



TAMPERE UNIVERSITY OF TECHNOLOGY

JOHANNES MINOR

**BRIDGING OPC UA AND DPWS FOR INDUSTRIAL SOA**

MASTER OF SCIENCE THESIS

Examiner: Professor Jose L. Martinez Lastra

Examiners and topic approved in the  
Automation, Mechanical and Materials  
Engineering Faculty Council Meeting on  
04.05.2011

## **ABSTRACT**

TAMPERE UNIVERSITY OF TECHNOLOGY

Master of Science Degree Programme in Machine Automation

**MINOR, JOHANNES:** Bridging OPC UA and DPWS for Industrial SOA

Master of Science Thesis, 80 pages, 9 Appendix pages.

February 2012

Major: Factory Automation.

Examiner: Prof. José Luis Martínez Lastra.

Keywords: Complex Event Processing, Devices Profile for Web Services, Event Driven Architecture, OPC UA, SCADA, Service-Oriented Architecture, Web Services

Two web-service based specifications, OPC Unified Architecture (OPC UA) and Devices Profile for Web Services (DPWS), have been proposed by various researchers and organizations as possible enabling technologies for an event-driven Service Oriented Architecture for monitoring and control in manufacturing applications. This paper aims to propose and demonstrate an approach for bridging these two technologies in a way that is applicable in existing industrial applications.

A merger between OPC UA and DPWS that effectively combines their complementary strengths could help pave the path toward future industrial event-driven SOA applications, with the inherent modularity, agility, and interoperability envisioned by researchers today.

A representation of DPWS devices, services, operations and events in the OPC UA data model is proposed, and a DPWS Module is developed for Ignition, a commercially available HMI/SCADA and MES platform with integrated OPC UA Server. The module discovers DPWS devices in a local network, creates the representation in the address space, and handles subscriptions, input and output parameter values, and invoking operations. A Complex Event Processing component based on Microsoft's StreamInsight is also integrated with the system, input and output adapters exposing web service interfaces.

The system prototype developed will be used as the base for a use case demonstrator in the European Commission's Framework Package 7 Project, "Architecture for Service-Oriented Process Monitoring and Control (IMC AESOP)." The project aims to develop a system of systems approach for monitoring and control, based on SOA for very large-scale systems in the process industries.

## **PREFACE**

The work described in this Thesis was performed at the Factory Automation Systems and Technologies Lab (FAST) in the Department of Production Engineering at Tampere University of Technology, under the direction of Prof. Dr. José Luis Martínez Lastra, and Associate Professor Andrei Lobov.

Funding for this research came from the European Commission's Framework Package 7 Project, "Architecture for Service-Oriented Process Monitoring and Control (IMC AESOP)."

Many thanks to Prof. Lastra and Dr. Lobov for giving me the opportunity to work in FAST Lab, and for the guidance they provided while I studied and completed my degree. A warm thank you as well to all the other FAST Lab members, and the staff that helped me out when I needed it, including Matti, Sonja, Hanna, and Taina.

Thank you Jill, for your good advice and encouraging words these last few years. Thanks also to Jorge and Jaacan for keeping life in the lab entertaining, for providing an outlet for exchanging ideas and venting grievances, and for motivating me when I failed to motivate myself.

Finally, thank you to my family for all the love and support over the years, and to all friends in Thunder Bay, Vancouver, Tampere, and elsewhere for giving me confidence and enriching my life.

Tampere, February 14, 2012.

Johannes Minor

## CONTENTS

<b>1</b>	<b>INTRODUCTION.....</b>	<b>1</b>
1.1	BACKGROUND .....	1
1.2	PROBLEM DEFINITION .....	2
1.2.1	<i>Problem Statement</i> .....	2
1.2.2	<i>Justification of the Work</i> .....	2
1.3	WORK DESCRIPTION .....	3
1.3.1	<i>Objectives</i> .....	3
1.3.2	<i>Methodology</i> .....	3
1.4	THESIS OUTLINE.....	3
<b>2</b>	<b>TECHNOLOGY OVERVIEW.....</b>	<b>5</b>
2.1	SERVICE-ORIENTED ARCHITECTURE.....	5
2.1.1	<i>Architectural Tenets</i> .....	5
2.1.2	<i>Advantages of SOA</i> .....	7
2.1.3	<i>SOA Design Methodologies</i> .....	8
2.2	WEB SERVICES.....	10
2.2.1	<i>SOAP</i> .....	10
2.2.2	<i>Web Services Description Language</i> .....	12
2.2.3	<i>Devices Profile for Web Services</i> .....	16
2.3	EVOLUTION OF SCADA SYSTEMS .....	21
2.4	SOA IN INDUSTRIAL APPLICATIONS.....	23
2.4.1	<i>Devices Profile for Web Services in Industry</i> .....	24
2.4.2	<i>OPC Unified Architecture</i> .....	25
2.4.3	<i>OPC UA Companion Specifications</i> .....	36
2.5	EVENT-DRIVEN ARCHITECTURE AND EVENT-DRIVEN SOA .....	38
2.5.1	<i>Complex Event Processing</i> .....	40
2.6	OWL WEB ONTOLOGY LANGUAGE .....	41
<b>3</b>	<b>APPROACH AND METHODOLOGY.....</b>	<b>49</b>
3.1	MERGING OPC-UA AND DPWS .....	49
3.1.1	<i>Comparison Between the Technologies</i> .....	49
3.1.2	<i>Adoption in Industry</i> .....	50
3.1.3	<i>Technology Merging Strategies</i> .....	51
3.1.4	<i>Chosen Approach</i> .....	54
3.2	COMPONENT SELECTION .....	55

