete network stack designed for the communication needs of simple, low-powered and constrained devices. It focuses on the remote M2M communication in the fields of smart energy and building automation. The protocol was specified by the Internet Engineering Task Force (IETF), Constrained RESTful environments (CoRE), Working Group and is available under the reference RFC 7252.

4.1.1. Architecture

The protocol follows a client-server architecture, based on the REST paradigm. For instance, interaction occurs through GET, PUT, POST and DELETE methods. Additionally, CoAP provides an OBSERVE method, which allows to retrieve a representation of a resource and keep this representation updated by the server over a period of time. The requirements that led to the design of the architecture for CoAP may be summarised as follows [4]:

- Web protocol fulfilling M2M requirements in constrained environments;
- UDP binding with optional reliability supporting unicast and multicast requests;
- Asynchronous message exchanges;
- Low header overhead and parsing complexity;
- URI and Content-type support;
- Simple proxy and caching capabilities;
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP;
- Security binding to Datagram Transport Layer Security (DTLS).

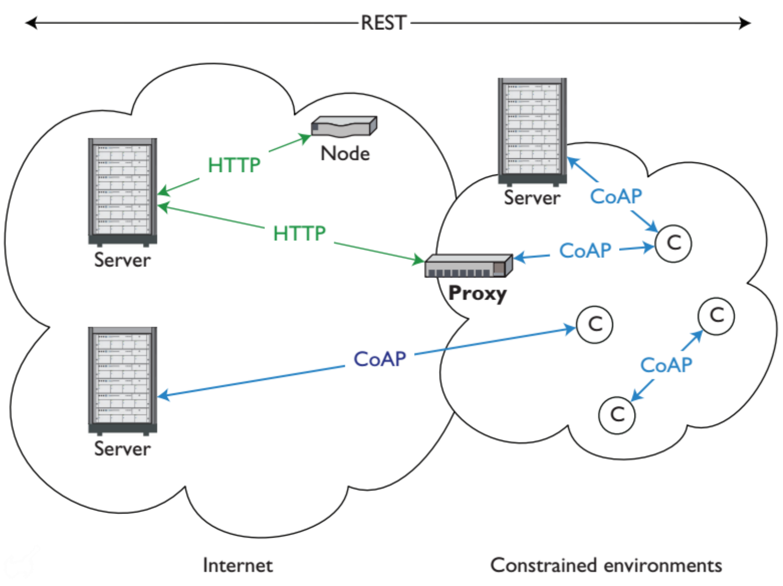


Figure 3 - CoAP and HTTP integration for communication between traditional Internet and constraint environments [5].

Figure 3 shows an architecture proposition to integrate networks operating in constrained environments (using CoAP) with the traditional Internet (using HTTP). This is possible because CoAP and HTTP have similar protocol stacks (Figure 4). Consequently, both protocols can be seamlessly mapped between each other in a proxy.

Payload	**Payload**
HTTP	**CoAP**
TCP	**UDP**
IP	**IP**
Ethernet link	**Constrained link**

Figure 4 - HTTP (left) and CoAP (right) protocol stacks.

As abovementioned, the CoAP stack is very similar to HTTP, however it is less complex. CoAP networks are designed to operate over constrained links, meaning that bitrate and bandwidth can be significantly low, read/write caches tend to be smaller, increasing the risk of packages being lost, and power is also limited. These aspects require exchanged messages to be as small as possible. CoAP uses the Internet Protocol (IP) for addressing devices in the network, maintaining full compatibility with HTTP, however it incorporates UDP for transport. By using UDP, the header size of messages can be significantly reduced[3], when compared with TCP[4].

[3] In CoAP, the header size of messages is typically between 10 and 20 bytes long, with 4 fixed bytes and the remaining reserved for optional fields.

[4] TCP message headers have a minimum size of 20 bytes and can reach up to 60 bytes (with optional fields).

This allows for fast retransmission of lost packages. The CoAP layer defines the messaging specifications, including the four types of request methods (GET, PUT, POST and DELETE) and the response codes (Success 2.xx, Client Error 4.xx and Server Error 5.xx). Finally, the payload is where data values are transmitted. While there is not a limit for payload size, the specification recommends that payload size should be kept small enough to avoid IP packet fragmentation[5].

4.1.2. Licensing and system integration

The list of current CoAP implementations is vast, mostly open source, and covers a diverse set of programming languages. For this survey, only the ones with better support, active development and documentation, was considered, as seen in below:

Table 1 - List of CoAP implementations.

Name	Language	License	Client	Server	Proxy
Californium	Java	Open-source (EPL/EDL)	Yes	Yes	Yes
cantcoap	C/C++	Open-source (BSD)	Yes	Yes	No
CoAP.NET	C#	Open-source (BSD)	Yes	Yes	Yes
CoAPSharp	C#	Open-source (LGPL)	Yes	Yes	No
CoAPthon	Python	Open-source (BSD)	Yes	Yes	Yes
FreeCoAP	C	Open-source (BSD)	Yes	Yes	Yes
libcoap	C	Open-source (BSD/GPL)	Yes	Yes	No
node-coap	JavaScript	Open-source (MIT)	Yes	Yes	Yes

Interoperability tests have been made to CoAP in plug-test events. A study [6] with about near 3000 tests of the protocol, across different vendor's devices, has shown the following results:

Table 2 - Results of the interoperability tests for CoAP [6].

Test	Nr. of tests	Non-Interoperable
CoAP Methods	2798	166 (5.9%)
Link Format	77	6 (7.8%)
Block Transfer	112	15 (13.4%)
Observe Option	94	4(4.3%)
Total	**3081**	**191(6-2%)**

4.1.3. Services and utilities provided

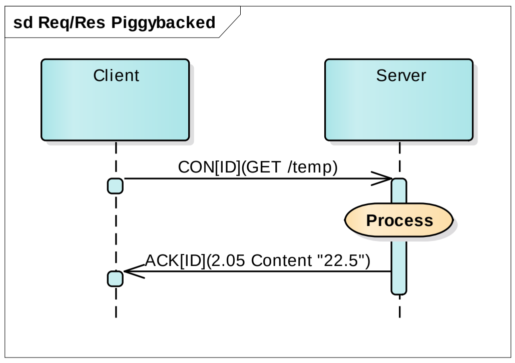
This subsection offers an overview on the major features and services provided by the CoAP solution.

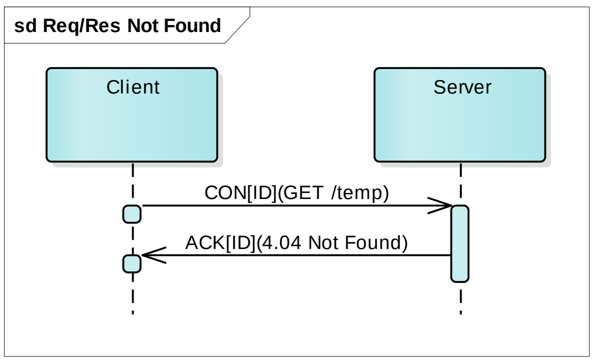
4.1.3.1. Messaging and data model

As seen in the architecture section, the way CoAP defines messaging is very similar with HTTP: a client requests a server to add, retrieve, change or delete data, in a request/response. Messages are typical small in size (~1024 bytes) and are

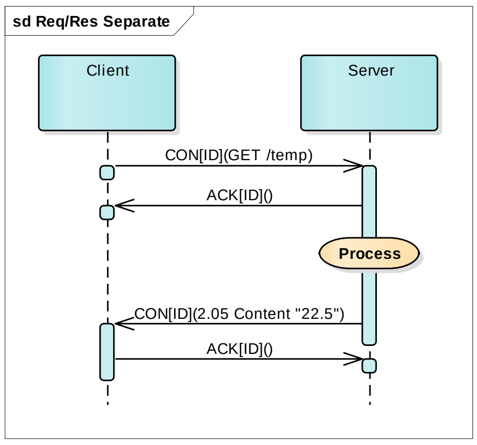
[5] According to the CoAP specification, 1024 bytes is the recommended payload size.

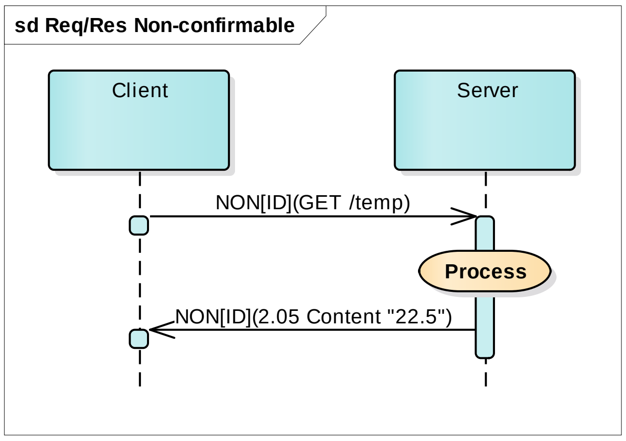
composed of a header and payload. Also, messages can be acknowledged or not but, by design, are asynchronous. As CoAP runs over UDP transport, it supports the use of multicast IP addresses. Figure 5 shows three typical request/response model used with CoAP networks.





a) b)





c) d)

Figure 5 - Possible types of CoAP messaging models.
a) and b) Confirmable request with piggybacked response. c) Confirmable request with separate response. d) Non-confirmable request and response.

Each request may be carried in a Confirmable ('CON') or in a Non-Confirmable ('NON') message. The first means that the message must be acknowledged by the server, while the second does not. If the server is able to process the request (from a CON message) immediately, it can send the response within the Acknowledgement ('ACK') message (Figure 5a, "piggybacked response"). Otherwise, it sends an empty ACK before starting to process the response (Figure 5c). The decision of "piggybacking", or not, a response, resides solely on the server's side, meaning that clients must expect either case when sending a request. If the server cannot handle the request, it replies with an error code (Figure 5b, e.g. 4.04 "Not Found"). In the case of Non-Confirmable requests, the response is always sent separately (Figure 5d). Responses are matched with requests by means of a client-generated "token" that must be echoed by the server in a resulting response.

The CoAP specifications also define the support for any type of data models, which is how data is encoded in the payload of CoAP messages. For instance, data can be exchanged in XML, JSON or any other format. In order to determine if a given server supports a given data format, a special option (Accept) can be used in requests to indicate a desirable format. If the format is not supported, the server will reply with an error code 4.06 "Not Acceptable".

4.1.3.2. Quality of service

Reliability:

CoAP messages are transmitted over unreliable datagrams. However, the protocol specifies that both reliable and unreliable communication is possible. This can be achieved by marking each message as Confirmable ('CON') in the message header options. CON messages are retransmitted at fixed time intervals until an Acknowledgement message ('ACK') is received. If the server receives a CON message but is not able to process it, a Reset message ('RST') is transmitted back to the client, instead of an 'ACK'. If a message does not require reliable transmission, then a Non-confirmable message ('NON') is sent.

Duplication:

When using asynchronous request/response data, one problem that can arise in networks using CoAP is duplication of messages. In order to avoid this issue, every message is sent with a 16-bit unique ID.

Congestion control:

Currently, CoAP specifies a simple congestion control technique that consists of limiting the number of simultaneous outstanding interactions (i.e., retransmission of non-acknowledged / non-replied requests). Further optimisations on congestion control are planned for the future.

Caching:

In order to optimise network load, avoid congestion or reduce the response time, CoAP endpoints may keep responses to further reutilise them with subsequent, similar requests. The validity of cached responses is guaranteed by a "freshness" mechanism that defines a time-limit for the response itself. Requests may also carry a "NoCache" option meaning that the expected response cannot be a cached one.

4.1.3.3. Proxies

Proxies are CoAP endpoints that can be tasked by CoAP clients to perform requests on their behalf, enabling, for instance, to better balance the traffic on the network and optimise resources and energy consumption.

Forward-proxy:

A forward-proxy can be explicitly selected by clients to send requests through the proxy (not as the client), making the endpoints fully aware that messages are being exchanged through a proxy.

Reverse-proxy:

A reverse-proxy makes the purpose of a server and can be used to retrieve cached responses. Also, it can be used to retrieve resources available on the network, without directly contacting with the servers (through discovery service).

Cross-Proxy:

A cross-proxy enables mapping and translation between CoAP and other protocols.

4.1.3.4. Discovery

The discovery feature allows determining the services offered by CoAP servers, that is, exposing the URI's of resources in the namespace of the servers. Resource discovery queries must always be done directly to the servers that offer services. This can be done directly if a client knows the default server URI, or alternatively by using multicast CoAP addresses to find all CoAP servers on a network. The discovery service should be offered through a default known port (5683, or 5684 for secured communication).

4.1.3.5. Security

To enable secured communications, CoAP can be used on top of the Datagram Transport Layer Security (DTLS), which defines four security options: none (NoSec); pre-shared keys that determine authorised peer-to-peer communication at the node level (PreSharedKey); public keys that authorise communication to a set of nodes (RawPublicKey); and certificates that can be used instead of public keys to sign message content (Certificate).

4.1.3.6. Block

CoAP is a protocol designed for the exchange of small chunks of data. However, if larger messages are to be sent, they may get fragmented in the transport layer. This means that CoAP endpoints must be prepared to receive multiple, unordered parcels of the same message and join them in the right order. To avoid this demanding task, CoAP messages may use the Block option to split information in a series of successive request/response pairs that can be handled separately and independently from each other.

4.1.3.7. Observe (push notifications)

While the typical request/response model implies that data exchange must be always initiated by clients, CoAP also predicts the need to support server-side notifications to clients (push). This is done through a special option (Observe) used in requests to indicate the interest of clients in receiving further updates on a specific resource. After receiving an Observable request, the server automatically sends asynchronous notification messages, each time the resource changes.

4.1.4. CoAP solution comparison chart

The following table provides a comparison of the most popular CoAP SDK's in terms of current functionality. It is important to emphasize that some of the implementations are currently under active development and still missing some of the functionality covered by the standard specifications.

Table 3 - CoAP Solution Comparison Chart

	Californium	cantcoap	CoAP.NET	CoAPSharp	CoAPthon	FreeCoAP	libcoap
Resource Discovery	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Security	Yes	Yes	No	No	No	Yes	Yes
Observe Option	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Block Transfer	Yes	Yes	Yes	Yes	Yes	Yes	Yes
XML support	Yes	Yes	No	No	No	No	Yes
JSON support	Yes	Yes	No	Limited	No	No	Yes

4.1.5. Performance

Performance tests were made in order to stress payload interchange between a client and a server and verify its impact on latency timings. The CoAP protocol was tested using Californium (Cf), an open source Java framework with the protocol implementation, provided by the Eclipse Foundation. The server and client specifications of the test are:

Table 4 - Specifications of hardware used for testing the performance of CoAP.

Specifications	Client	Server
CPU	Intel core i7 3630QM @3.2GHz	Intel core i5 4300u @1.9GHz
RAM	16 GB	8 GB
OS	Ubuntu 14.04	Ubuntu 14.04
Network switch	Draytek Vigor3900	

The test was made within the local network. A graph with the average latency versus message payload can be seen in Figure 6.

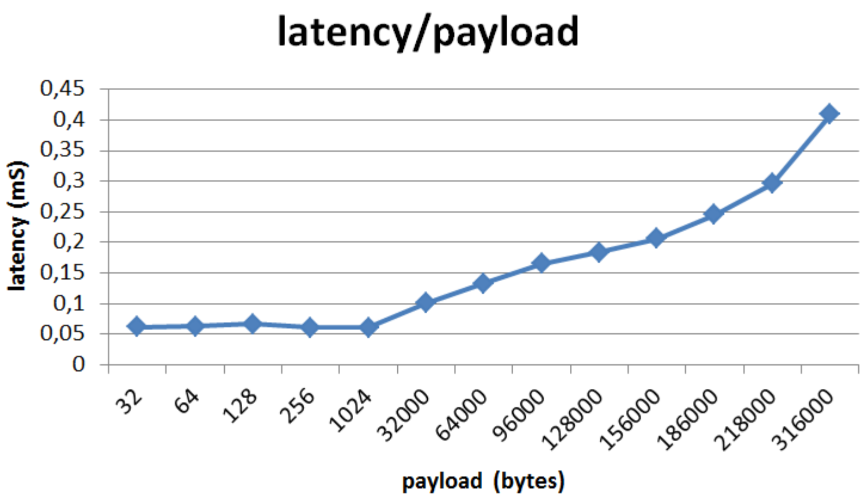
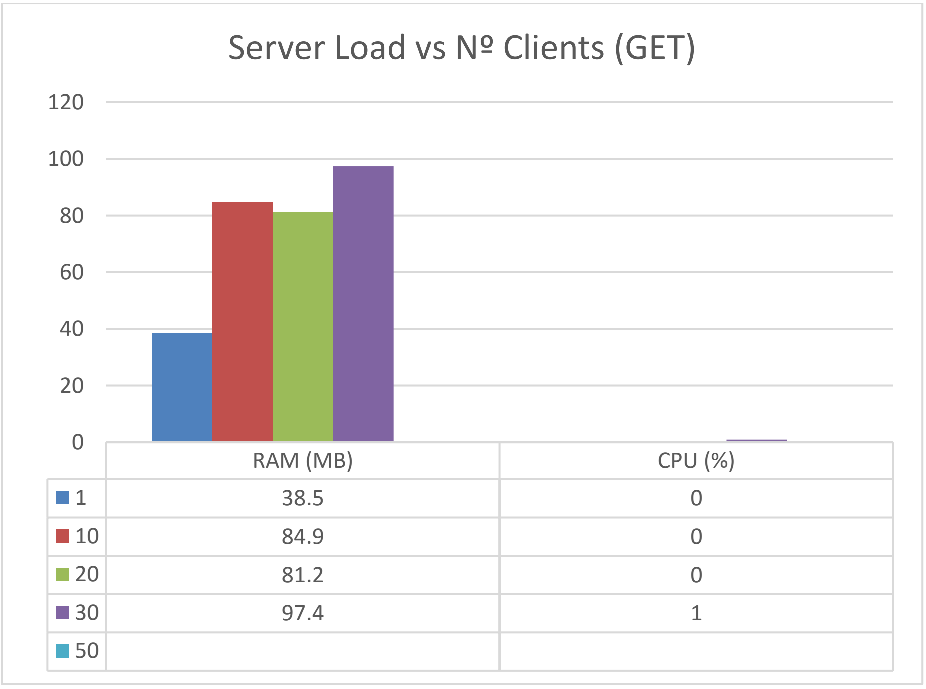


Figure 6 - CoAP protocol performance (latency) vs. payload size.

From Figure 6, it can be observed that the average latency only gets affected when the payload of messages payload exceeds 1024 bytes. The approximately linear increase in latency, from 32 to 1024 bytes, is verified due to the CoAP specification itself which provides an upper bound to the message size of 1280 bytes, to avoid undesirable packet fragmentation.