3.2.1	<i>OPC UA Client and Server SDKs</i> .....	55
3.2.2	<i>Ignition OPC UA Server</i> .....	56
3.2.3	<i>The Ignition SDK</i> .....	58
3.2.4	<i>JMEDS WS4D DPWS Java Stack</i> .....	59
3.2.5	<i>InicoTech S1000 Smart RTU</i> .....	59
3.3	PROPOSED INTEGRATION APPROACH .....	60
3.3.1	<i>Mapping WSDL to OPC UA Address Space</i> .....	60
<b>4</b>	<b>IMPLEMENTATION</b> .....	<b>63</b>
4.1	SYSTEM OVERVIEW .....	63
4.2	DPWS MODULE FOR IGNITION .....	63
4.2.1	<i>Ignition Designer Interface</i> .....	66
4.3	MICROSOFT STREAMINSIGHT COMPONENT .....	69
4.4	OVERALL SYSTEM STRUCTURE .....	71
<b>5</b>	<b>DISCUSSION OF RESULTS</b> .....	<b>74</b>
5.1	TESTING .....	74
5.2	ASSESSMENT .....	75
5.3	NEXT STEPS .....	76
<b>6</b>	<b>CONCLUSION</b> .....	<b>77</b>
<b>7</b>	<b>REFERENCES</b> .....	<b>78</b>
	<b>APPENDIX A: OPC UA EVENTING MECHANISM</b> .....	<b>83</b>
	<b>APPENDIX B: INSTRUCTIONS FOR IGNITION MODULES</b> .....	<b>87</b>
	INSTALLING AND SETTING UP IGNITION AND THE DPWS DRIVER .....	87
	BUILDING MODULES .....	88
	DPWS DRIVER PROJECT OVERVIEW .....	88
	CEP OUTPUT ADAPTER SINK PROJECT OVERVIEW .....	88
	BUILDING THE STREAMINSIGHT PROJECT FOR VISUAL STUDIO (C#) .....	89

## LIST OF FIGURES

FIGURE 1: THREE LAYERS OF SERVICE ABSTRACTION .....	6
FIGURE 2: A SERVICE-BASED BUSINESS APPLICATION, COMPOSED OF ATOMIC AND COMPOSITE SERVICES.....	6
FIGURE 3: THE SOMA METHOD[34] .....	9
FIGURE 4: SOAP ENVELOPE [3] .....	11
FIGURE 5: WSDL 1.1 AND 2.0 DOCUMENT STRUCTURE[14].....	13
FIGURE 6: ARRANGEMENT OF DPWS CLIENTS, DEVICES, AND SERVICES[10] .....	16
FIGURE 7: NETWORK PROTOCOLS AND SPECIFICATIONS INCLUDED IN THE DEVICES PROFILE FOR WEB SERVICES[25] .....	20
FIGURE 8: EVOLUTION OF SCADA SYSTEMS [33] .....	22
FIGURE 9: OPC UA STACKED ARCHITECTURE .....	27
FIGURE 10: OPC UA NODE MODEL[38] .....	28
FIGURE 11: OPC UA OBJECT MODEL[39] .....	30
FIGURE 12: A VIEW OF THE TOP-LEVEL NODES OF AN IGNITION OPC-UA SERVER [43], AS SEEN FROM UNIFIED AUTOMATION UAEXPERT OPC UA CLIENT[42].....	32
FIGURE 13: OPC UA WEB SERVICES STACK.....	36
FIGURE 14: OPC UA OBJECT TYPES INTRODUCED BY OPC UA FOR DEVICES COMPANION SPECIFICATION [49].....	37
FIGURE 15: OPC UA DEVICES EXAMPLE [48] .....	38
FIGURE 16: OPC UA - DPWS CONVERGENCE PROTOTYPE [47] .....	52
FIGURE 17: DPWS/OPC UA ARCHITECTURE FOR EVENT PROCESSING .....	54
FIGURE 18: IGNITION OPC UA SERVER CONNECTIONS .....	57
FIGURE 19: GENERIC MAPPING FROM DPWS TO THE OPC UA FOR DEVICES OBJECT MODEL.....	61
FIGURE 20: IGNITION DPWS MODULE: SYSTEM OVERVIEW .....	64
FIGURE 21: SIMPLIFIED DPWS TO OPC UA DEVICE REPRESENTATION FOR IGNITION OPC UA SERVER .....	65
FIGURE 22: DPWS MODULE FOR IGNITION SERVER - VIEW FROM DESIGNER .....	66
FIGURE 23: OPC UA ADDRESS SPACE AND SQLTAGS BROWSER PANELS IN IGNITION DESIGNER .....	67
FIGURE 24: IGNITION CLIENT HMI .....	68
FIGURE 25; VIEW OF ADDRESS SPACE OF IGNITION OPC UA SERVER FROM A THIRD PARTT OPC UA CLIENT .....	68
FIGURE 26: DIAGRAM OF STREAMINSIGHT CEP COMPONENT PERFORMING DPWS EVENT FILTERING .....	69
FIGURE 27: CONSOLE APPLICATION EXECUTING STREAMINSIGHT QUERIES .....	71
FIGURE 28: IGNITION OPC UA ADDRESS SPACE, AS SEEN IN UAEXPERT.....	71
FIGURE 29: DIAGRAM OF COMPLETE PROTOTYPE SYSTEM, SHOWING INTERACTIONS AND IMPORTANT CLASSES .....	72
FIGURE 30: CREATING AN HMI IN THE IGNITION DESIGNER .....	75
FIGURE 31: EVENT REFERENCE EXAMPLE [39] .....	84
FIGURE 32: MONITORED ITEM MODEL [44] .....	85

## LIST OF TABLES

TABLE 1: SOAP FAULT CODES .....	11
TABLE 2: STRUCTURE OF WSDL 1.1 AND WSDL 2.0 FILES .....	13
TABLE 3: WSDL-S EXTENSIBILITY ATTRIBUTES AND ELEMENTS .....	15
TABLE 4: SPECIFICATIONS IN THE DEVICES PROFILE FOR WEB SERVICES.....	17
TABLE 5: ATTRIBUTES COMMON TO ALL OPC UA NODES.....	29
TABLE 6: OPC UA SERVICE SETS[44] .....	32
TABLE 7: OBJECTTYPES DEFINED IN THE OPC UA FOR DEVICES COMPANION SPECIFICATION .....	37
TABLE 8: STACK OF W3C RECOMMENDATIONS RELATED TO THE SEMANTIC WEB .....	41
TABLE 9: OWL SUB-LANGUAGES [40] .....	42
TABLE 10: OWL LANGUAGE CONSTRUCTS .....	44
TABLE 11: COMMERCIAL OPC UA CLIENT AND SERVER SDKS.....	55
TABLE 12: MAPPING BETWEEN OPC UA PRIMITIVE TYPES AND XML SCHEMA TYPES [55].....	61
TABLE 13: EVENT SOURCE REFERENCE TYPES .....	84

## LIST OF ABBREVIATIONS

6LoWPAN	Internet Protocol version 6 over Low-power Wireless Personal Area Networks
API	Application Programming Interface
ANSI	American National Standards Institute
AWT	Java AWT: Abstract Window Toolkit
BPEL	Business Process Execution Language
BPM	Business Process Management
BPMN	Business Process Model and Notation
CDLC	Connected Limited Device Configuration (Java)
CEP	Complex Event Processing
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CRM	Customer Resource Management
DCOM	Distributed Component Object Model
DCS	Distributed Control System
DDS	Data Distribution Service for Real Time Systems
DPWS	Devices Profile for Web Services
EA	Enterprise Architecture
EDA	Event-Driven Architecture
EDDL	Electronic Device Description Language
EPL	Event Patterning Language
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
EXI	Efficient XML Interchange
FDI	Field Device Integration
FDT	Field Device Tool
GUID	Globally Unique Identifier
HMI	Human-Machine Interface
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
IMC-AESOP	ArchitecturE for Service-Oriented Process - Monitoring and Control
IEC	International Electrotechnical Commission
IP	Internet Protocol
IPv4	IP version 4
IPv6	IP version 6
ISA	International Society of Automation
ITEA	Information Technology for European Advancement
J2SE	Java 2 Platform Second Edition
JBOWS	Just a Bunch of Web Services



JMEDS	Java Multi-Edition DPWS Stack
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LAN	Local Area Network
LINQ	Language Integrated Query
MEP	Message Exchange Pattern
MES	Manufacturing Execution System
MOM	Manufacturing Operations Management
OASIS	Organization for the Advancement of Structured Information Standards
OLE	Object Linking and Embedding
OMG	Object Management Group
OPC	OLE for Process Control (Obsolete Acronym)
OPC UA	OPC Unified Architecture
OSGi	Open Services Gateway initiative framework
OWL	OWL Web Ontology Language
OWL DL	OWL Description Logics
PLC	Programmable Logic Controller
PROFIBUS	Process Field Bus
PROFIBUS-DP	PROFIBUS Decentralized Peripherals
PROFIBUS-PA	PROFIBUS Process Automation
QoS	Quality of Service
RDF	Resource Description Framework
RDFS	RDF Schema
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RT	Real-Time
RTU	Remote Terminal Unit
SCA	Service Component Architecture
SCADA	Supervisory Control and Data Acquisition System
SIRENA	Service Infrastructure for Real Time Embedded Networked Applications
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOA4D	SOA for Devices
SOAP	Simple Object Access Protocol (Obsolete Acronym)
SOCRADES	Service Oriented Cross-Layer Infrastructure For Distributed Smart Embedded Devices
SOI	Service-Oriented Integration
SOMA	Service-Oriented Modeling and Architecture, from IBM
SPARQL	SPARQL Protocol and RDF Query Language