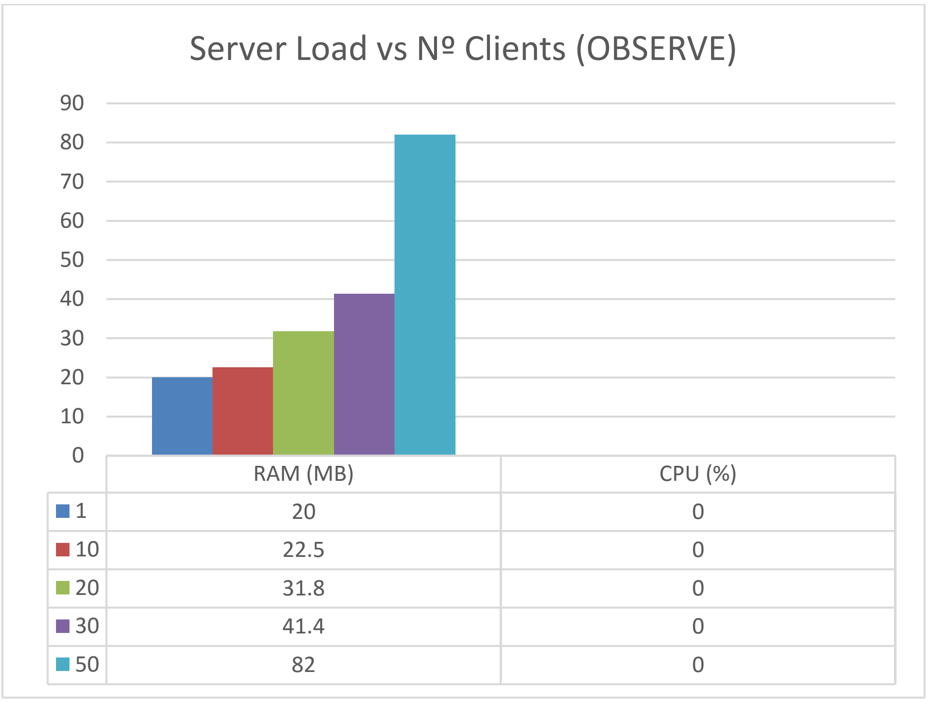
Scalability

The scalability tests involved requesting the GET method (Figure 7) from the server to retrieve a value and subscribing the OBSERVE method (Figure 8) to receive notifications from the server on value changes. In a) and b), the requests and value updates were made at a rate of 1Hz, respectively.



	RAM (MB)	CPU (%)
🟦 1	38.5	0
🟥 10	84.9	0
🟩 20	81.2	0
🟪 30	97.4	1
🟦 50		

Figure 7 - CoAP scalability test results (GET).



	RAM (MB)	CPU (%)
🟦 1	20	0
🟥 10	22.5	0
🟩 20	31.8	0
🟪 30	41.4	0
🟦 50	82	0

Figure 8 - CoAP scalability test results (OBSERVE).

From the realized tests, only the one which had 50 clients simultaneously requesting the GET method from the server failed, because the connection with a portion of the clients was losing packets. In terms of system load (CPU and Memory), the results from the performed tests proved the suitableness of CoAP as a communication middleware in constrained devices.

4.1.6. CoAP solution analysis

CoAP was designed for small, low powered and constrained devices operating in loss-prone networks. Taking this into consideration, it is also expected that the same devices are meant to exchange small amounts of data at a low rate. Making these assumptions, CoAP may be the ideal solution, given its simplicity and interoperability. Also, even though being a request/response protocol, it allows for server side notifications, meaning that bidirectional communication is possible, which is mandatory for industrial devices (e.g., a monitoring system must know about abnormal situations without the need to constantly having to request devices about their statuses).

Regarding the different solutions that currently exist to implement CoAP enabled devices and components, there is a wide range of choice, already with commercial friendly, open source and full featured software platforms (e.g., **Californium**, **cantcoap** and **libcoap**). Given the "lightweight-by-design" feature of CoAP, there is not much distinction, in terms of performance, in the different solutions. However, one must consider that Java implementations require the additional resources to run a Java Virtual Machine (JVM).

Although showing good performance, CoAP is not meant for fast and reliable communications in large networks. This is mainly due to the simple congestion control mechanisms that it employs and that can lead to a substantial amount of packet collision and loss of data when the network becomes overloaded. In such case, a complex planning for load balancing is needed, which may incur in high deployment and commissioning costs that could be avoided by adopting other protocol.

4.2. DDS

The Data Distribution Service (DDS^{TM}) is a middleware protocol and API standard for data-centric connectivity, from the Object Management Group (OMG). It is aimed to integrate the components of a system together, while providing low-latency data connectivity, extreme reliability and a scalable architecture.

It enables the multiple components of a system to communicate and share information more easily. It simplifies the development of distributed systems by abstracting the mechanics of passing information between applications and systems.

4.2.1. Architecture

DDS is a publish/subscribe service. Data samples are sent through the system for conceptual "Data Objects". The association of a publisher and a DataWriter sends samples to one or more associations of a DataReaders and Subscribers. The basic components of the system (Figure 9) are:

1) Topics,
2) Publishers,
3) Subscribers,
4) DataWriters and
5) DataReaders.

Their interaction paradigm is the following:

- Topics are information placeholders for a single data type;
- Publishers control and restrict the data flow from DataWriters;
- Subscribers control and restrict the data flow from DataReaders;
- DataWriters create samples of a single application data type;
- DataReaders receive samples of a single application data type;
- A Publisher can have one or more DataWriters;
- A Subscriber can have one or more DataReaders;
- A DataWriter has a single Topic;
- A DataReader has a single Topic;
- A Topic can have many DataReaders and DataWriters;
- A publication can have one or more associated subscriptions;
- A subscription can have one or more associated publications

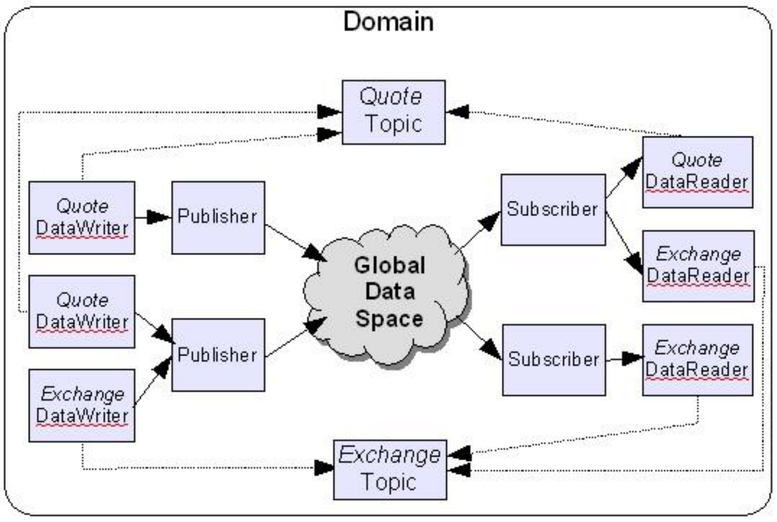


Figure 9 - Conceptual view of DDS Architecture [7].

4.2.1.1. Transport Layer

The transport possibilities for the protocol are presented in Figure 10. To promote interoperability, the OMG has a specification to enable interoperability amongst DDS implementations. It is known as DDS Interoperability Protocol (DDSI). This wire protocol uses the OMG Common Data Representation (CDR) to encapsulate data in a neutral way, such as it can be transferred over the network. DDS implementations can support multiple transport protocols.
In order to accommodate the existing legacy systems, DDS implementations separate the transport from the higher level protocols, by means of Extensible Transport Frameworks and APIs.

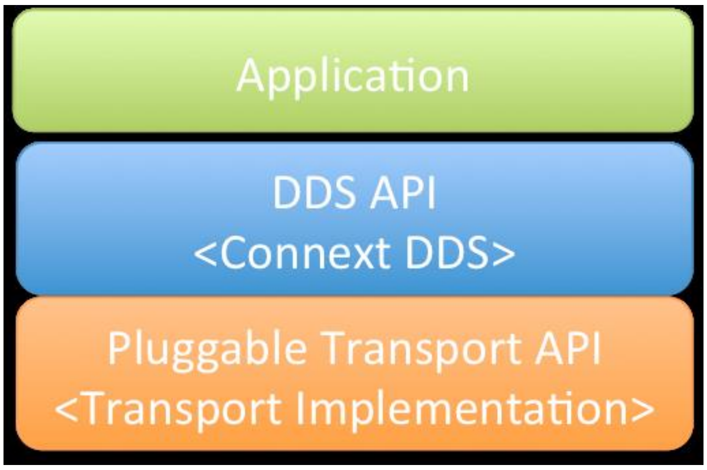


Figure 10 - Extensible Transport aspects of DDS [8].

4.2.1.2. Security

Securing DDS means providing:

a) Confidentiality of the data samples;
b) Integrity of the data samples and the respective messages;
c) Authentication and authorization of DDS writers and readers;
d) Non-repudiation of data.

DDS ensures such requirements by implementing HMAC [9] (Hash-based Message Authentication Code) cryptography.

4.2.1.3. Conceptual Model

The overall conceptual module is shown in Figure 11. All the main communication objects (specialized **Entities**) follow the unified patterns below:

- Supporting QoS, which provides a generic mechanism for the application to control the behaviour of the Service and tailor it to its needs. Each **Entity** supports its own specialised kind of QoS policies.
- Accepting a **Listener**. Listeners provide a generic mechanism for the middleware to notify the application of relevant asynchronous events (data arrival, violation of a QoS setting, etc). Each entity supports its own specialised kind of listener.
- Accepting a **StatusCondition**. These conditions provide support for two alternate communication styles between the middleware and the application: 1) wait-based style and 2) notification-based.

All these entities are attached to a **Domain Participant.** A domain participant represents the local group of entities of an application, in a domain. A *domain* is a distributed concept that links all the applications able to communicate with each other (i.e., only the publishers and subscribers within the same *domain* may interact).

The **Domain Entity** is an intermediate object whose purpose is to impose that a **Domain Participant** cannot contain other domain participants.

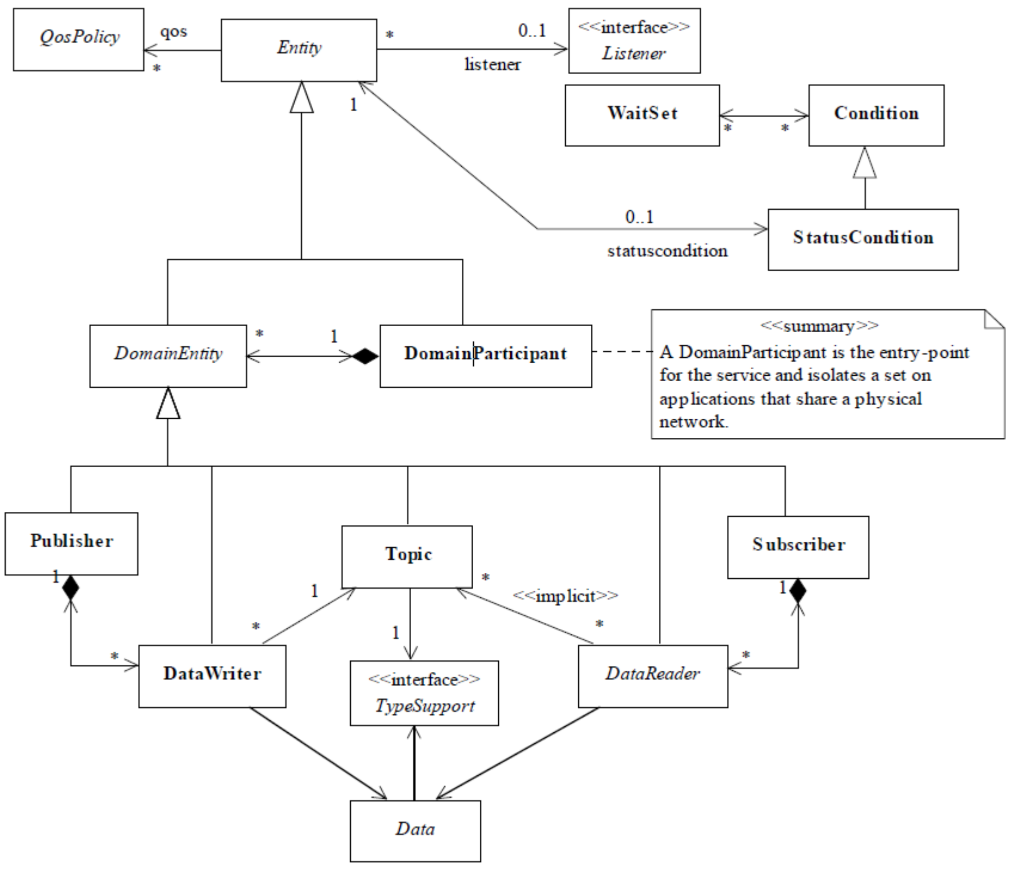


Figure 11 - DDS conceptual model. Adapted from [10]

4.2.1.4. Data Types

The information flows in between publishers and subscribers of a DDS domain, through topics. A topic defines a domain-wide information's class and is defined by means of a (name, type, QoS) tuple:

- Name: Identifies the topic within the domain;
- Type: Programming language type associated with the topic. Types are extensible and evolvable.
- QoS: A collection of policies that express the properties of the topic (reliability, persistence, etc.)

DDS handles the following transfer tasks: message addressing, data marshalling and de-marshalling, delivery, flow control and retries. The topics support the following data types:

- Built-in types provided by existing middleware;
- Structures specified in OMG IDL;
- DDS-specific XML format.

Also, Extensible and Dynamic Topic Types for DDS (DDS-XTypes) provides support to communication where topics are defined with specific data structures. DDS specifies that any node can be a publisher, subscriber, or both simultaneously.

4.2.2. Base Services

DDS base services ensure a distributed (publish/subscribe) message exchange and allow to track where each message is coming from. DDS also allows the user to specify QoS (Quality of Service) parameters to configure discovery and behaviour mechanisms up-front.

Particularly, DDS enables predictable and deterministic data sharing between applications through the use of QoS specifications which enable control timing, communication channel priority and resource utilisation.

4.2.3. Licensing

DDS implementations are provided by three main vendors, with commercial and open source licensing:

Table 5 - DDS Vendors implementation languages and licenses.

Vendor	Product	Language	License
OCI	OpenDDS	C++	Open Source, No licensing fees
Real-Time Innovations, Inc	Connext DDS	Java, C, C++	Open Community Source and Commercial Licenses
PrismTech	Vortex OpenSplice	Java, C, C++, C#	Open Source LGPLv3, Commercial License
PrismTech	Vortex OpenSplice Community Edition	Java, C, C++, C#	Open Source LGPLv3
Twinoaks	coreDX	Java, C, C++, C#	Evaluation, Development and Deployment Licenses

4.2.4. System integration

DDS is interoperable between vendors. DDS interoperability specification allows each vendor to publish and subscribe to each other's topics [11]. Thus, it is possible to design a system where DDS participants were developed with different implementations.

4.2.5. Provided Features

4.2.5.1. Quality of Service

The sharing of data using DDS can be done with flexible Quality of Service (QoS) specifications. These QoS specifications implement services which provide reliability, system health (liveliness) and security to the DDS applications.

In a real system, not every endpoint needs every item in the local store. DDS is smart about sending just what it needs. If messages don't always reach their destinations, the middleware implements reliability where needed. When systems change, the middleware dynamically figures out where to send which data, and intelligently informs participants about the changes. If the total size of data is very large, DDS intelligently filters and sends only the data each endpoint really needs. When updates need to be fast, DDS sends multicast messages to update many remote applications at once. As data formats evolve, DDS keeps track of the versions used by various parts of the system and automatically translates. For security-critical applications, DDS controls access, enforces data flow paths, and encrypts data "on-the-fly".

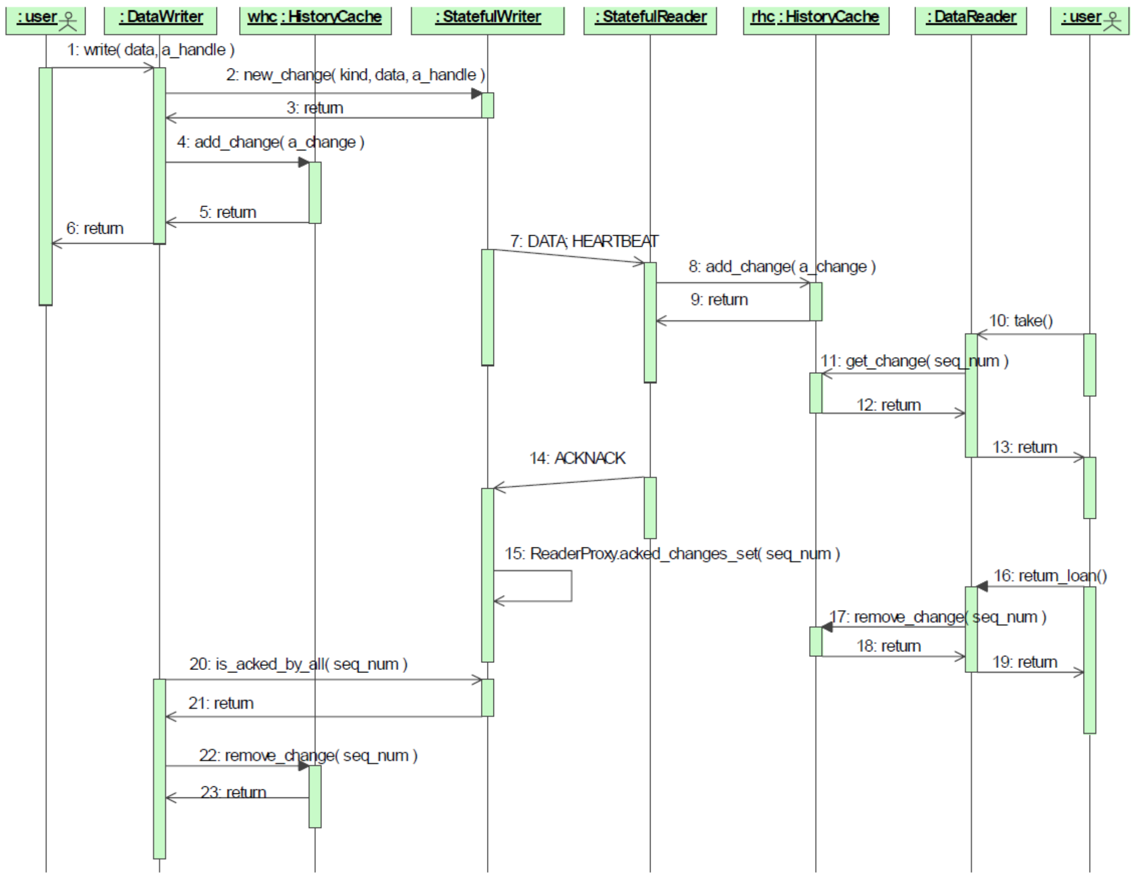
4.2.5.2. Real-Time Publish-Subscribe

The Real-Time Publish Subscribe (RTPS) protocol is the interoperability protocol used to allow multi-vendor DDS implementations to communicate between them. This protocol features Performance and QoS properties, which enable best-effort and reliable publish-subscribe communications for real-time applications, using standard IP networks.