SPOF	Single Point of Failure
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDDI	Universal Description Discovery and Integration
UDP	Universal Datagram Protocol
UML	Unified Modeling Language
UPnP	Universal Plug And Play
URI	User Resource Identifier
URL	Uniform Resource Locator, or Universal Resource Locator
UsiXML	User Interface Extensible Markup Language
W3C	World Wide Web Consortium
WAN	Wide Area Network
WBF	World Batch Forum
WS	Web Service
WS4D	Web Services For Devices
WS-*	Collectively refers to Web Service-related standards
WS-BPEL	Web Services Business Process Execution Language
WS-I	Web Services Interoperability Organization
WSDAPI	Web Services on Devices API
WSDL	Web Services Description Language
WSDL-S	Web Service Semantics
WSRF	Web Services Resource Framework
XML	eXtensible Markup Language
XSD	XML Schema

# 1 INTRODUCTION

## 1.1 BACKGROUND

The Service-Oriented Architecture (SOA) Paradigm has been applied successfully in IT systems for several years. Many benefits have been achieved by designing complex enterprise systems from the ground up, as a set of loosely-coupled, combinable, services that expose business-relevant functionality through a well-defined interface. Service-oriented systems have been demonstrated to be agile, reconfigurable, and scalable. Message-oriented communication using standard network protocols and encoding enables easier integration of heterogeneous systems, and the availability of real-time information from all parts of the enterprise has enabled businesses to quickly respond to changing market conditions.

Manufacturing enterprises depend on complex business processes, often spanning globally distributed production systems and supply chains. Minimizing downtime for equipment integration and reconfiguration, and maximizing visibility throughout all enterprise layers are essential for maintaining business agility, and responding to a dynamic market. SOA is seen by some as a strong candidate solution for enabling the modularity, interoperability, and fast reconfigurability needed to achieve these goals. Furthermore, extending SOA down to the device level has become feasible, due to the increasing presence of networked embedded devices on the shop floor, sufficiently sophisticated to act autonomously, and to collaborate with other devices.

A responsive system requires that interested parties (devices and subsystems) be informed of notable events inside or outside the manufacturing enterprise with minimal latency. In large-scale factory monitoring systems, intelligent devices on the factory floor must be capable of reading values from sensors, controlling actuators, and performing some limited filtering and processing. In an Event-Driven Architecture (EDA), devices and subsystems must also be able to asynchronously push notifications of changes or alarms to higher-level systems. Powerful tools, such as Complex Event Processing (CEP) engines, can be used to derive high-level information about the present and future health of a system from analysis of these low-level, atomic events.

Much research effort has been spent on applying the concepts of SOA and EDA to creating an event-driven SOA for manufacturing systems.

Two Web-Service based approaches have been proposed as strong candidates for industrial SOA: DPWS and OPC UA. Devices Profile for Web Services (DPWS) was designed to enable Universal Plug and Play (UPnP) –like functionality for networked devices using Web Service technologies. The profile defines a minimal set of implementation requirements for dynamic discovery, service description, secure messaging, and events and subscriptions. DPWS-compliant devices will expose their capabilities or data as a set of custom hosted services, or pre-configured events,

exposing a service endpoint with an interface described in Web Services Description Language (WSDL). OPC Unified Architecture (OPC UA) is the Web Service-based evolution of classic OPC, a standard for accessing device data over COM on Windows platforms. The standard defines a rich data model, and a fixed set of services for navigating, reading, and modifying an OPC UA server's address space.

The two technologies have complementary strengths. OPC UA was designed from the ground up for security, and the specification defines a data model for enriching raw data with semantics. OPC UA can be used across networks and through firewalls, and is better suited to exposing simple device or Programmable Logic Controller (PLC) memory or physical IOs to client applications. OPC UA servers are typically found in ISA 95 Layer 3 (Manufacturing Execution Systems, MES). DPWS is lighter-weight, supports dynamic discovery in local networks, and can be composed into higher-level services using orchestration or choreography standards, such as WS-BPEL (Business Process Execution Language for Web Services) or WS-CDL (Web Services Choreography Description Language). DPWS is well suited to devices at the lowest enterprise levels, such as systems at layers 2 and 1 (controllers, devices, and sensors/actuators) of the ISA 95 Enterprise hierarchy model, but the web service interface means that device access is possible even from the highest levels.

## **1.2 PROBLEM DEFINITION**

### **1.2.1 Problem Statement**

Two web-service based specifications, Devices Profile for Web Services (DPWS) and OPC Unified Architecture (OPC UA), have been proposed by various researchers and organizations as possible enabling technologies for an event-driven Service Oriented Architecture for monitoring and control in manufacturing applications. This research aims to propose and demonstrate an approach for merging these two technologies in a way that is applicable in existing industrial applications.