The RTPS specification does not mandate a single implementation. Instead, it defines a minimum requirements list for interoperability and, then provides two reference implementations: Stateful and Stateless Implementations. Also, the protocol behaviour depends on the RELIABILITY QoS and keyed topics (reserved topics), which should be set to specific combinations.

- **Stateless Reference Implementation:** optimized for scalability, this Implementation keeps virtually no state on remote entities and, therefore scales very well with large systems. The trade-off for improved scalability and reduced memory usage is requiring additional bandwidth usage. This Implementation is ideally suited for best-effort communication over multicast.
- **Stateful Reference Implementation:** maintains full state on remote entities. This approach minimizes bandwidth usage, therefore requiring more memory, and may also imply a reduced scalability capability. Compared with the Stateless Reference Implementation, this one can guarantee strict reliable communication.

An example behaviour of RTPS, using the Stateful Reference Implementation (most suitable within the OpenMOS requirements), is presented in Figure 12.



: user
: DataWriter
whc : HistoryCache
: StatefulWriter
: StatefulReader
rhc : HistoryCache
: DataReader
: user
1: write(data, a_handle)
2: new_change(kind, data, a_handle)
3: return
4: add_change(a_change)
5: return
6: return
7: DATA, HEARTBEAT
8: add_change(a_change)
9: return
10: take()
11: get_change(seq_num)
12: return
13: return
14: ACKNACK
15: ReaderProxy.acked_changes_set(seq_num)
16: return_loan()
17: remove_change(seq_num)
18: return
19: return
20: is_acked_by_all(seq_num)
21: return
22: remove_change(seq_num)
23: return

Figure 12 - Example RTPS Behavior. Adopted from [12].

Summarily, the DDS user writes data through the **write** operation on the DDS DataWriter, who will invoke the **new_change** operation on the Writer to create a new CacheChange (uniquely identified). Next, the DDS DataWriter uses the **add_change** operation to store the CacheChange into the RTPS Writer's HistoryCache. Then, the RTPS Writer sends the contents of the CacheChange to the RTPS Reader, and requests acknowledgment by also sending a Heartbeat submessage. The DDS user will then be notified by a DDS mechanism (e.g., a listener) and initiates reading the data. An acknowledgement is then sent by the RTPS Reader indicating that the CacheChange has been placed into the Reader's History Cache.

Finally, the **remove_change** operation is called by the DataWriter, to remove the change associated with a sequence number. In doing this, the DDS DataWriter also takes into account other DDS QoS such as DURABILITY (data persistence).

4.2.5.3. Dynamic Discovery

DDS provides Dynamic Discovery of publishers and subscribers, and in some implementations their Data Writers and Readers. Having such service means the application does not have to acknowledge or configure the endpoints for communications, because they are automatically discovered and managed by DDS.

Because the service may be executed at runtime, DDS enables real "plug-and-play" to its applications.

DDS Dynamic Discovery services go further than discovering endpoints, by actually providing information about each endpoint: if it is publishing data, subscribing data and the type of data being published or subscribed to. It will also discover the publisher's offered communication settings and the subscriber's requested communication settings.

Finally, the discovery mechanism does not need to know or configure IP addresses, or consider the differences in machine architectures. Hence, adding additional participants on any operating system or hardware platform is an easy, almost trivial task.

4.2.5.4. Data Modeling

In order to ease the process of implementing a system which uses DDS as a communication middleware, the DDS providers already include tools (SDKs) that intuitively aid the generation of the complete **domain entities** and interactions of a system. Once a model has been captured, it is used to generate source code. That source code is then compiled into link libraries that can be linked into a user application. Applications access the DDS API by instantiating classes[6] generated from the model and accessing the entities represented in it. Thus, the application will be free to send and receive data directly from the DDS defined entities.
DDS specifies a standardised way of generating model data, hence, models generated in different SDKs may be interchangeable across code generators. The existing SDKs for modelling and generating models feature:

a) UML modelling elements, which are based on the metamodel described in UML Profile For Data Distribution Specification;

b) SDK model files use the standards-based XML Metadata Interchange (XMI) format [13];

c) A user-friendly, UML-based, graphical designer;

d) Automatic code generation tools.

Availability:

- **OpenDDS Modeling SDK** [14]: Open source and free tool to model and generate DDS systems in C++. The SDK consists on a set of Eclipse plug-ins and is also open source (Java). The SDK is built on top of the Eclipse Modeling Framework (EMF) [15]. Because of the universal acceptance of EMF, it is feasible to consider interoperability with other modeling tools

[6] depends on the programming language.

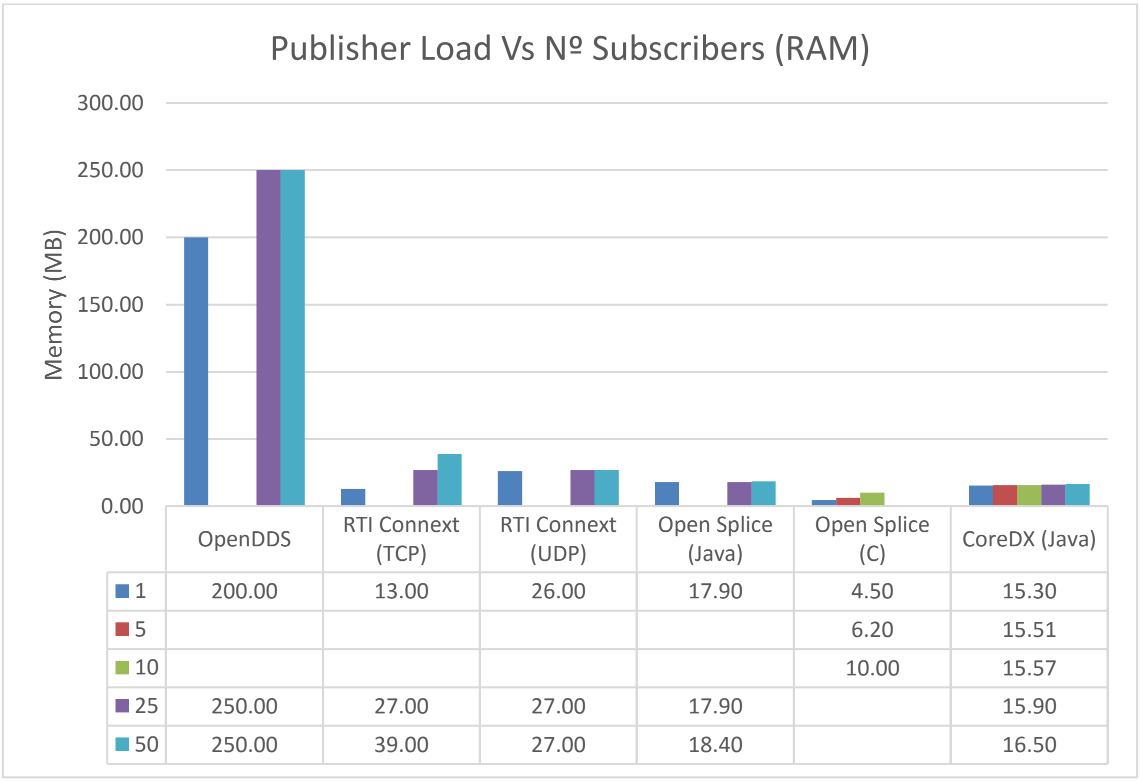
- **Vortex OpenSplice Modeler** [16]: A Commercial version modelling SDK. Similarly to OpenDDS, it is an Eclipse based modelling tool based on DDS specific development. The code generator is compatible with Java and/or C++.

4.2.6. Performance

The previously presented SDK's were evaluated, in terms of performance, according to their scalability and data transfer latency. Tests were made on an Intel Core i7 2.4 GHz laptop, with 6 GB RAM and Windows 7 Professional operating system.

Scalability

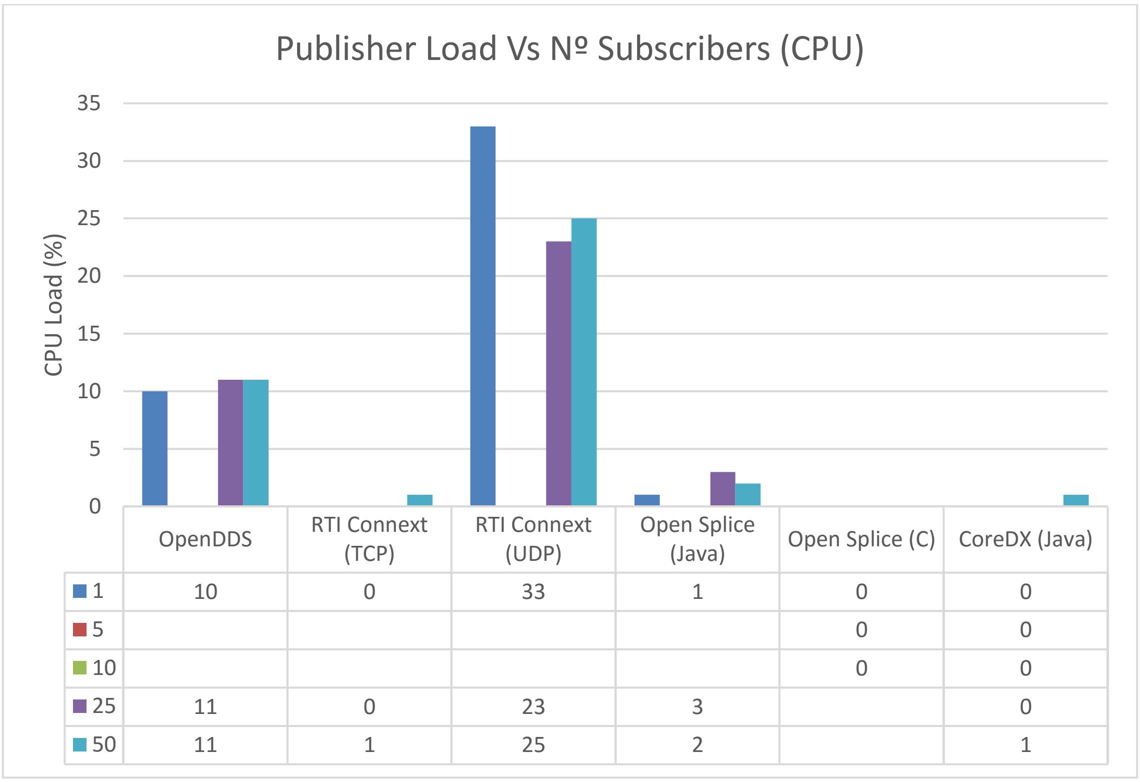
Scalability was tested connecting up to 50[7] subscribers (with a single DataReader) to each publisher (with a single DataWriter), at a publishing rate of 5 Hz. The progression of CPU and Memory load was observed for both the publisher and subscriber processes. All tests were made using TCP as the transport protocol. Only RTI Connext was tested with UDP due to presenting very high memory loads on TCP. Results are shown in the following graphs:



	OpenDDS	RTI Connext (TCP)	RTI Connext (UDP)	Open Splice (Java)	Open Splice (C)	CoreDX (Java)
🟦 1	200.00	13.00	26.00	17.90	4.50	15.30
🟥 5					6.20	15.51
🟩 10					10.00	15.57
🟪 25	250.00	27.00	27.00	17.90		15.90
🟦 50	250.00	39.00	27.00	18.40		16.50

Figure 13 – DDS Publisher load VS nº of Subscribers (RAM)

[7] OpenSplice C implementation is hardcoded to only support up to 10 participants in the same machine, hence the tests only present results up to this value for this implementation.

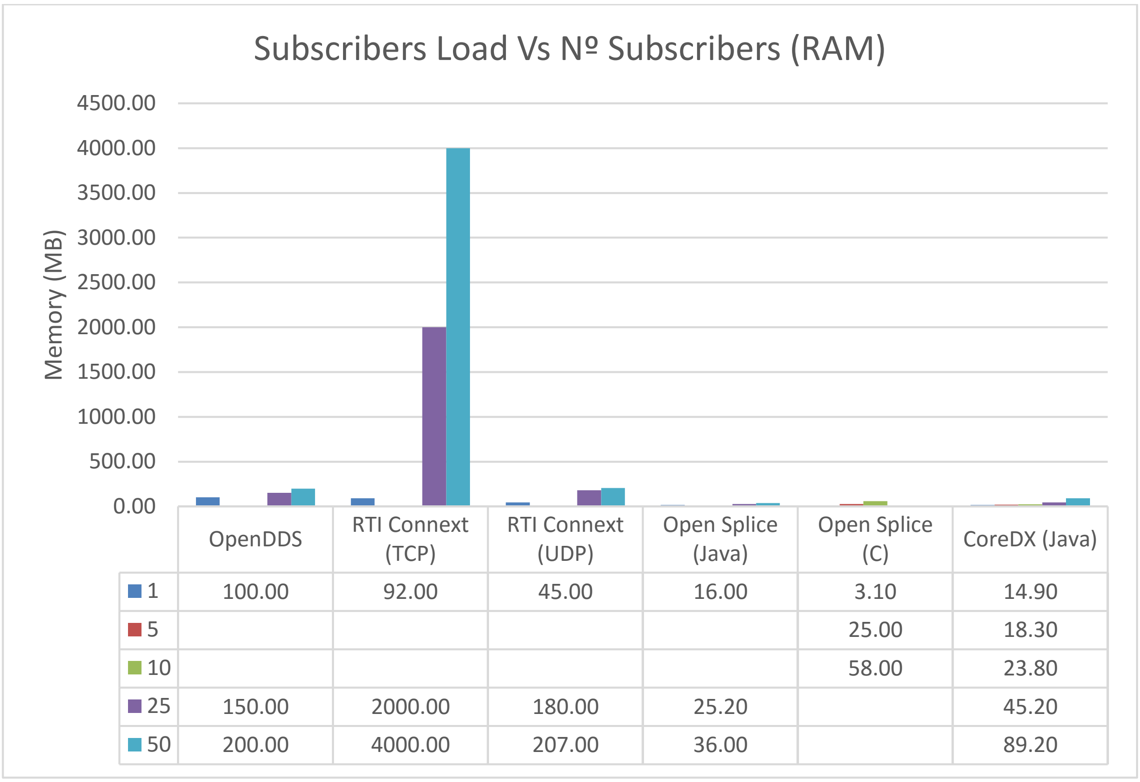


	OpenDDS	RTI Connext (TCP)	RTI Connext (UDP)	Open Splice (Java)	Open Splice (C)	CoreDX (Java)
🟦 1	10	0	33	1	0	0
🟥 5					0	0
🟩 10					0	0
🟪 25	11	0	23	3		0
🟦 50	11	1	25	2		1

Figure 14 – DDS Publisher load VS nº of Subscribers (CPU)

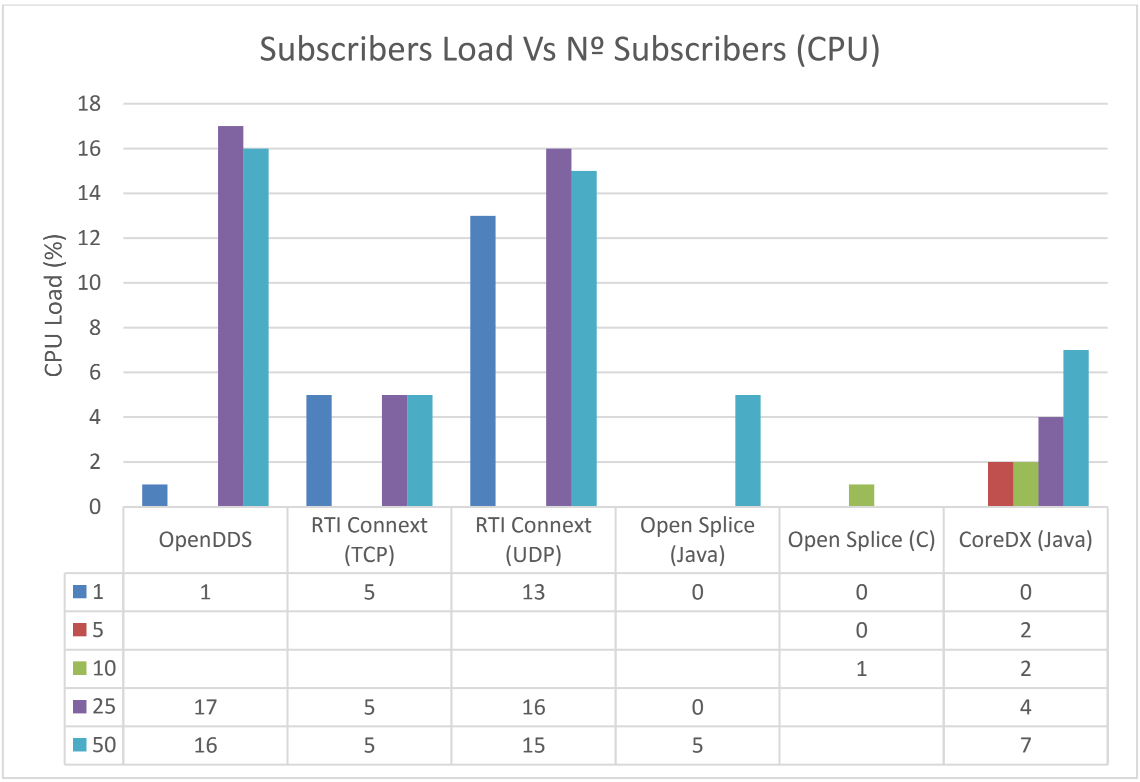
In terms of memory load performance, OpenDDS is clearly behind the other implementations, which share similar load values.

In terms of CPU, OpenDDS and RTI Connext (on UDP) implementations present considerably high CPU load values, whereas the others present a low CPU load (0% to 2%).



	OpenDDS	RTI Connext (TCP)	RTI Connext (UDP)	Open Splice (Java)	Open Splice (C)	CoreDX (Java)
🟦 1	100.00	92.00	45.00	16.00	3.10	14.90
🟥 5					25.00	18.30
🟩 10					58.00	23.80
🟪 25	150.00	2000.00	180.00	25.20		45.20
🟦 50	200.00	4000.00	207.00	36.00		89.20

Figure 15 – DDS Subscribers load VS nº of Subscribers (RAM)



	OpenDDS	RTI Connext (TCP)	RTI Connext (UDP)	Open Splice (Java)	Open Splice (C)	CoreDX (Java)
🟦 1	1	5	13	0	0	0
🟥 5					0	2
🟩 10					1	2
🟪 25	17	5	16	0		4
🟦 50	16	5	15	5		7

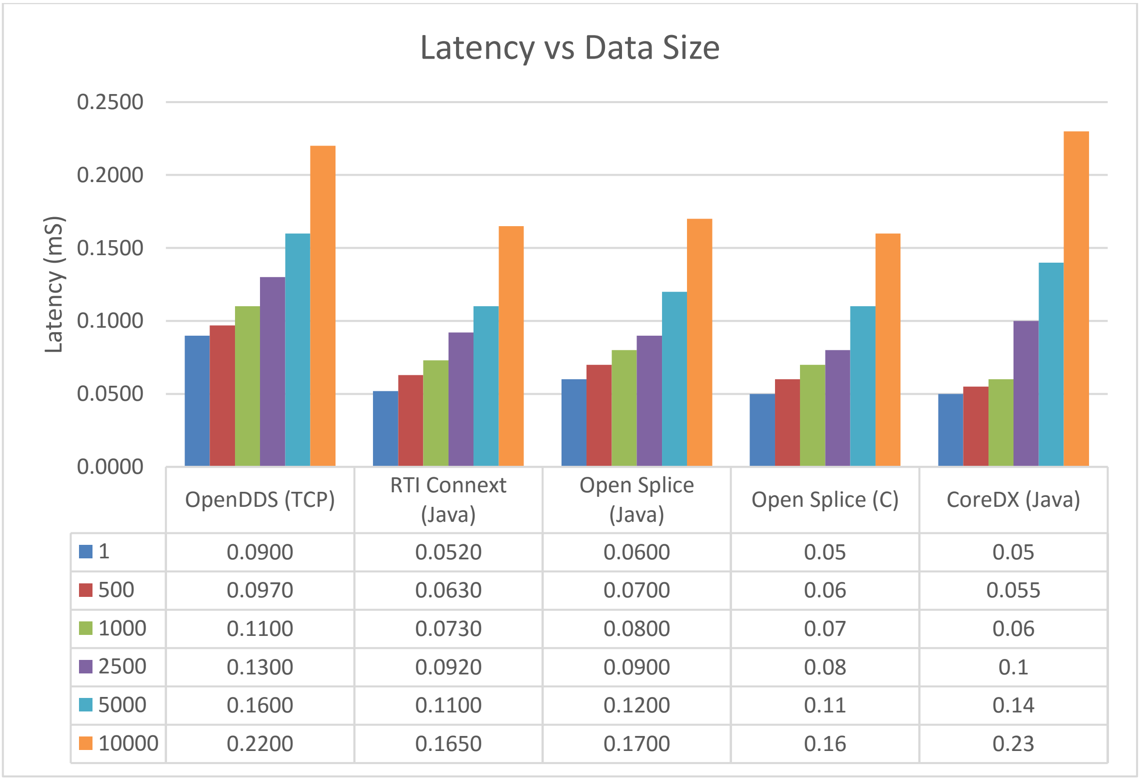
Figure 16 – DDS Subscribers load VS nº of Subscribers (CPU)

In terms of memory load performance, OpenDDS and RTI Connext seem to be the least suitable for constrained devices, consuming a minimum of 90MB per subscriber. However, other implementations such as OpenSplice (Java) require as little as 36MB to simultaneously run 50 subscribers.

In terms of CPU, all Java implementations share the same range of values (between 0% and 10%) except for RTI Connext using UDP, which values are considerably higher compared to using TCP as the transport protocol.

Latency

Latency performance was tested [17] [18] [19] [20] by measuring the time elapsed of data sent by a publisher to reach a subscriber across the network. The tests were performed with a payload up to 10kB, and the results are presented below:



	OpenDDS (TCP)	RTI Connext (Java)	Open Splice (Java)	Open Splice (C)	CoreDX (Java)
🟦 1	0.0900	0.0520	0.0600	0.05	0.05
🟥 500	0.0970	0.0630	0.0700	0.06	0.055
🟩 1000	0.1100	0.0730	0.0800	0.07	0.06
🟪 2500	0.1300	0.0920	0.0900	0.08	0.1
🟦 5000	0.1600	0.1100	0.1200	0.11	0.14
🟧 10000	0.2200	0.1650	0.1700	0.16	0.23

Figure 17 - DDS latency VS data size

Analysing the results, it can be concluded that all the implementations have very similar latency performance. However, CoreDX Java implementation is the one which performs the best when dealing with small payloads (up to 1kB). OpenDDS is the implementation with the least performance from the list.

4.2.7. DDS Solution Analysis

Four SDKs and five implementations were tested for this survey:

OpenDDS	RTI Connext	OpenSplice DDS (Community)	CoreDX
C++	Java	Java and C	Java

The selection of these SDKs were based on their popularity and support at the time of this document. Because the main selling point of DDS is its high performance and speed, the focus of the tests was set in latency and load of the implementations.
Both the open source and commercial DDS SDK's provide the same set of features specified by the OMG. Therefore, the difference between them lays on the ease of implementation, configuration and provided documentation and examples.
Particularly, OpenDDS was the least straightforward SDK to configure its QoS and transport settings. RTI Connext has good documentation, a large and complete set of examples and is easy to configure through XML templates but disappointed by using a bigger amount of Memory and CPU load. Although CoreDX is commercially supported, from the four SDKs it is the least popular in the DDS community. Finally, OpenSplice (Community Open Source) proved to be probably the best option to implement DDS communication due to its broad implementation support (C, C++, Java, C#), the combination of commercial and community support and quality documentation and examples. The only disadvantage of using OpenSplice Community edition is the lack of commercial support from Prismtech. However, the large open source community provides wide support in its forums.

4.3. MQTT

MQTT stands for Message Queuing Telemetry Transport and is a publish-subscribe messaging protocol invented in 1999 by Dr. Andy Stanford-Clark of IBM, and Arlen Nipper of Eurotech (formely Arcom). It is an ISO standard (ISO/IEC PRF 20922) and, since 2014, an OASIS[8] open standard. MQTT was originally designed to be a lightweight messaging mechanism prone to be used in networks with very limited bandwidth. Since its creation, MQTT got several updates mainly because of the crescent demand for IoT solutions, being nowadays still actively used as part of widely spread services, such as Facebook Messenger [21], Amazon IoT [22], or OpenStack [23].

4.3.1. Architecture

The protocol follows a client-server approach, serving as a simple bus for the exchange of binary data. Communication is made over TCP, thus, clients must establish a TCP connection to a central server (broker) to be able to publish or subscribe to data (Figure 18).

[8] OASIS – Organization for the Advancement of Structured Information Standards is nonprofit consortium aimed at the standardisation of specifications for, between others, the Internet of Things (IoT)

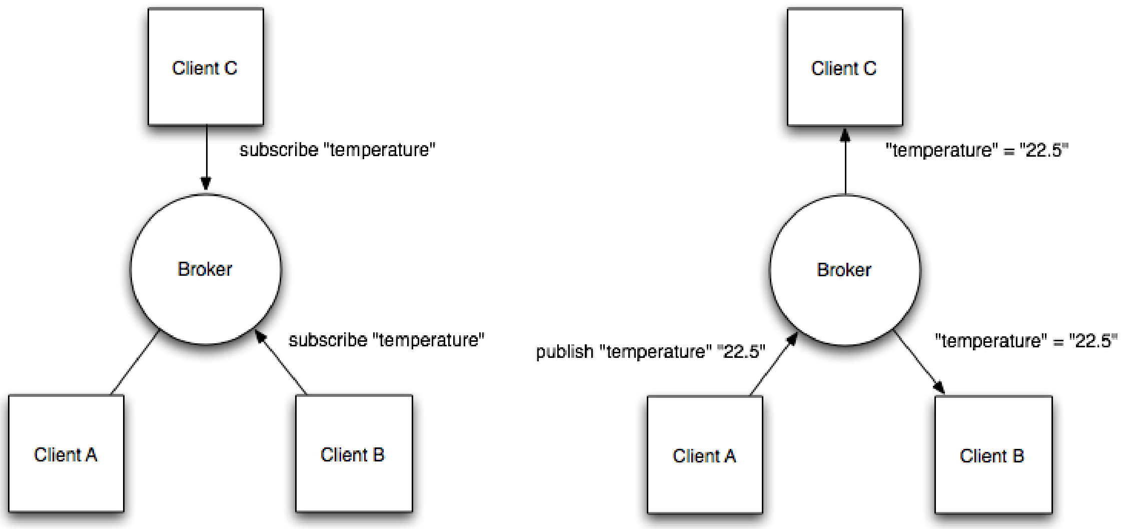


Figure 18 - Example of a network using MQTT. Client A is publishing to topic "temperature" which is being subscribed by clients B and C.

The function of the broker is only to manage the address space of clients and topics, and to provide simple QoS services. The content of messages is opaque to the broker, being responsibility of clients to interpret and parse data.

By being centralised, clients only need to know the address (IP and port) of the MQTT broker, either hard-coded, or through a third-party discovery service. Also, communication is possible either one-to-one, one-to-many, many-to-one or many-to-many, as topic data may be fed or consumed by one or several clients at a time.

4.3.2. Licensing and system integration

The list of current MQTT implementations is vast, mostly open source, and covers a diverse set of programming languages and devices. Being a relatively old and mature protocol, most of the implementations already support all, or at least, most of the specifications of the protocol. For this survey, only the most popular, complete, stable and open-source (commercial friendly) solutions were considered: Eclipse Paho, Moquette and Mosquitto. These implementations should cover all necessities risen by the openMOS project, for what the protocol itself can give.

Table 6 - List of popular MQTT implementations.

Name	Language	License	Client	Broker
Eclipse Paho	Java, Python, C, C#, Embedded C, Javascript	Open-source (BSD)	Yes	Yes[9]
Moquette	Java	Open-source (Apache v2/EPL)	No	Yes
Mosquitto	C/C++	Open-source (EPL/EDL)	Yes	Yes

[9] Not supported in the Java and Javascript versions.

4.3.3. Services and utilities provided

This subsection offers an overview on the major features and services provided by the MQTT solution.

4.3.3.1. Messaging and data model

MQTT control packets:

All data exchange in MQTT is made through control packets (

Figure 19). A control packet always contains a fixed header and, optionally, a variable header and payload. The fixed header has a type field for identifying which type of message is being sent (see Table 7), flags for accessing broker services (depending on the message type), and a remaining length field for encoding the size of data included in the variable header and the payload.

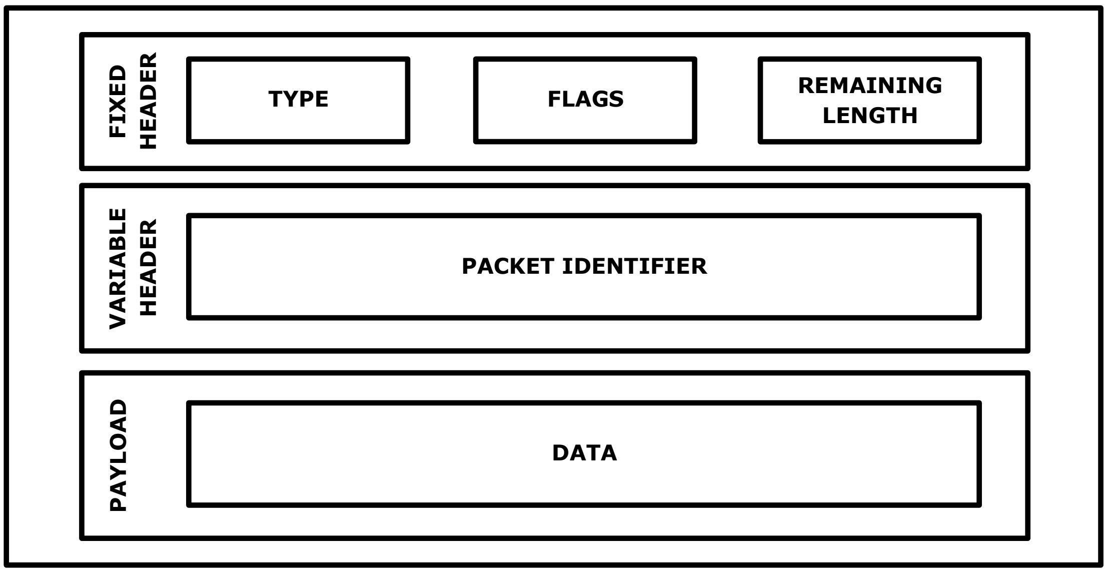


Figure 19 - Structure of MQTT control packets.

Table 7 - MQTT control packet types [24].

Name	Value	Direction of flow	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Client request to connect to Server
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Client to Server or Server to Client	Publish message
PUBACK	4	Client to Server or Server to Client	Publish acknowledgment
PUBREC	5	Client to Server or Server to Client	Publish received (assured delivery part 1)
PUBREL	6	Client to Server or Server to Client	Publish release (assured delivery part 2)
PUBCOMP	7	Client to Server or Server to Client	Publish complete (assured delivery part 3)
SUBSCRIBE	8	Client to Server	Client subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Client to Server	Client is disconnecting
Reserved	15	Forbidden	Reserved

Data model:

MQTT is payload agnostic, which means that basically any data model can be encoded inside MQTT messages. However, it should be taken into account that large payloads may lead to IP fragmentation, which may greatly increase latency and decrease performance.

Topic structure:

Although there is only one endpoint for communication in MQTT, which is the broker, systems implementing this protocol may have a notion of semantics for the way data is exposed. This is due to the hierarchical nature of topics, which makes it possible to group data sources into different levels. For instance, a sensor may publish data in a topic lying inside a component, pertaining to a given workspace and the topic name would be /workspace/component/sensor/value, while the component may expose a status in /workspace/component/status. This allows for clients to either subscribing to individual topics or groups of topics. Topic filtering is also supported using wildcards.

4.3.3.2. Security

MQTT offers simple authentication (username/password) for clients registering to a broker. Also, as transport is TPC, messages may be encrypted with SSL/TLS. Authentication is made during the connection phase, where clients may use a reserved field inside the control packet to send their username and password. The content of this field is application specific, thus, the use of third party solution (e.g. LDAP or OAuth is possible). When using TLS, SSL certificates may be used to authenticate clients.

4.3.3.3. Quality of service

Upon connection, each client may define one of three levels for the QoS employed to the exchange of messages with the server.

Level 0: Fire and forget:

This is the lower level of QoS in MQTT. Using this level, messages are delivered according to the capabilities of the network and no response is sent by the server. This level does not assure that packets are not loss, however, allows for faster delivery and better performance, for higher publishing rates.

Level 1: Delivered at least once:

This level ensures that each message arrives at the receiver, at least once. In this sense, each message is sent and resent until the sender receives an acknowledgment (PUBACK) for the message's packet identifier (Figure 20). This level does not ensure that messages cannot be duplicated, thus it comes to the receiver to identify and manage duplicated packets.

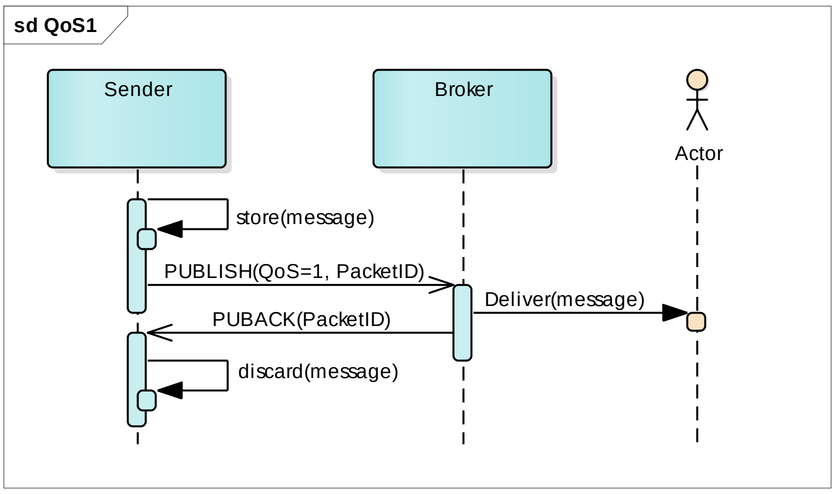


Figure 20 - MQTT QoS 1 example.

Level 2: Delivered exactly once:

This is the higher level of QoS used in MQTT and is used when messages are to be delivered only once. This level not only ensures that messages are not lost, but also guarantees that there is no duplication of packets. This is also the less efficient method, as it includes an additional send/wait/receive step (Figure 21).

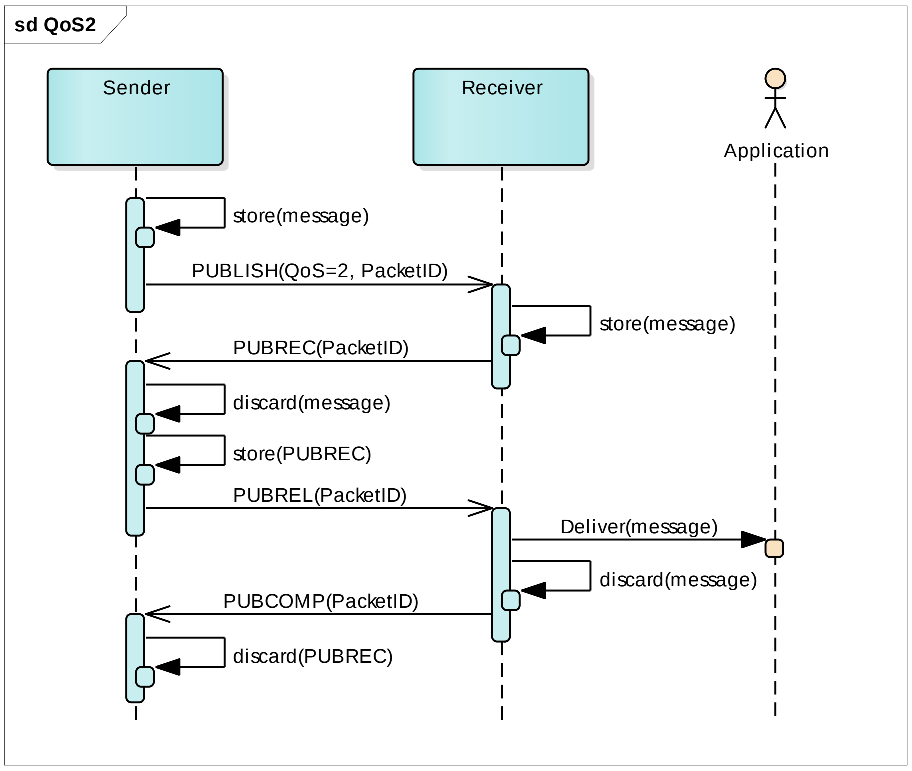


Figure 21 - MQTT QoS 2 example.

Keep alive:

When a client establishes a connection, it can configure a time interval to publish a keep alive signal. This way, the broker can easily detect a defective device, if it doesn't publish anything within the keep alive cycle. The keep alive for MQTT is, however, measured in the order of seconds, with a maximum value of 18 hours, 12 minutes and 15 seconds. Values under 1 second are not supported.