### **1.2.2 Justification of the Work**

Neither technology alone is sufficient for realizing all the advantages promised by industrial SOA proponents. OPC UA approaches the problem from a "web services for integration," rather than an architectural point of view, while DPWS currently lacks the sophisticated tools, clearly-defined adoption roadmap and reference architecture, and standard data and security models required by system designers and integrators to confidently deploy a full-fledged, large scale SOA across all levels of a production enterprise. A successful merger between these two technologies would leverage their combined strengths, and may help pave the path toward future industrial SOA applications, with the inherent modularity, agility, and interoperability envisioned by researchers today.

## **1.3 WORK DESCRIPTION**

### **1.3.1 Objectives**

The objectives of this work is as follows:

1. Propose a practical approach for integrating two web service-based standards, OPC UA and DPWS, to leverage their respective strengths for creating a Service Oriented Architecture for monitoring and control of process plant and manufacturing systems
2. Implement a proof-of-concept system, incorporating DPWS-enabled devices, an OPC UA client and server, and a CEP engine with web service input and output adapters.

### **1.3.2 Methodology**

The approach followed to achieve the research and development objectives are:

- Perform an extensive review of the DPWS and OPC UA specifications
  - Determine current best-practices for systems implementing the standards,
  - Investigate usage scenarios for both specifications, including application domains of existing deployments, and relative adoption rates
  - Review previous published research on bridging DPWS and OPC UA
- Propose a new approach for integrating DPWS and OPC UA, appropriate in the context of existing systems
- Investigate and evaluate available open source and commercial solutions for the relevant technologies: OPC UA Clients and Servers, DPWS client and device stacks, Complex Event Processing engines.
- Implement and demonstrate proof-of-concept system, integrating the two technologies

## **1.4 THESIS OUTLINE**

Chapter 2 introduces and discusses relevant background knowledge; theories, specifications, and technologies, including SOA, EDA, DPWS, and OPC UA. It also describes the state of the art in SOA for industrial applications. Chapter 3 describes the methodology, including development platform selection and approach for technology bridging. Chapter 4 documents the proposed approach for merging OPC UA and

DPWS. The proof-of-concept system is presented in Chapter 6, and conclusions in Chapter 7.

## 2 TECHNOLOGY OVERVIEW

### 2.1 SERVICE-ORIENTED ARCHITECTURE

Service-Oriented Architecture (SOA) is an architecture style for building autonomous, interoperable, agile systems. The term describes a flexible set of design principles for use in systems development and integration, whereby heterogeneous systems expose their functionality as a set of granular, loosely-coupled services with well-defined, standards-compliant interfaces, which can be used across multiple business domains. Autonomy and interoperability are contradictory properties. One of the challenges of SOA is, therefore, to reconcile these opposing principles [7].

Components of a Service-oriented system are implemented independently, and have some capabilities that are abstracted and exposed as services. Services provide no API, but rather a description of the functionality and protocols. Other components that are aware of the service interface can use the capabilities provided, and large, complex applications can be strung together using existing services.

#### 2.1.1 Architectural Tenets

A service-oriented system is designed according to the following core principles:

- Encapsulation**      The service implementation is opaque to the service consumer. All relevant details about the results of invoking the service or the quality of service are outlined in the service contract.
- Granularity**      Services expose a coarse-grained piece of business functionality. Generally, the preference is for a small number of operations with complex input and output messages. Fine grained services are used to help realize the higher-level services.
- Autonomy**      Services have control over the logic they encapsulate, and each service implementation is independent of other services. Services contain no embedded calls to each other, and will not fail if other services fail.
- Service Contract**      Services adhere to a well-defined communications agreement a Service Contract, which provides unambiguous information about a service's functionality, message formats and exchange patterns, and acceptable communication protocols. Service policies can also include non-functional information, such as Quality of Service

(QoS), security information, and semantic requirements.

### Loose-Coupling

In a loosely-coupled system, a service requester has no knowledge of the internal implementation details of a service provider. A service's functionality is exposed at its boundary, and described in an interface contract. The functionality can be provided by any component that implements the interface, and can be replaced without affecting the dependent component. This promotes agility and reuse.

### Abstraction

SOA design is business process-centric, not technology-centric. A service is an abstracted, logical view of an actual program, database or business process. There are three layers of abstraction:

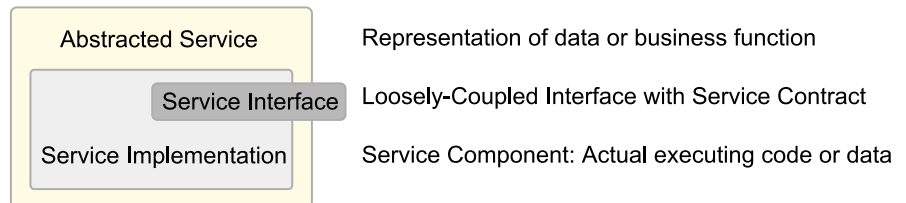


Figure 1: Three layers of service abstraction

### Composability

Atomic services are just abstract implementations of a service. Atomic Services can be composed into composite services, which represent a more complex business process. These processes can themselves be exposed as services. Composite services appear as atomic services to service consumers, because any knowledge of the service implementation would violate the rules of loose-coupling.