4.3.3.4. "Last will and Testament" (LWT)

The LWT service allows clients to register a custom message in the broker, to be sent for subscribers of any of its topics, in the case a device is disconnected or unavailable. The "Will Message" will be published when:

- An I/O error or network failure is detected by the broker;
- A keep alive stream is broken;
- A client closes a connection without sending a DISCONNECT packet;
- A protocol error occurs.

4.3.3.5. Message persistence (caching)

An MQTT broker can store messages for data persistence. The decision to store or not store a given message is made by clients which should activate a special flag inside the message itself. However, only the last message published to a given topic is stored, thus, being this data persistence service more close to a caching mechanism, that allows new clients to retrieve topic information without the need for waiting the next topic update.

4.3.3.6. WebSocket support

MQTT also provides support for using WebSocket as the transport protocol. WebSocket allows for full-duplex communication, over a single TCP channel, between a server and a browser, thus allowing fast (low latency) data transfer between web-based applications.

4.3.3.7. Constrained device support

As TCP socket connections may be a drawback to very constrained environments, MQTT may also support UDP transport. This support is given by an extension to the protocol, namely MQTT-SN (MQTT for sensor networks), which also defines a way for MQTT-SN enabled brokers indexing topic names, so messages need only to carry numeric topic indexes, instead of very long string addresses.

4.3.4. MQTT solution comparison chart

The following table provides a comparison of the most popular MQTT SDK's in terms of current functionality. It is important to emphasize that some of the shown implementations are currently under active development and missed functionality may be added after the writing of this document.

Table 8 - MQTT Solution Comparison Chart

	Eclipse Paho				Moquette	Mosquitto
	Java	Python	C	C#	Java	C/C++
MQTT 3.1	Yes	Yes	Yes	Yes	Yes	Yes
MQTT 3.1.1	Yes	Yes	Yes	Yes	Yes	Yes
Security	Yes	Yes	Yes	Yes	Yes	Yes
QoS 1	Yes	Yes	Yes	Yes	Yes	Yes
QoS 2	Yes	Yes	Yes	Yes	Yes	Yes
Keep alive	Yes	Yes	Yes	Yes	Yes	Yes
LWT	Yes	Yes	Yes	Yes	Yes	Yes
Automatic reconnect	Yes	Yes	Yes	No	Yes	Yes
Offline buffering	Yes	Yes	Yes	No	Yes	Yes
Message persistence	Yes	No	Yes	No	Yes	Yes
WebSocket support	Yes	Yes	No	No	Yes	Yes
MQTT-SN	Yes	No	Yes	No	No	Yes

4.3.5. Performance

Performance tests were made in order to stress payload interchange between a Publisher and a Subscriber and verify its impact on latency timings and scalability. Table 9 shows the hardware setup used for testing MQTT.

Table 9 - Specifications of hardware used for testing the performance of MQTT.

Specifications	Client	Server
CPU	Intel core i7 3630QM @3.2GHz	Intel core i5 4300u @1.9GHz
RAM	16 GB	8 GB
OS	Ubuntu 14.04	Ubuntu 14.04
Network switch	Draytek Vigor3900	

For latency testing, first it was used a public MQTT broker provided by Eclipse, and second, a locally hosted broker. The communication was tested using the Python version of Eclipse Paho API. For the local hosted broker test, Mosquitto was used.

Latency - Internet Broker Test:

The results from the communication test between the publisher and subscriber over a broker hosted on the internet can be observed in Figure 22.

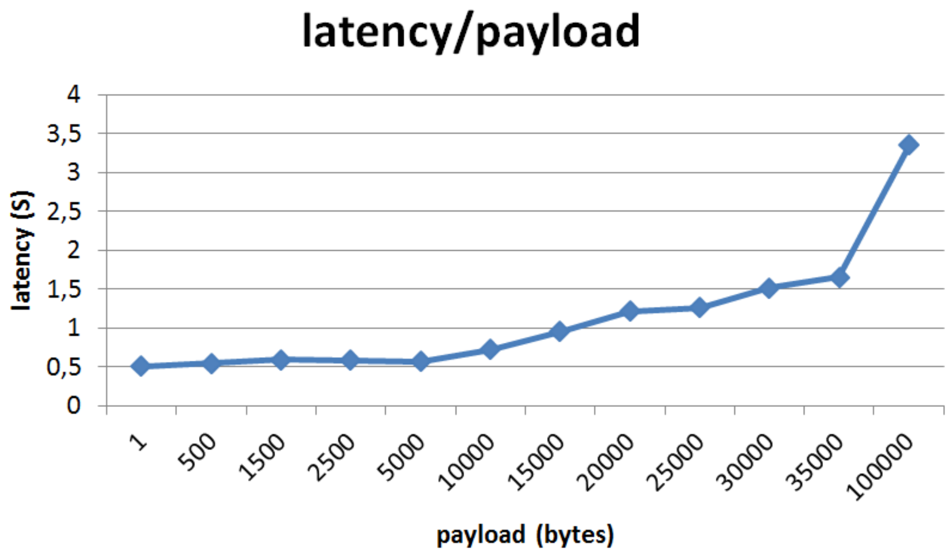


Figure 22 - MQTT protocol performance (latency)/payload size over the internet.

It can be observed that the average latency only gets affected when the messages payload exceeds 5Kb and that for smaller sizes, the latency value floats between around 0.5 seconds.

Latency - Local Broker Test:

The results from the communication test between the publisher and subscriber over a broker hosted on the local network, can be observed in Figure 23.

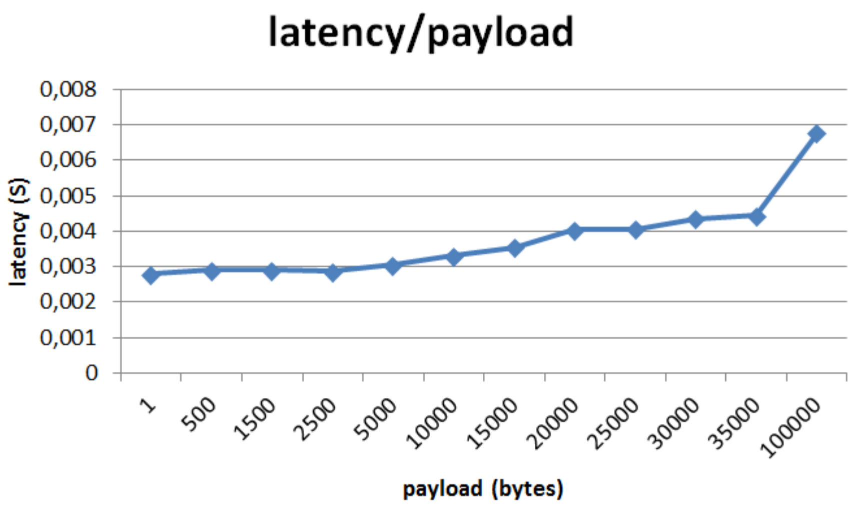


Figure 23 - MQTT protocol performance (latency)/payload size over a local broker.

It can be observed that the average latency only gets affected when the messages payload exceeds 5Kb and that for smaller sizes, the latency value floats around values lower than 0.003 seconds.

Scalability:

For scalability, two different brokers (Moquette and Mosquitto) where used for hosting a topic subscribed by up to 50 clients over a local network. The topic was fed by a single client publishing the current date at a rate of 10 Hz, using QoS of level 2 (see Section 4.3.3.3). Both CPU usage and memory consumption was observed. The results obtained are shown below:

Table 10 - CPU load, in percentage, of the tested MQTT brokers.

Nr. Of clients	**0**	**1**	**10**	**20**	**30**	**40**	**50**
Moquette (Java)	0	0.3	1.0	1.2	1.3	1.6	2.3
Mosquitto (C)	0	**0**	**0.3**	**0.3**	**0.7**	**0.7**	**1.0**

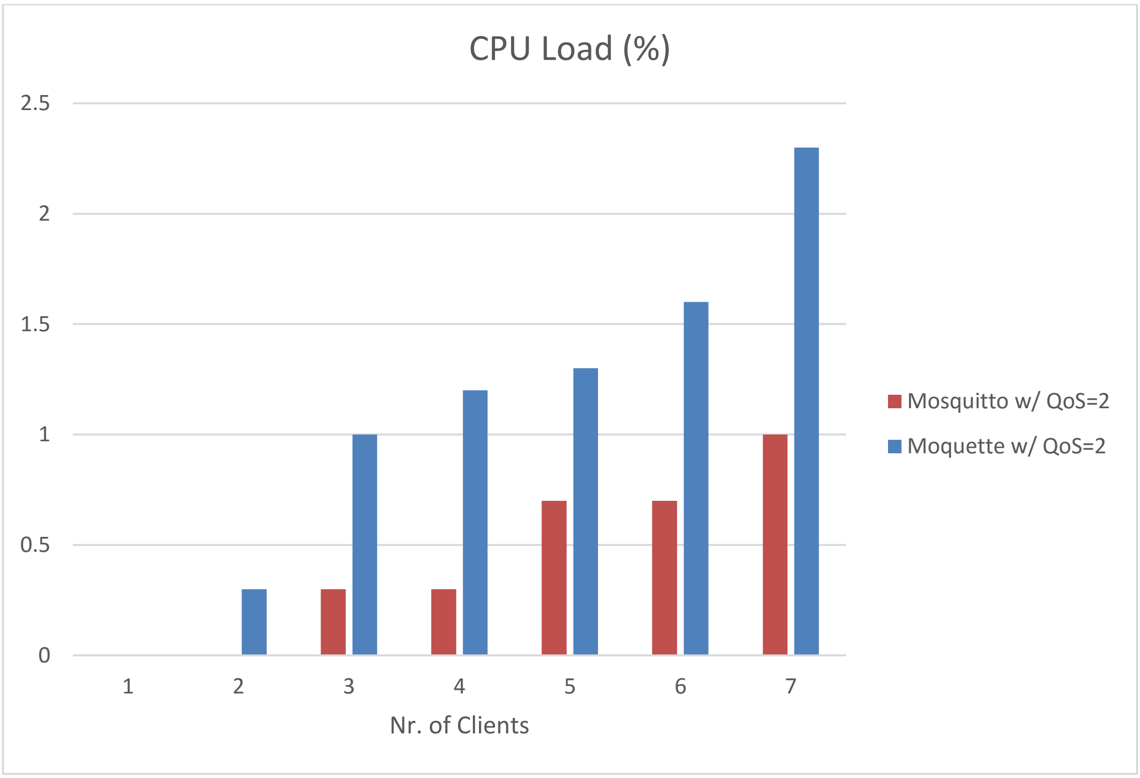


Figure 24 - CPU load, in percentage, of the tested MQTT brokers.

Table 11 - Memory usage, in MB, of the tested MQTT brokers.

Nr. Of clients	**0**	**1**	**10**	**20**	**30**	**40**	**50**
Moquette (Java)	95.552	96.904	98.696	100.060	101.708	102.772	103.856
Mosquitto (C)	0.128	**0.156**	**0.160**	**0.164**	**0.168**	**0.172**	**0.176**

As is possible to observer, both solutions are relatively lightweight and highly scalable, as the load increase is not very substantial. However, Mosquitto has clear advantages, mainly in memory usage, over Moquette.

4.3.6. MQTT solution analysis

MQTT was designed mainly for small mobile devices and low-bandwidth networks. By centralising data on brokers, data exchange can be more efficient, with less overhead. Also, client configuration is very simple, however, it must know which topics should be subscribed as there is no direct way of knowing the complete address space of topics registered in the broker.

In the security side, MQTT offers some degree of customisation and security, namely by supporting TLS/SSL. The quality of service provided is fair and balanced, giving the possibility for applications to configure its devices for better performance/ reliability trade-off without much complexity. The communication is bi-directional and does not require notification services, as these may be implemented just like another topic.

The current MQTT SDK implementations are complete and mature, offering a large range of programming languages devices. The lightweight is evident, as is also the scalability. However, observing the tested solutions, it is clear that a Java implementation (Moquette) demands for more resources and should be considered only when it is more of a developer requirement (to be able to implement code in Java), rather than a hardware issue. In other cases, **Mosquitto** should be used.

4.4. OPC UA

The OPC Unified Architecture (OPC UA) protocol is an M2M communication standard widely used in industry. It is the successor of Open Platform Communications (OPC), developed by the OPC Foundation[10]. Classic OPC protocols, such as 'Data Access', were designed to allow a standardised read and write access to current data in automation devices, without interfering with their normal operation. This design was adopted mainly by HMI and SCADA systems that could, with the use of OPC, concentrate data obtained by different devices and vendors at once. Following the success of OPC DA, the release of subsequent OPC standards allowed it to be extended with more functionality (e.g. OPC HDA added access to historical data, while OPC A&E introduced specific message types for exchanging alarms and events through OPC). However, classic OPC had a major limitation by using MS Windows COM/DCOM technology to get remote access to servers. To circumvent this issue, the OPC foundation first released OPC XML-DA which used XML-based web services to eliminate the platform dependency. Due to poor performance and incompatibility issues, this solution was not considered satisfactory. OPC UA is, thus, result of the effort in creating a new generation of OPC that could maintain functionality and performance of the original and could be truly platform independent.

To better meet the requirements for openMOS, such as openness and adaptability, only OPC UA will be considered in this assessment. The following offers an analysis on this protocol regarding mainly functionality and performance testing.

4.4.1. Architecture

As stated in [25], the architecture for the set of standards that integrates the OPC UA protocol were conceived to satisfy the following requirements:

[10] https://opcfoundation.org/

Table 12 - Requirements for OPC UA (as defined in [25])

Communication between distributed systems	Modelling data
• Reliability by: ▪ Robustness and fault tolerance ▪ Redundancy • Platform-independence • Scalability • High performance • Internet and firewalls • Security and access control • Interoperability	• Common model for all OPC data • Object-oriented • Extensible type system • Meta information • Complex data and methods • Scalability from simple to complex models • Abstract base model • Base for other standard data models

This set of requirements motivated the creation of a bottom-up architecture, as depicted in Figure 25 - OPC UA layered architecture (courtesy of Unified Automation GmbH)., where new functionality can be stacked above the core foundations (base technology) of OPC UA.

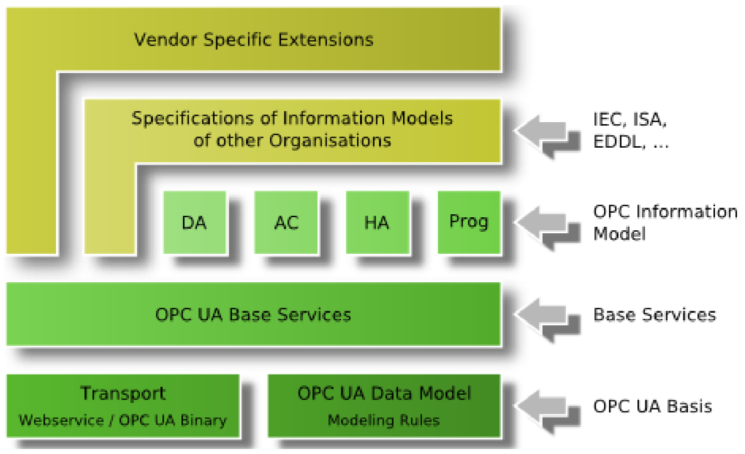


Figure 25 - OPC UA layered architecture (courtesy of Unified Automation GmbH).

The main components for the basis of OPC UA are 'Transport' and 'Data Modelling'. Base services are laid on top of those foundations as a mean to standardise the way applications and protocols can interact with OPC UA core components. After the services comes the information models which semantically describe the data and methods of the OPC UA application. Basic data types, object nodes and references between nodes are used to offer a certain kind of data wise functionality for third-party applications and HMI's. The following sections provide a more detailed view of each module.

4.4.1.1. Transport layer

Figure 26 details the current transport possibilities for the protocol, which include: a) XML-based Web Services, b) SOAP/HTTP interface with UA binary coded security and c) OPC UA native binary communication. They differ from the more firewall-friendly and open approach, obtained with a), to the most optimized, performance oriented version, achieved with c). Either way, data transport carried out with OPC UA is always done through TCP/IP socket connections (client-server). Both IPv4 and IPv6 addresses are supported.

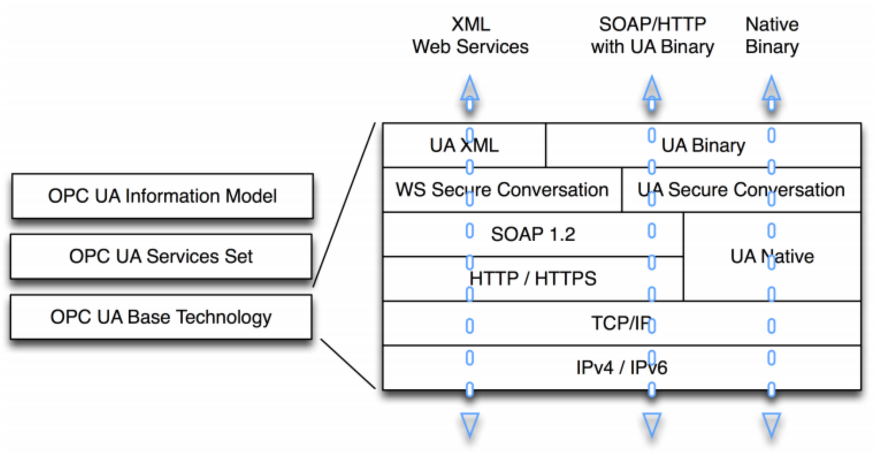


Figure 26 - OPC UA specification stack (as shown in [26])

4.4.1.2. Security

For secure connections, OPC UA offers three options, inherited from Web Services: none (unsafe), username/password credentials and X.509 certificates. Security may be implemented through WS-SecureConversation policy or UA Secure Conversation, which is the same but binary coded, for better performance. Also, a hybrid option is available: binary coding over SOAP transport. If the transport is using UA Binary, the Security also has to be binary coded. A more detailed view about OPC UA security and authentication is given in section 4.4.3.3.

4.4.1.3. Data model

Also included in the base technology is the data model (sometimes referred as address space), which defines the way data is structured and exposed to OPC UA applications. This data model allows for every physical system to be virtually represented in an OPC server. Thus, the OPC UA data model considers objects in terms of variables and methods, as well as their relationships (references and inheritance), in a similar way as any object oriented programming language (Figure 27). Variables represent values, which can be either properties/definitions or data.

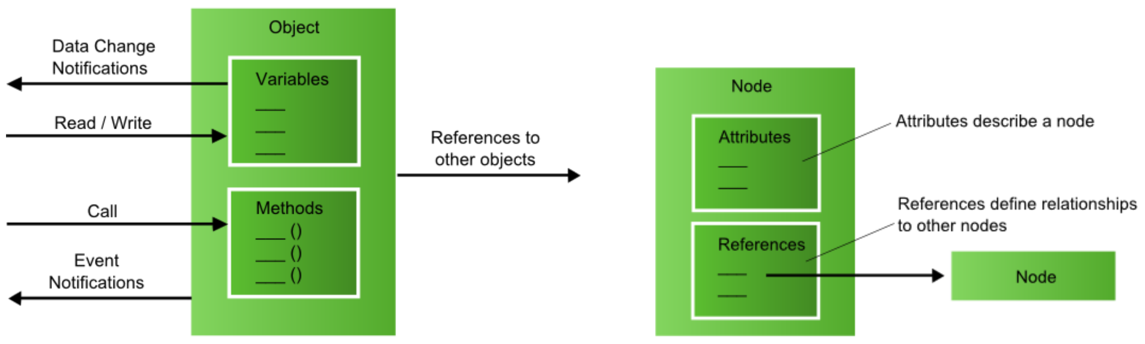


Figure 27 – Left: OPC UA object model. Right: nodes and references. (Courtesy of Unified Automation GmbH).

UA objects and their components are represented, in the OPC UA address space, as nodes described by attributes and interconnected by references. Consequently, each node can represent an object, variable or method. The node attributes that describe the node can be accessed similar to other variables in the OPC UA data model (read, write, query, or subscription) and consists of a node id, a name, a description, a data type and a mandatory/optional indicator. References relate nodes to each other and allow for building a topological model of objects. Nodes can also reference other nodes outside their own address space or server.

4.4.1.4. Base services

The OPC UA base services stack grants basic access (read/write) to object data and allows to specify how data needs to be exchanged. Also, it allows for browsing a server's namespace, that is, finding its nodes and node relationships. Available services depend on the type of the related object component. A more detailed description of the services provided by OPC UA is given in section 4.4.3.

4.4.1.5. Information model

Apart from data modelling, *i.e.* describing how binary data can be encoded in order to represent some physical value, OPC UA also considers information modelling, *i.e.* describing how data models can be combined in order to describe a given situation/status of the system, or its *semantics*. In fact, OPC UA uses the concept of information models to represent data related to physical systems in a semantically abstract/generic way that can be recognized and handled by computer-based automated processes. This abstraction allows for fully detaching data production from data consumption. Thus, in the architectural design of OPC UA, the information model stack represents the link to interlock third-party components and systems given that the complex data that defines the cyber-physical system at hand follows a certain protocol.

The use of generic and complex data in information models may arise transport and scalability issues, despite allowing interoperability. On one hand, having large structures that define objects and relationships inside a given system implies that such structures must be split in order to be exchanged over the network. This entails a consistency problem when the data consumer has to join them again. On the other hand, having a static specification model limits the internal structure of data representation, as well as the possible relationships. To avoid those issues, OPC UA uses references to describe relationships between data models. By using references (pointers to data sources) rather than static data embedded inside the structure itself, OPC UA information models are scalable, adaptable and can be easily partitioned and reconstructed.

In sum, information models offer a way for vendors to customize how data is exposed, without limitations, similarly to a computational program. These "programs" (semantic models) have the notion of hierarchy and inheritance and reflect behaviour and logics. OPC UA also includes base models that can be used and extended just like programming libraries. Figure 28 shows an example how this modelling technique can be used to expose data from a sensor device.

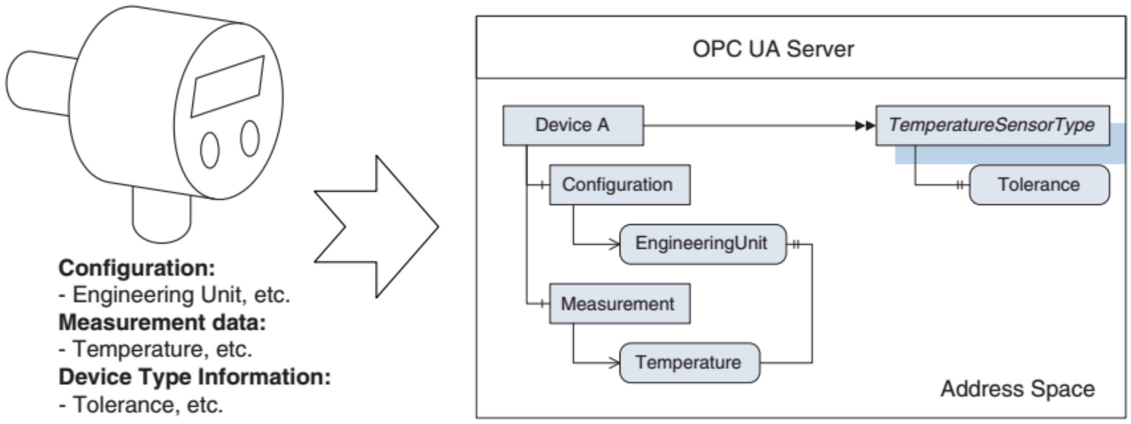


Figure 28 - Temperature sensor and provided data in OPC UA [25].

4.4.2. Licensing and system integration

Most of the OPC UA core implementation, defined and provided by the OPC Foundation [27], is open source. However, OPC UA technology is copyrighted and its use must comply with the *OPC Foundation license agreement*[11]. This core/reference implementation is available in three different, widely used, programming languages, as listed in the table below:

Table 13 - List of OPC UA reference implementations.

Name	Language	License
UA-.NET	C# (.NET Framework 4.5)	GPL 2.0
UA-AnsiC	C	GPL 2.0
UA-Java	Java	GPL 2.0

Regarding system integration, besides offering the basic infrastructure for the creation of vendor specific OPC-based products, this core implementation does not provide high level functionality, nor an SDK that can be easily used to develop full featured OPC UA clients and servers. Therefore, several third-party companies already developed comprehensive SDKs to facilitate de adoption and customisation of this technology among vendors and system integrators. The following table summarises the open source OPC UA SDK implementations known to date:

[11] https://opcfoundation.org/license-agreement/

Table 14 - List of open source OPC UA SDK implementations.

Name	Language	License	Client	Server
Eclipse Milo **[28]**	Java	Eclipse Public License	Yes	Yes
FreeOpcUa **[29]**	C++ / Python	LGPL	Yes	Yes
Node-OPCUA **[30]**	JavaScript	MIT	Yes	Yes
Opcua4j **[31]**	Java	Creative Commons 3.0	No	Yes
Open62541 **[32]**	C	LGPL + static linking exception	Yes	Yes
OpenSCADA **[33]**	C++	GPL	No	Yes
Uaf **[34]**	C++ / Python	LGPL	Yes	No
Worstation-UaClient **[35]**	C#	MIT	Yes	No

Despite offering more functionality than the reference implementations, namely ready to use OPC UA clients and servers, these open source SDKs still lack some of the services and features proposed in the OPC Foundation standard specifications. In fact, most of these SDKs were built mainly by academic institutions and researchers with the main purpose of providing proof of concept and testbeds for OPC UA technology (e.g. FreeOpcUa, Node-OPCUA). Even industry oriented implementations (e.g. Eclipse Milo, open62541) are being carried out by community developers, thus having slower paced updates and less documentation than seen in solutions provided by for-profit organisations. The following table summarises the most important commercial SDK implementations known to date:

Table 15 - List of commercial OPC UA SDK implementations.

Name	Language	Platform	Client	Server
MatrikonOPC **[36]**	Ansi C	Embedded devices	No	Yes
Prosys OPC **[37]**	Java	JRE 1.6 and above	Yes	Yes
Softing **[38]**	C++ / C#	Multi-platform	Yes	Yes
Unified Automation **[39]**	C / C++ / C#	Multi-platform, including embedded devices	Yes	Yes

If the goal is only to expose current device data with minimal configuration and programming effort, one can use third-party applications. Such solutions typically work as standalone applications, previously built for different host OS's, and are able to use device drivers to manage communication and data gathering from legacy devices (e.g. PLC's). The most flexible solution currently known to adopt this philosophy is the *KEPServerEX* [40] from Kepware Technologies. It is composed of a core application that can be combined with additional drivers and plugins to meet the connectivity and functionality needed. It has drivers for over 150 PLC's and other automation devices from the major brands used in industry. Also, it integrates with current IoT technologies through Web Services and/or MQTT protocols. *MatrikonOPC* and *Softing* have a similar solution, however, with less supported devices and without the IoT extensions. Moreover, MatrikonOPC drivers are only implemented for OPC classic, which can be converted for OPC UA protocol using a wrapper that is available also from MatrikonOPC. For more dedicated solutions, some hardware suppliers already offer OPC UA integration packages for their devices. For instance, *Beckoff* provides OPC UA servers and clients for any device that supports their proprietary communication protocol TwinCAT. Finally, CODESYS integrates an OPC UA Server module that is available, without fees, for their SoftPLC systems.

4.4.3. Services and utilities provided

In the previous subsections, a glance on the OPC UA protocol was presented, which included its architecture and available solutions. This subsection will offer a more focused view on the current state of OPC UA services and utilities. For each presented topic, a brief assessment on current availability is given. This assessment was taken mainly from six tested SDK implementations (three open-source and three commercial solutions).

4.4.3.1. Information modelling

As seen in subsection 4.4.1.5, the OPC UA protocol predicts that any physical system can be represented reliably, in a virtual environment, by means of information models, or semantic models, that retain information about not only data types but also interactions between objects and current system states. However, modelling an entire system under these moulds can be far from trivial. To ease this process, some solution providers for OPC UA already include tools that are able to generate the complete OPC UA address space (*i.e.* the representation of the information model), for a given input of structured data provided by the user, which can be transferred to the OPC UA server upon its deployment. A complete solution for this utility (see

Figure 29) would contain:

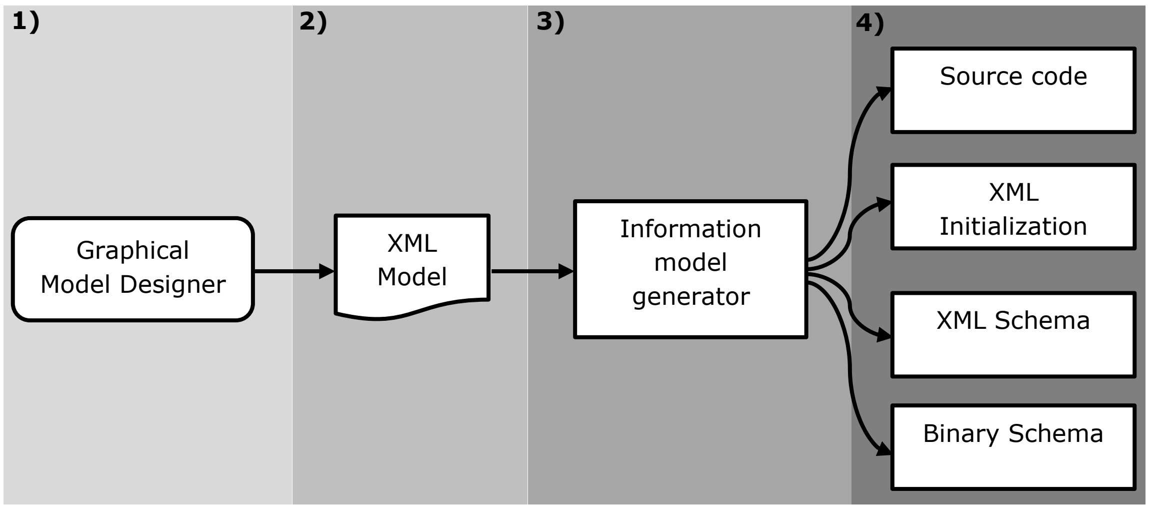


Figure 29 - Block diagram of a typical Information Modelling tool for OPC UA.

1) a graphical designer for the user to easily and intuitively build a representation of the system endpoints and interactions (e.g. by means of a block diagram);

2) a translator that generates structured data, as an XML file, from 1);

3) a generator (typically included in the SDK) that gets the XML Model file from 2) and generates the address space (information model) for the OPC UA server;

4) a set of generated files that are used to build the OPC UA server's address space during its deployment and initialization phases.

Availability:

CAS OPC UA Address Space Model Designer [41] is a graphical tool that incorporates all the modules from

Figure 29. The designer allows a form filling approach to create nodes, types, views, methods and references. The application then generates all the necessary files to integrate the address space in the OPC UA server. A 'Professional' version of this tool offers 2D and 3D visualization of the model. However, the source code is only generated in C# language and the application itself is still "work in progress", with some serious limitations and unimplemented functionality.

Unified Automation UaModeler [39] is a tool similar to the above. It also has a form filling approach for editing objects and a 2D visualization mode. It lacks the 3D visualisation but is able to generate code for C#, C++ and Ansi C. Also, it is possible to export the namespace in a standard XML file format. Moreover, is a more stable solution, able to run on Windows or Linux and more intuitive.

FreeOpcUa opcua modeler is a free alternative to a graphical information model designer, provided by FreeOpcUa [29]. However, is still work in progress, with substantial issues to be resolved.

open62541 code generator is a simple generator, included in the open62541 SDK [32], that lacks the graphical interface. However, it is able to generate the server's address space, from a given XML model file, in C language.

Prosys code generator [37] is similar to the above but generates code for Java instead of C.

4.4.3.2. Discovery

A set of discovery features is defined in OPC UA such that clients know how to connect to servers on the same network. The needed information to successfully establish a connection is stored in an 'endpoint', provided by the server. One server can provide several endpoints, each one containing: an URL (including protocol, IP address or host name, and port), a security policy, the supported message security modes (signature and/or encryption) and the supported user token types (password, certificates and/or WS-Secure).

In order to provide discovery functionality, each server must register itself in a 'Discovery Server'. When using this feature, clients may query the Discovery Server (calling the 'FindServers' service) for available OPC UA servers. The Discovery Server then replies with the discovery URL of the registered servers (usually the same as the endpoint URL) that can be used by clients to retrieve endpoint information (by calling the).

Local discovery:

In the case when more than one server is running on the same host machine, a Local Discovery Server (LDS) is necessary. To interact with the LDS, clients must know the IP address of the machine it is running on. The default/standard port number for the LDS is 4840. The figure below shows the sequence diagram for the local discovery service:

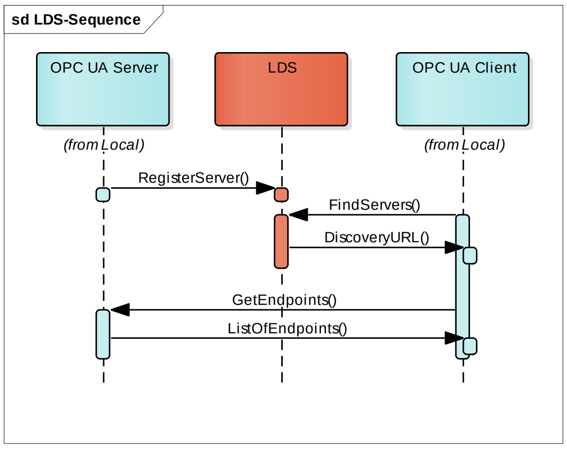


Figure 30 - OPC UA local discovery sequence diagram.

Multicast subnet discovery:

As seen, the fact that clients must know the addresses where the servers are located may be very limitative. Thus, OPC UA offers an option for finding servers on a local subnet, using *zeroconf* technology (mDNS, see section 3.6.1), which enables name resolution without the need for having a centralised DNS server. This multicast extension for the LDS (LDS-ME) allows the LDS to automatically announce servers that get registered to it. When new servers are announced, each LDS updates a local cache with the addresses of each server in the network. When a new client arrives, it calls the service 'FindServersOnNetwork' on the local LDS-ME, which retrieves its cached addresses. The figure below shows the sequence diagram for the multicast subnet discovery service:

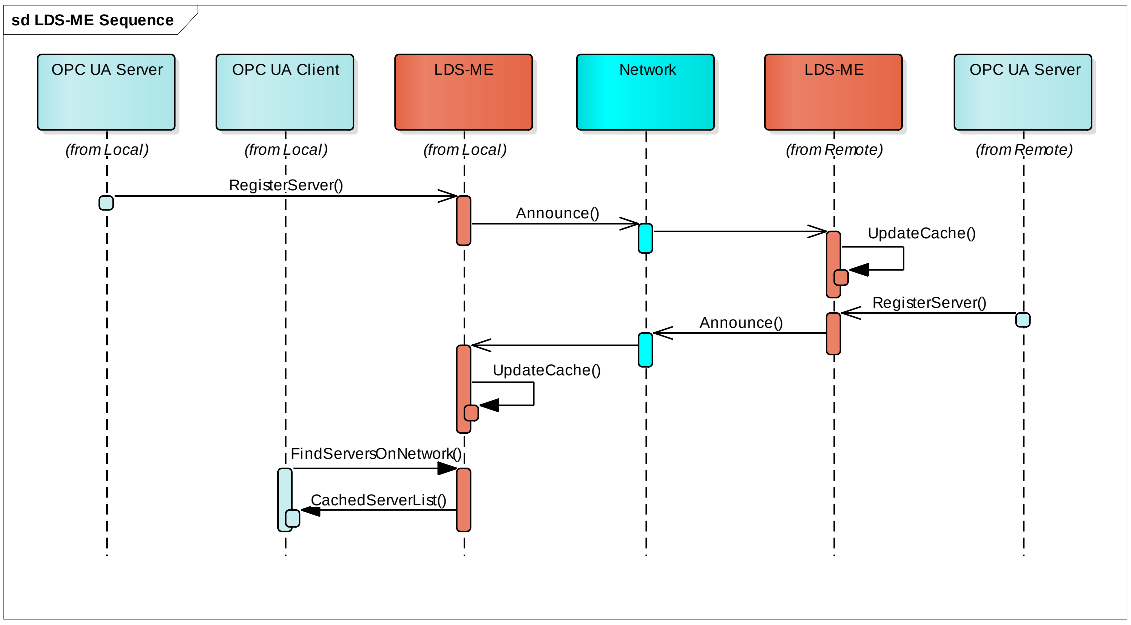


Figure 31 - OPC UA multicast subnet discovery sequence diagram.

Global Discovery:

The Global Discovery Server (GDS) is a concept that proposes to use an OPC UA server to provide the discovery option over the entire network. This approach requires that all OPC UA servers shall register themselves with the GDS. The GDS acts as a central management system for OPC UA server addresses and also certificates. This functionality, however, requires that all OPC UA servers and clients know the location of the GDS, i.e., the IP address and port.

Availability:

The LDS and LDS-ME are available as standalone applications, for Windows based OS's, from the OPC Foundation itself. LDS is already a stable release, based on ANSI C code. LDS-ME is yet a release candidate. open62451 also supports LDS-ME in a current feature branch which will be included into the master branch soon. The GDS is still a conceptual specification. The OPC foundation already have a beta implementation of a sample GDS, in C#, but to date is only available for members.