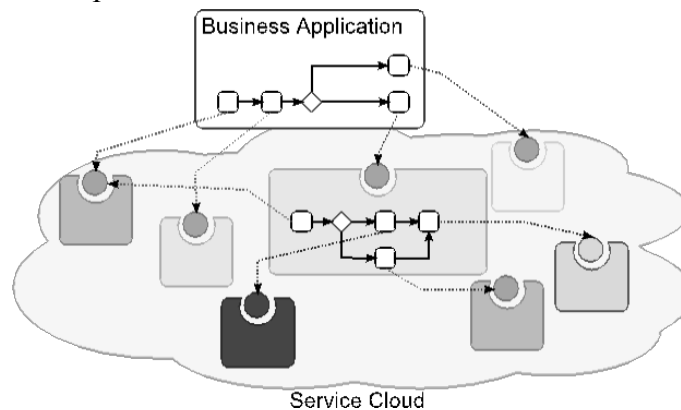


Figure 2: A Service-Based Business Application, composed of atomic and composite services

### Reusability

A set of granular, loosely couple services can be re-used in a variety of business processes across a variety of business contexts. In the SOA design phase, a balance must be struck between coarse granularity and reusability.



Some additional principles extend the concept of Service Oriented Architecture to provide further benefits:

<b>Discoverability</b>	Services are outwardly descriptive, and accessible via available discovery mechanisms. Once discovered service consumers can bind to the services and invoke them
<b>Message-Orientation</b>	Communication is based on a conversation-style exchange of document-style messages. Services are formally defined in terms of the message exchanges between service providers and consumers.
<b>Asynchronous Communication</b>	Loose-coupling and message-orientation enables asynchronous communication between services. Actions are invoked by sending a message, and responses, if any, are returned without requiring the invoking entity to suspend execution. This offers more scalability than RPC.
<b>Platform-Neutral</b>	Messages are sent in a standardized format, not tied to any particular platform, operating system, or programming language.
<b>Reliance on Open Standards</b>	Where possible, services are implemented using ubiquitous, open standards for data representation and transport protocols (XML, HTTP, TCP/IP)

Service-Oriented Architectures are well suited to applications distributed across a network. When service-oriented applications are implemented in accordance with the above-mentioned principles, using well established, widely used communication protocols and encoding standards, many benefits can be achieved.

### 2.1.2 Advantages of SOA

A well-implemented service-oriented system can yield a number of benefits over traditional systems:

- **Integration Capability**  
Systems can be designed to use standard network infrastructure. The abstraction between the implementation and the interface means that services can be implemented on any hardware or software platform. Services can be easily integrated with other services, and composed into higher-level services using some process description tools. A gateway or mediator can provide a service interface to expose the functionality of legacy systems.
- **Agility, Flexibility, Reconfigurability**  
The aim of reconfigurable manufacturing systems is to compose and execute several atomic, re-usable processes in given sequences in order to create

complex processes of a higher order [8]. The SOA approach also enables incremental deployment, and scalability. Ease of integration also implies ease of reconfiguration. When business requirements change and new services are created, the interoperable components can be re-combined using new logic with minimal effort. Service implementations can be changed without affecting service consumers, who are only aware of the service interface. Complex information can be exchanged peer-to-peer, creating a more responsive and adaptive system with better decision-making abilities.

- **Reduced Development Costs**

Simplified integration and reconfiguration leads to faster system setup, and shorter downtime when redesign is required. This leads to lower development costs, and more uptime. Improved integration capability also allows organizations to choose the best-of-breed from all vendors for all components. The service-oriented model facilitates the development of applications by providing coarse-grained services that encapsulate clearly defined tasks. The task-oriented paradigm reduces application development to workflow sequencing and the coordination of subtasks [5].

- **Encapsulated Complexity**

Internal implementation complexities are encapsulated in the service, and complex processes and combinations of services can also be exposed as services.

- **Programming at high level of abstraction**

Services are abstracted business processes. By following a business-centric service identification approach and defining services using the business domain vocabulary, services can be combined and recombined into business processes using relatively simple flowchart-like programming languages.

- **Fault-Tolerance**

A Single Point of Failure (SPOF) is a part of a system that, if failed, will cause the whole system to fail. Systems made up of autonomous, loosely-coupled components are inherently more fault-tolerant than traditional tightly-coupled systems, because there are fewer SPOFs. Service implementations are replaceable, and redundancy is easy to implement.

### 2.1.3 SOA Design Methodologies

To maximize the potential benefits, SOA design should be implemented from an architectural rather than an integration-centric approach. Several SOA design methodologies have been published by vendors, academics, and standards organization. Two of the most prominent are the Service-Oriented Modeling and Architecture approach from IBM [34], and the OASIS SOA Reference Model [35]. Generally, they both advocate a similar approach. By analyzing the service-oriented application from

the top-down, identifying the business processes and required process steps, and the systems that can provide the required capabilities, a set of services that align with business processes can be identified.

SOMA proposes an iterative approach, combining top-down analysis (domain decomposition) to discover high-level business processes, goal-service modeling to match services to business goals and sub-goals, and existing asset analysis (bottom-up) to identify existing assets that can be externalized as services, or used to realize service functionality.

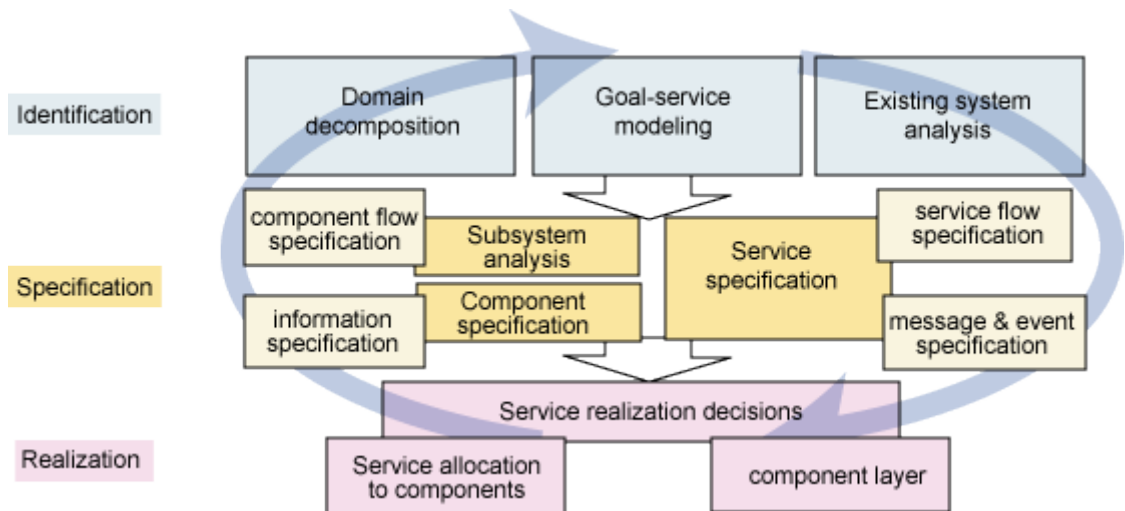


Figure 3: The SOMA Method[34]

The OASIS Methodology advocates a pure top-down approach, starting from high-level business functions, not business processes. It excludes bottom-up analysis entirely, because it leads to an architecture that is technology-oriented, not purely service-oriented. It also discourages the use of high-level business processes in service discovery, because of the focus on ‘how’ (steps required to complete key functions) rather than ‘what’ (the key functions themselves).

Choosing the right methodology depends very much on the application domain. An entirely top-down approach risks identifying services that are difficult to implement with existing systems. An entirely bottom-up approach risks exposing services that are difficult to incorporate into a business application. Identifying services that are too fine-grained can result in impractical overhead, while overly coarse-grained services can be too specific to be reusable [36].

## 2.2 WEB SERVICES

According to the W3C Web Services Architecture Working Group, a web service is defined as “a software system designed to support interoperable machine-to-machine interaction over a network [1].”

Generally, Web Service provider exposes its capabilities as an interface, abstracting away the implementation details. The interface specifies a set of operations, with input and output parameters. An operation invocation involves an exchange of messages between the service requester, and service provider.

The interface, or contract, is described in an XML-based machine-processable format, called Web Services Description Language (WSDL). Machines interact with the Web Service by exchanging SOAP messages, typically using HTTP, in a manner described by the service description.

Web Services technology can be used to implement a service-oriented architecture, where SOAP messages are the basic unit of communication.

### 2.2.1 SOAP

SOAP is an XML-based protocol for exchanging structured, typed information between machines in a distributed environment. SOAP was once an acronym for Simple Object Access Protocol, but this meaning has been dropped in SOAP 1.2. SOAP is a key component in the implementation of Web Services.

Raw SOAP is fairly lightweight compared to other distributed computing standards, because it provides only the messaging framework, and relies on other standards to provide other features, such as registry/discovery, location, transport, security, and guaranteed delivery. SOAP is based on XML, a familiar and widely-used standard, and retains all the extensibility and machine-readability advantages. It is language and platform independent, and does not define any constraints on the transport protocol to be used, so it is possible to pass through corporate firewalls without the need to open ports. An incomplete summary of the relevant components of the standard is provided here.

The SOAP specification defines a stateless, one-way message transmission between SOAP nodes, but it is expected that applications can define more complex Message Exchange Patterns (MEPs) by combining one-way exchanges. A message exchange pattern is defined by providing

- A URI to name the MEP
- Describe the lifecycle of the exchange with a state machine
- Describe the temporal/causal relationships between the messages
- Describe the normal and abnormal termination of the MEP.
- Rules for generating SOAP Faults during MEP operation

The specification provides a framework for conveying application information in an extensible manner, but does not constrain the application-specific message content, or specify anything about underlying routing or transport protocol. A SOAP document consists of three key elements; the envelope, the header, and the body. Figure 4 shows the structure of a SOAP Message.