4.4.3.3. Security and authentication

The OPC UA protocol is meant to be used in heterogeneous environments, with different networks (industrial or not), consequently with different security policies. Moreover, it can be used for different access levels, for instance running different servers bound with different data providers. This implies that different data can have also different requirements in terms of security, e.g. if performance is more important or if data can only be accessed by a specific user or subsystem. In order to support all possibilities, the OPC Foundation has defined security for OPC UA as a multilayer mechanism (Figure 32) designed to offer the most flexible way of protecting data integrity, according with the user needs.

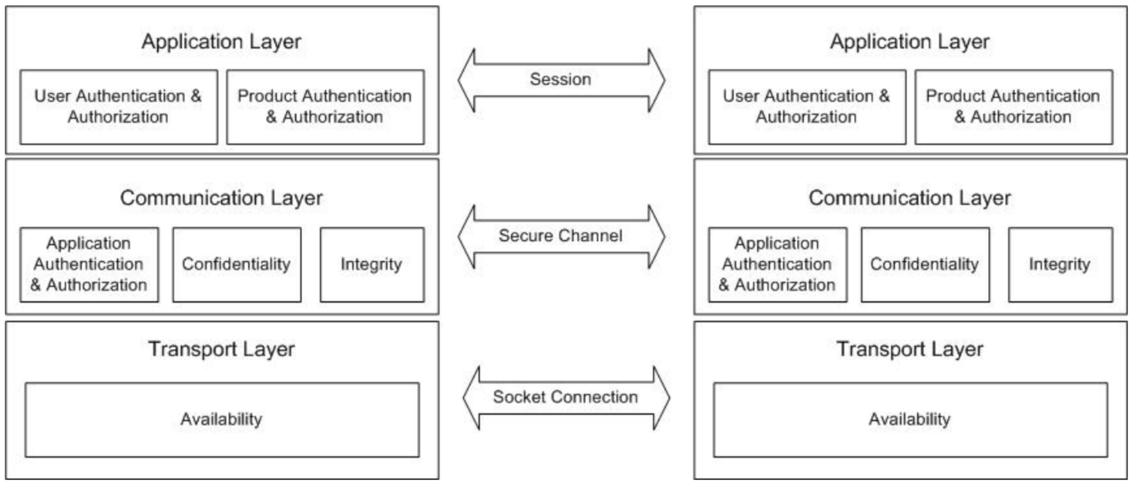


Figure 32 - OPC UA security architecture [25]

Authentication, in conjunction with authorisation, allows to define user / product / application centric data flow down to the node level inside the server's structure. This is done by creating sessions (in the application layer) identified either by a username and password, or by a certificate. Those sessions run on top of a security channel (in the communication layer) that uses digital signatures and encryption to secure data exchanged between the server and the client. If, on the one hand, authentication gives permission for data access, on the other hand, authorisation enables to determine which data / operation a client is allowed to get / perform.
For Web Service communications, OPC UA uses standard WS Secure Conversation, maintaining compatibility with SOAP implementations.

Availability:

From the tested solutions, only the SDK's from Prosys and Unified Automation offer a fully featured security bundle. Current open source solutions still lack some of the security features defined by the OPC Foundation (see Table 16 for details).

4.4.3.4. Data access

As previously stated, the OPC UA protocol follows a client-server approach for data exchange. The typical use case for data access have clients directly reading or writing data in server's nodes (direct access). However, OPC UA supports two more types of data exchange: subscriptions and notifications.

Read/write:

Direct reading and writing is a basic service provided by OPC UA. To access this service, clients must know the node's address and data type of the value they want to access. Both can be browsed in the server's address space. This method of exchanging data may, however, not be the optimal for most cases as it adds unnecessary communication overhead in case of continuous and subsequent calls for data read/write.

Subscriptions:

The subscription service is a relatively new functionality specified in the OPC UA standards, that is meant to add a publish/subscribe methodology to the protocol. When using this service, clients can subscribe to one or more nodes for reading data. The server then periodically sends data updates to the client with current values. Optionally, the client can also choose to receive data only when values change, thus reducing network load.

Notifications (events):

The notification service, also mentioned in the OPC UA standard as events, is a special case of subscription service. This service is meant to broadcast a special kind of message when something meaningful happens on the system (e.g. a certain process reaches the end of execution). According with the OPC UA data model, events are common objects, however, they incorporate a special kind of data type - 'EventType' – which includes a data structure composed of an identifier, a timestamp, a message a level of severity. Clients can browse and subscribe to events through the service 'EventNotifier'.

Alarms and conditions:

The OPC Foundation specifies an information model for conditions, acknowledgeable conditions, confirmations and alarms. In their terms, conditions are a special type of events that always maintain a certain state (e.g. enabled or disabled). Thus, are used mainly to reflect current system state or an infinite state machine. Acknowledgeable conditions are the same as conditions but require user input when a state is about to change, which can be done calling a method. Alarms are acknowledgeable conditions that are used to request a user attention on critical system states. Alarms can also be shelved to handle more important alarms and postpone the less important alarms.

Methods (call service):

OPC UA methods are atomic components that can be included inside objects. Using the Call Service, clients can call an individual method, or a list of methods exposed by a server[12]. When calling a method, a client is actually triggering a callback on the server that is designed to produce a certain action. These methods/callbacks can accept input arguments and must always return an output value (or list of values).

Historical Data:

The OPC UA standard allows the access and changing of historical data accumulated on servers. For this, the 'AccessLevel' and 'UserAccessLevel' attributes must be defined. Currently, only the value attributes of variables support "Historizing". Services used to access and change historical data are known as 'HistoryRead' and 'HistoryUpdate' respectively.

[12] Depending on the client's SDK functionality.

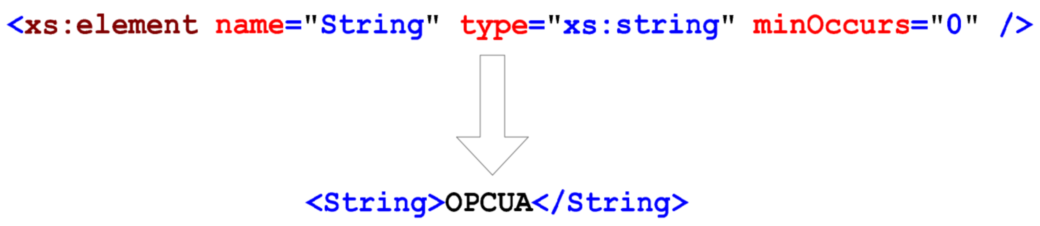
Availability:

The commercial SDK's tested for this survey (Unified Automation and Prosys) implement all of the data access solutions flawlessly. The open sourced implementations, besides offering the basics (read/write), still lack some of the additional functionality (see Table 16 for details).

4.4.3.5. Complex data

XML:

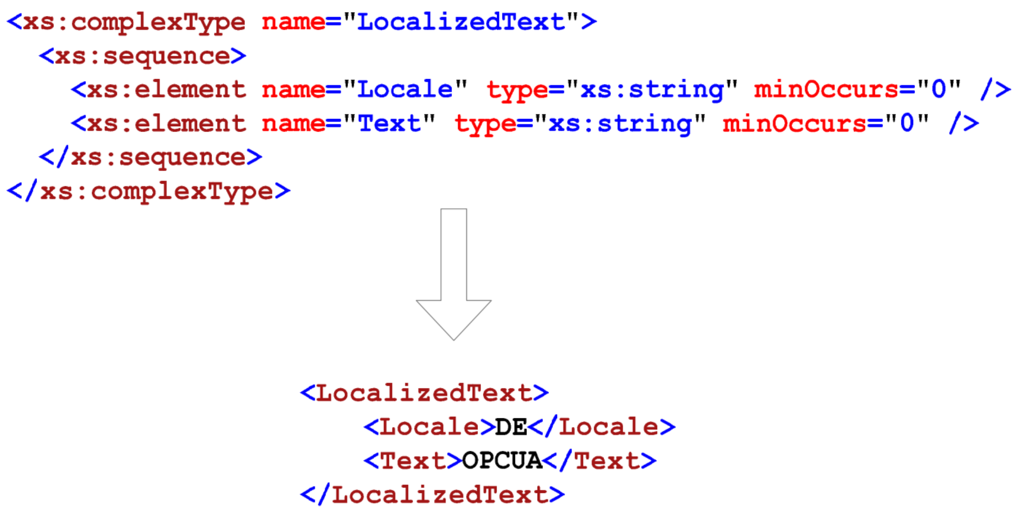
The XML format plays a predominant part on OPC UA, at several levels. Thus the support for this format is logically included in the specifications. In fact, OPC UA can use XML for transport (when using Web Services protocol) or to define a server's information model (using a specific schema – Opc.Ua.NodeSet2.xml). However, this subsection focuses on the support for exchanging data encoded with XML format, which is crucial to maintain compatibility with third party applications. In order to enable this support, the OPC Foundation specifies the XML element data type that can be used either to encode primitive types (Figure 33) or complex structures (Figure 34).



```
<xs:element name="String" type="xs:string" minOccurs="0" />
```

```
<String>OPCUA</String>
```

Figure 33 - Example of a primitive type (string) encoded in XML [25].



```
<xs:complexType name="LocalizedText">
  <xs:sequence>
    <xs:element name="Locale" type="xs:string" minOccurs="0" />
    <xs:element name="Text" type="xs:string" minOccurs="0" />
  </xs:sequence>
</xs:complexType>
```

```
<LocalizedText>
    <Locale>DE</Locale>
    <Text>OPCUA</Text>
</LocalizedText>
```

Figure 34 - Example of a complex type (LocalizedText) encoded in XML [25].

File access:

A recent addition to the OPC UA standard is the specification of a special type of object – 'FileType' – which enables the exposure of files and content, from directories located on the server, through the OPC UA address space. This object defines four properties (Size, Writable, UserWritable and OpenCount) and six methods, that resembles file operations already available in several programming languages (Open, Close, Read, Write, GetPosition [of cursor] and SetPosition [of cursor]

Availability:

This additional functionality is almost an exclusive of the commercial platforms. In fact, for the open source SDKs, only Eclipse Milo supports the XML transport profile.

4.4.4. OPC UA Solution Comparison Chart

The following table provides a comparison of the most popular OPC UA SDK's in terms of current functionality. It is important to emphasize that some of the shown implementations are currently under active development and missed functionality may be added after the writing of this document.

Table 16 - OPC UA Solution Comparison Chart

	open 62541	FreeOpcUa		Eclipse Milo	Prosys	Unified Automation			CODESYS OPC UA Server[13]
		C++	Python			C#	C++	C	
Discovery									
- Register server	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
- Find servers	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Security									
- Authentication	Yes	No	No	Yes	Yes	Yes	Yes	Yes	No
- Certificates	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
- Encryption	No	No	Yes	No	Yes	Yes	Yes	Yes	No
Address Space									
- XML import	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	No
- Browsing	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
- Add/delete nodes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
- Views/Filters	No	No	No	No	Yes	Yes	Yes	Yes	No
Data access									
- Read/write	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
- Subscriptions	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
- Events	No	Yes	Yes	No	Yes	Yes	Yes	Yes	No
- Alarms	No	No	No	No	Yes	Yes	Yes	Yes	No
- Methods	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No
- Historic data	No	No	Yes	No	Yes	Yes	Yes	Yes	No
Complex data									
- XML protocol	No	No	No	Yes	Yes	Yes	Yes	Yes	No
- File transfer	No	No	No	No	Yes	Yes	Yes	No	No
License									
- Open source	Yes	Yes	Yes	Yes	No	No	No	No	No
- Commercial	No	No	No	No	Yes	Yes	Yes	Yes	Yes

[13] The CODESYS OPC UA Server follows the "Micro Embedded Device Server Profile" as defined in part 7 of the OPC UA specifications.

4.4.5. Performance

The abovementioned SDK's were evaluated, in terms of performance, according to their scalability and read/write latency. Tests were made on an Intel Core i7 2.6 GHz laptop, with 16 GB RAM and Linux Mint (Debian based) operating system[14]. The FreeOpcUa (Python version) and the CODESYS OPC UA server were not included in the testing.

Scalability

Scalability was tested connecting up to 50 clients to each server, subscribing to the same node (current time), at a publishing rate of 10 Hz[15]. The progression of CPU load and memory usage was observed. Results are shown in the following:

Table 17 - CPU load, in percentage, of the tested SDK's.

Nr. Of clients	**0**	**1**	**10**	**20**	**30**	**40**	**50**
Eclipse Milo (Java)	0	2	5.7	7.3	8	9.3	10
FreeOpcUa (C++)	0	**0.1**	**1.3**	2.7	3	4.3	**5**
Open62541 (C)	0	0.2	1.7	**2.2**	**2.7**	**3.1**	n/a
Prosys OPC (Java)	0.7	2.3	12	13.7	15.3	17	18.6
Unified Automation (C)	0.3	0.7	2.3	3.7	4.3	4.7	**5**
Unified Automation (C++)	0.3	0.7	4	7.3	9.7	12	14.3

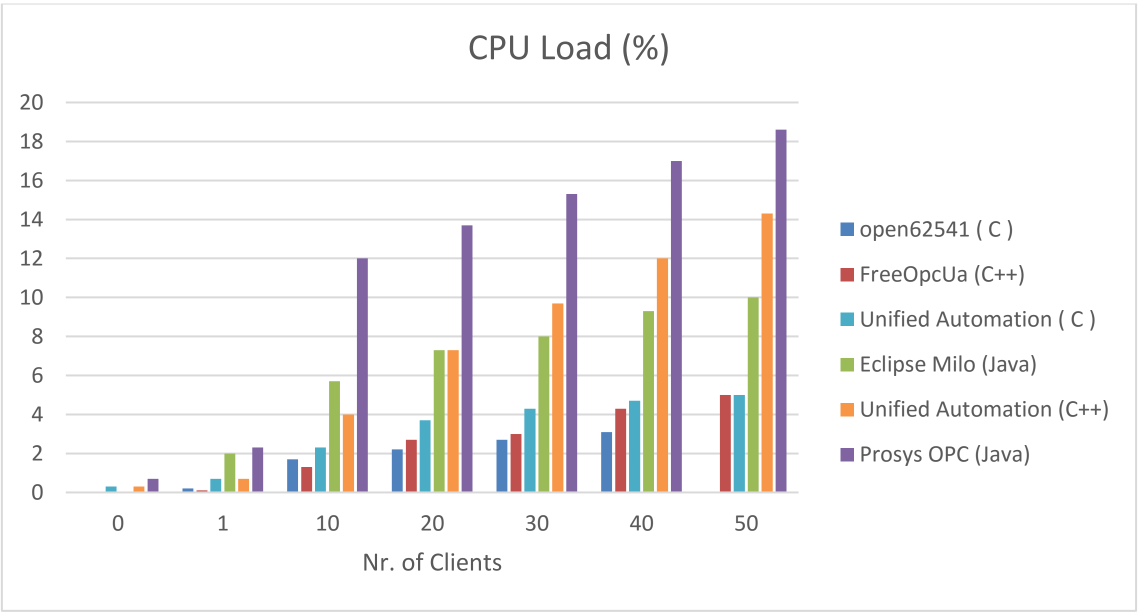


Figure 35 - CPU load, in percentage, of the tested SDK's.

As is possible to observe, all server implementations present a relatively linear increment of the CPU load with the increase of the number of clients connected to them. This fact can be explained by the use of individual TCP socket connections for each client, which consequently increases the number of threads managed by the

[14] Except open62541, which was tested in Windows 10 64-bit.

[15] In the tested implementations, FreeOpcUa C++ limited to a publishing rate of 2 Hz and open62541 was limited to a maximum of 40 clients/sessions.

server. Also, the CPU load is more noticeable in the servers with more functionality (logical operations) as the ones of Prosys OPC and Unified Automation (C++). Conversely, the open-source implementations and Unified Automation (C) shown to be relatively lightweight.

Next, a progression on memory usage is presented, taking into account the same premises as above:

Table 18 - Memory usage, in Megabyte, of the tested SDK's.

Nr. Of clients	**0**	**1**	**10**	**20**	**30**	**40**	**50**
Eclipse Milo (Java)	133.5	302.1	442.4	520.5	562	643.5	655.3
FreeOpcUa (C++)	6.8	6.8	7	7.3	7.5	7.7	8
Open62541 (C)	**1.1**	**1.2**	**1.4**	**1.4**	**1.6**	**1.8**	n/a
Prosys OPC (Java)	299.8	375.5	383.9	372.6	375.2	361.7	346.1
Unified Automation (C)	4.2	4.5	6	7.6	9.2	11	12.4
Unified Automation (C++)	8.7	9.1	10.1	11.1	12.1	13	14

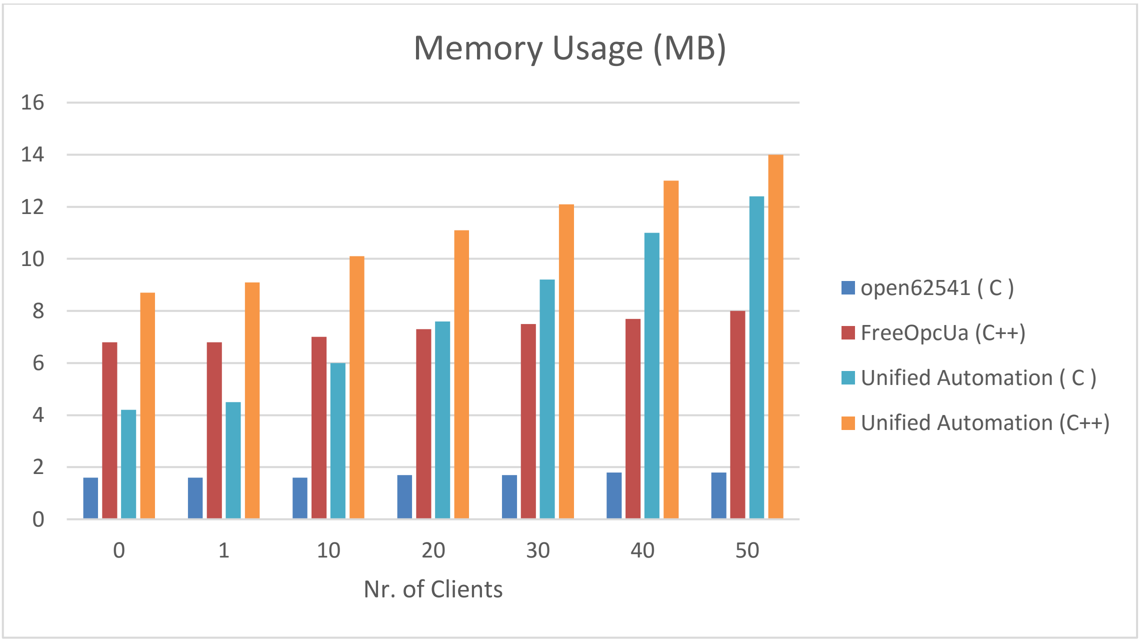


Figure 36 - Memory usage, in Megabyte, of the tested C/C++ SDK's.

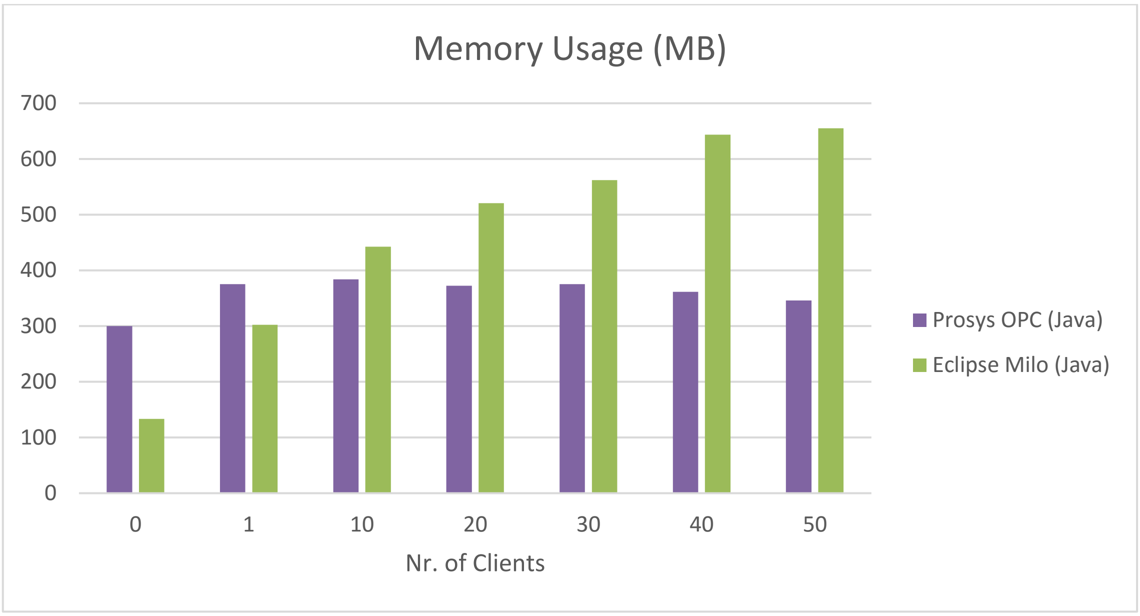


Figure 37 - Memory usage, in Megabyte, of the tested Java SDK's.

Due to the discrepancy between the C/C++ and Java implementations in terms of memory usage, their values were plotted in different charts. In fact, the Java implementations use much more memory, which obviously hamper their use in small embedded devices. Also, although the Prosys OPC does not seem to be affected by the number of clients, the same is not observed in the Eclipse Milo implementation. When looking at the C/C++ implementations, their memory usage is much more restrained. The fact that the memory usage appears to be relatively constant in some implementations might be justified by the use of shared memory to keep track of monitored items, which means that all clients are accessing the same object on the server. Other implementations might not be using this approach, having monitored items replicated by the number of clients. Another aspect that greatly affects memory usage of a server is the extension of the nodeset (i.e., the number of objects exposed by the server) as this nodeset is usually fully loaded in memory upon the initialization of the server.

Latency

Latency was tested using the UaExpert tool from Unified Automation, which is able to estimate the time used per call when reading or writing a server's variable. For this testing, each server was subjected to 10,000 cycles of reading and writing on a variable of type Double. The minimum, maximum and average values of each testing are presented below:

Table 19 - Time per read/write, in milliseconds, of the tested SDK's.

Time per call (ms)	Read				Write		
	Min.	Max.	Avg.		Min.	Max.	Avg.
Eclipse Milo (Java)	0.116	2.634	0.158		0.138	4.243	0.218
FreeOpcUa (C++)	0.094	1.236	0.128		0.082	0.363	0.107
Open62541 (C)	0.021	0.190	**0.028**		0.021	0.270	**0.029**
Prosys OPC (Java)	0.064	3.084	0.092		0.073	4.556	0.159
Unified Automation (C)	0.036	0.220	0.046		0.034	0.235	0.045
Unified Automation (C++)	0.044	0.246	0.060		0.043	0.325	0.062

As it is possible to observe, reading/writing to/from an OPC UA server is not problematic, as the average time is significantly low. This is mainly due to the nature of socket based connections, where handshakes are only needed when establishing a connection. However, it is important to notice that this testing was made in optimal circumstances (local wired network, with no data encryption) so as to not negatively influence any of the testing cycles. Thus, it is expected that these values may increase depending on the network quality or encryption.

4.4.6. OPC UA Solution Analysis

Three free and open source (*FOSS*) and three commercial OPC UA SDK's were tested for this survey. The focus was pinpointed on server functionality, which is the key part for allowing plug-and-produce in a heterogeneous network such as the one considered for the openMOS project (*i.e.*, interoperable with several different protocols and devices). Besides more solutions exist, the tested ones were chosen for being the most advanced (at the time), with better support and active development.

Looking at the *FOSS* implementations, it is clear that they are substantially behind the progress of the standards defined by OPC Foundation, which result in a lack of available functionality, when compared with commercial solutions like Prosys and Unified Automation (see Table 16 for details). Also, available documentation and examples are not very comprehensive, making the understanding of the SDK's a rather ad-hoc experience. From the three tested solutions, open62541 is the better documented and more mature, while Eclipse Milo does not have documentation whatsoever. Moreover, Eclipse Milo is the continuation of the DigitalPetri project, now under the umbrella of the Eclipse IoT movement. Besides having clear potential, it is not ready for production neither for "fast" implementation and deployment of openMOS solutions based on its SDK. However, such *FOOS* solutions may reveal to be more cost-effective if complexity is kept low. For instance, the cost of implementing a Prosys or Unified Automation server inside a simple sensor or actuator for exposing basic data and skills may be prohibitive when compared with the cost of the device itself.

Low-level:

Regarding performance, it is clear that the Java based implementations (Eclipse Milo and Prosys) can hardly be used in embedded devices, mainly due to excessive memory usage.
After this evaluation it seems that, currently, the better choice to implement in an industrial environment, i.e., the industrial automation devices, could be the **open62541** and the **Unified Automation** solutions. While open62541 offers a cost free SDK for simple usage (basic service and security needs), the Unified Automation offers a full featured SDK (with events, alarms and historic data) programmable in both C[16] or C++. Also, the possibility to integrate information models described in XML format (present in both implementations) is a plus for integrating with third parties such as AutomationML.

High-level:

On the higher level, and with less hardware restrictions, the priority should be given to compliance and dependability. This may inevitably demand for a commercial solution where the Java-based **Prosys** SKD could be preferred. This inclination towards a Java-based solution relies on two main conditions: i) offers cross-platform support for a vast set of systems and devices; and ii) is the programming language used for other openMOS high-level software development, such as the agent-based cloud.

[16] Note that the C implementation of Unified Automation does not support file transfer.

5. Comparison of evaluated technology

The main requirement of any IoT system is *interoperability*. The way to obtain this interoperability (the used protocols) is what distinguishes the specifications among IoT architectures. Client-server protocols are best used <u>when the infrastructure of the system is understood</u> whereas publish-subscribe protocols are a better option <u>when the infrastructure is unknown or susceptible to big changes</u>.

Client-Server

This model follows a paradigm where the server holds the data and responds to requests from the clients. Therefore, there is a need for the server to be well-known on the system. If a system has multiple servers (i.e. multiple devices), it must implement an additional discovery feature in order to enable interoperability between them.

- **OPC-UA**

  In this protocol, the client connects, browses, reads and writes to industrial equipment. It defines communication from the application to the transport layer, breaching the interoperability between vendors. Also, OPC-UA security is strong, supporting two-way message signing and transport encryption.

  The base transport of OPC-UA is TCP connection; thus it is not meant for hard real-time. However, it is able to deliver data very fast and with low latency (<1 ms over local networks).

  OPC-UA is a good solution for making PLC and industrial sensory data available for other IoT and IIoT applications where the protocol is already available, or is easy to integrate. Because OPC-UA is industry-orientated, hence more complex than the other available solutions, it is still new to the IT space. Therefore, at the time of this document, the recommended usage of this protocol is to get PLC and sensor data into other existing systems.

- **CoAP**

  This protocol provides interoperability of HTTP with minimal overhead. It uses UDP/multicast and simplifies the HTTP header, reducing the size of each request. CoAP is mostly seen in edge-based devices where HTTP would be too resource intensive. However, CoAP is not recommended for fast and reliable communication. This is mainly due to being datagram oriented, with a simple QoS approach.

Publish-Subscribe

This model requires the participants of the system (devices) to know the place where they can place and retrieve information from. In this scenario, the only entity who needs to be well-known (or discovered) on the system is the broker server. The broker server is responsible for acknowledging and registering all the clients (publishers and subscribers) and the existing topics, in a way that allows any client to discover and communicate with the others.

- **DDS**

  DDS is mainly focused on communication at the edge of the network. Because DDS is an open standard, it has substantial community support in either open source and commercial implementations. DDS is decentralised. Therefore, a participant in a DDS system communicates directly in a peer-to-peer way using multicast. Because DDS centralised network management is optional (it also supports brokers to integrate with the enterprise), it is a faster protocol, reaching sub-millisecond resolution.

  DDS is the best option for reliable and real-time QoS data delivery services, being widely used on fast M2M communications.

- **MQTT**

  MQTT is designed for SCADA and remote networks. It is targeted for lightness, reliable communication and simplicity. MQTT is a mature and open standard originally designed by IBM. Thus, there is a wide availability of open source implementations in almost every programming language.

  MQTT is easy to implement and deploy, and it is highly recommended for systems where the infrastructure is unknown.

5.1. Relationship with the openMOS requirements.

Support all interactions:

All the evaluated technology supports the exchange of different types of data structures, including the widely used XML format. Thus, it is possible to have a "common language", either using a single or different protocols.

Support legacy systems:

The most industry oriented protocol addressed is the OPC-UA, which has already support for a vast number of current industrial devices (such as PLC's).

Real-time or, at least, fast communication:

Regarding real-time capability, DDS is the best choice, as it considered this feature as a design goal. However, OPC-UA is also able to deliver fast and reliable data. CoAP and MQTT may be considered for their simplicity and lightweight, but are not meant to fast data exchange and large payloads.

Service discovery:

All the evaluated technology offers solutions either directly, or using third-party applications, for discovering services on the network, with minimum configuration effort.

Data persistence:

All the evaluated technology offers solutions, either directly, or using third-party applications, for persisting exchanged data.

Single or multiple standards:

All the evaluated technology is interoperable over IP networks. However, an additional translation or re-mapping step may be required for interlocking different protocols.

Synchronous or asynchronous communication:

All the evaluated technology can be configured for both synchronous and asynchronous communication.

Scalability:

All the evaluated technology is easily scalable. However, if using MQTT with a centralised broker, the interactions with the broker may suffer congestion. Also, the loose QoS control of CoAP may lower the efficiency of deliver within congested networks.

Offline functionality:

Offline functionality must be implemented in the MSB logical layer independently of the protocol being used. Nonetheless, all standards can be deployed in local networks.

Security:

All the evaluated technology supports authentication, authorisation and encryption, being the OPC-UA the stronger in this aspect, with more possibilities.

Constrained network environments:

Due to their lightweight nature, CoAP and MQTT are clearly the more prone for this type of environments. However, all evaluated technology should perform well in constrained environments, depending on the level of complexity implemented.

Conflicting requirements:

The conflicting requirements should be addressed directly in the MSB logical layer, independently of the protocols being used.

6. Conclusions

This document had the main purpose of assessing candidate technology to be incorporated into the manufacturing service bus that will be developed in the scope of the openMOS project.

First, it was made a comprehensive analysis on the requirements addressed by the consortium. This analysis enabled to identify the most important features to consider when evaluating M2M communication protocols.

Thereafter, it was shown an overview on methods and protocols commonly used in networked systems. This overview permitted to recognise the individual building blocks that are often put together to build a full featured communication stack.

Finally, the candidate technology, which already provides different types of services, was evaluated, namely four open standards were considered: CoAP, DDS, MQTT and OPC UA. These standards were compared (see Section 5) in order to give a set of advantages/disadvantages, as well as a reflection on which target scenarios each one may better adapt and perform.

From the evaluation, it was concluded that all the tested protocols have features which fit within the OpenMOS project. OPC-UA is suitable for integrating data provided by industrial PLCs and sensors. DDS is optimized for scenarios where the M2M communication require a fast, reliable and real time data exchange. MQTT and CoAP are most suitable for higher level communication, such as database interaction and communication with agents.

7. References

[1] The openMOS consortium, *openMOS - Description of Work,* Proposal 680735 for the EU Horizon 2020 framework programme, 2015.

[2] The openMOS consortium, *openMOS - D1.1: Industrial Requirements for Plug&Produce Systems and Components,* Public report, 2016.

[3] The openMOS consortium, *openMOS - D1.3: Detailed Review and Evaluation of Relevant Strandardisation Activities Report,* Public report, 2016.

[4] Z. Shelby, ARM, K. Hartke and C. Bormann, *The Constrained Application Protocol (CoAP),* Internet Engineering Task Force (IETF), 2014.

[5] C. Bormann, A. P. Castellani and Z. Shelby, "CoAP: An Application Protocol for Billions of Tiny Internet Nodes," *IEEE Internet Computing,* vol. 12, pp. 1089-7801, 2012.

[6] C. V. A. B. X. H. J. Z. Nanxing Chen, "Ensuring Interoperability for the Internet of Things: Experience," 2013.

[7] O. DDS, 6 6 2016. [Online]. Available: http://opendds.org/documents/architecture.html.

[8] "Transports," OpenDDS, [Online]. Available: http://opendds.org/about/transport.html. [Accessed 15 9 2016].

[9] W. Stallings, "The HMAC Algorithm," 1 April 1999. [Online]. Available: http://www.drdobbs.com/security/the-hmac-algorithm/184410908. [Accessed 15 9 2016].

[10] "Specification 1.4," OMG, [Online]. Available: http://www.omg.org/spec/DDS/1.4/PDF/. [Accessed 16 9 2016].

[11] "DDS interoperability specification," [Online]. Available: http://www.omg.org/spec/DDSI/2.1/PDF/.

[12] "DDSI-RTPS 2.2," object management group, 01 09 2014. [Online]. Available: www.omg.org/spec/DDSI-RTPS/2.2/. [Accessed 23 09 2016].

[13] "XMI," OMG, [Online]. Available: http://www.omg.org/spec/XMI/. [Accessed 16 9 16].

[14] "Modeling SDK," OCI, [Online]. Available: http://opendds.org/about/modeling/. [Accessed 16 9 2016].

[15] Eclipse, [Online]. Available: http://www.eclipse.org/modeling/emf/. [Accessed 16 9 2016].

[16] “Vortex OpenSplice Modeler,” Prismtech, [Online]. Available: http://www.prismtech.com/vortex/vortex-opensplice/tools/modeler. [Accessed 16 9 2016].

[17] [Online]. Available: http://www.twinoakscomputing.com/coredx/performance_network. [Accessed 27 09 2016].

[18] [Online]. Available: http://opendds.org/perf/lab100125/latency_results.html. [Accessed 27 09 2016].

[19] [Online]. Available: http://www.prismtech.com/vortex/vortex-opensplice/performance. [Accessed 27 09 2016].

[20] [Online]. Available: https://www.rti.com/products/dds/benchmarks.html. [Accessed 27 09 2016].

[21] L. Zhang, “Building Facebook Messenger,” 12 August 2011. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/building-facebook-messenger/10150259350998920/. [Accessed 22 September 2016].

[22] J. Barr, “AWS IoT - Cloud Services for Connected Devices,” 8 October 2015. [Online]. Available: https://aws.amazon.com/blogs/aws/aws-iot-cloud-services-for-connected-devices/. [Accessed 22 September 2016].

[23] OpenStack Infrastructure Team, “OpenStack Infra Firehose,” [Online]. Available: http://docs.openstack.org/infra/system-config/firehose.html. [Accessed 21 September 2016].

[24] OASIS, “MQTT Version 3.1.1,” 29 October 2014. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. [Accessed 22 September 2016].

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

[26] G. Cândido, F. Jammes, J. B. de Oliveira and A. W. Colombo, “SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications,” in *2010 8th IEEE International Conference on Industrial Informatics*, Osaka, Japan, 2010.

[27] “OPC Foundation - The Industrial Interoperability Standard,” OPC Foundation, [Online]. Available: https://opcfoundation.org/. [Accessed 24 8 2016].

[28] “Milo,” The Eclipse Foundation, [Online]. Available: https://projects.eclipse.org/projects/iot.milo. [Accessed 24 8 2016].

[29] “FreeOpcUa: Open Source C++ and Python OPC-UA Server and Client Libraries and Tools,” [Online]. Available: https://freeopcua.github.io/. [Accessed 24 8 2016].

[30] “Build OPC UA applications in JavaScript and NodeJS,” [Online]. Available: http://node-opcua.github.io/. [Accessed 24 8 2016].

[31] “opcua4j,” [Online]. Available: https://github.com/gavioto/opcua4j. [Accessed 24 8 2016].

[32] “open62541,” TU Dresden PLT, Fraunhofer IOSB, RWTH-PLT, [Online]. Available: http://open62541.org/. [Accessed 24 8 2016].

[33] “openSCADA,” IBH Systems GmbH, [Online]. Available: http://openscada.org/. [Accessed 24 8 2016].

[34] “UAF - the Unified Architecture Framework,” [Online]. Available: http://uaf.github.io/uaf/. [Accessed 24 8 2016].

[35] “Build a free HMI using OPC Unified Architecture (UA) and Visual Studio.,” [Online]. Available: https://github.com/convertersystems/workstation-uaclient. [Accessed 24 8 2016].

[36] “MatrikonOPC,” Honeywell International Inc., [Online]. Available: http://www.matrikonopc.com. [Accessed 24 8 2016].

[37] “Prosys OPC,” Prosys PMS Ltd, [Online]. Available: https://www.prosysopc.com/. [Accessed 24 8 2016].

[38] “Softing,” Softing Industrial Automation GmbH, [Online]. Available: http://industrial.softing.com. [Accessed 24 8 2016].

[39] “Unified Automation,” Unified Automation GmbH, [Online]. Available: https://www.unified-automation.com. [Accessed 24 8 2016].

[40] “KEPServerEX,” Kepware, Inc., [Online]. Available: https://www.kepware.com/products/kepserverex/. [Accessed 24 8 2016].

[41] “CommServer,” CAS, [Online]. Available: http://www.commsvr.com. [Accessed 24 8 2016].

[42] The openMOS consortium, *openMOS - D2.1: Plug&Produce Automation Device Specification,* 2016.

[43] “The Constrained Application Protocol (CoAP),” [Online]. Available: https://tools.ietf.org/html/rfc7252#section-4.6http://.

[44] “Paho,” [Online]. Available: http://www.eclipse.org/paho/.

[45] “Mosquitto,” [Online]. Available: http://mosquitto.org/.