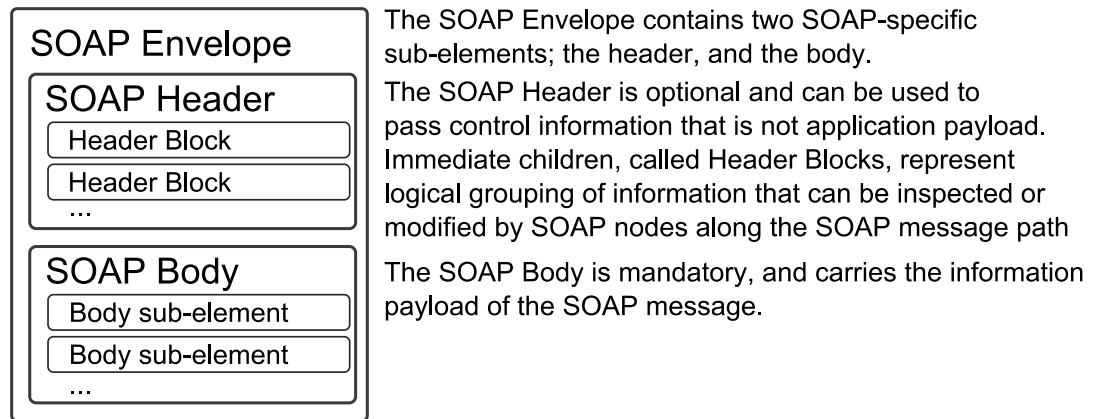


Figure 4: SOAP Envelope [3]

SOAP Header Blocks generally contain the metadata, which are used to control how the message is transmitted. They can contain addressing information, authentication data, tracking information, or security tokens. Introducing header blocks is the primary way in which Web Services standards extend SOAP. The SOAP body is usually application-defined, with the exception of SOAP faults.

When an envelope cannot be interpreted by a SOAP Node, a SOAP fault is generated. SOAP faults are specified to carry error information within a SOAP message. A SOAP Fault contains, at minimum, a fault code and a text explanation of the reason.

Table 1: SOAP fault codes

Fault Code	Description
VersionMismatch	Some element other than the Envelope element was found.
MustUnderstand	A header block targeted at the faulting node with the attribute mustUnderstand with value “true” was not understood
DataEncodingUnknown	A header block or body child element is scoped with an unsupported data encoding.
Sender	The SOAP message was incorrectly formed, or did not contain the appropriate information
Receiver	The message could not be processed for some reason other than the contents of the message itself.

SOAP provides a distributed processing model. Every SOAP message is sent from a SOAP Sender to a SOAP Receiver, and can be relayed through zero or more SOAP Intermediaries. Senders, Receivers, and Intermediaries are all SOAP Nodes, identified by a URI. Each SOAP Node is required to perform some processing according to the SOAP processing model [2].

The SOAP messaging framework is designed to be extensible, using SOAP Features. SOAP Features can be expressed through the SOAP Processing Model, or the SOAP Protocol Binding Framework. These features can describe the behavior of a node with respect to an individual message, or mediate the sending and receiving of SOAP messages on an underlying transport protocol.

A SOAP Protocol binding operates between two adjacent nodes on a SOAP message path. The standard provides the general rules for specifying protocol bindings, and the relationship between bindings and Nodes that implement those bindings. A protocol binding specification augments the core SOAP processing rules with additional protocol-specific processing, and describes how the protocol is used to transmit messages between nodes. The binding must enable at least one message exchange pattern (MEP) and, at minimum, specify how a SOAP message infuset, consisting of an XML document with one soap envelope child element, is encoded, transferred, and reconstituted by the binding at the receiving SOAP node, and specify how the transfer is initiated. The Protocol Binding Framework does not require that XML Serialization be used for transmission; compression and encryption are also appropriate.

An HTTP binding is specified in Part 2 of the SOAP Specification [4]. It does not require a full implementation of all HTTP features, but only the ones necessary to transmit SOAP messages. Two message exchange patterns are supported, request-response, and response, which map to the HTTP methods “POST” and “GET,” respectively.

The HTTP Binding specification also maps HTTP Status Codes to MEP state transitions, and SOAP Faults, where appropriate.

Web Service specifications (WS-\*) from standards organizations such as OASIS and the World Wide Web Consortium (W3C) build on SOAP, defining Header Blocks and MEPs to provide features such as addressing and security.

SOAP-over-HTTP is most widely used, but other protocol bindings have been defined, such as SOAP-over-HTTPS, and SOAP-over-UDP by OASIS [5].

## **2.2.2 Web Services Description Language**

To make effective use of Web Services, clients need an unambiguous, machine-interpretable description of the interface to the Web Service. The Web Services Description Language (WSDL) specification from W3C was created to provide a mechanism for describing the Web Service interface, including:

- All supported operations

- Input and output parameters for each operation
- Types for all parameters, described in XML Schema format
- Binding address information for each Web Service, including location (URL) and transport protocol

The current W3C recommendation is WSDL 2.0, but WSDL 1.1 is still relatively widely used. The structure of the documents is quite similar for both versions, as illustrated in Figure 5.

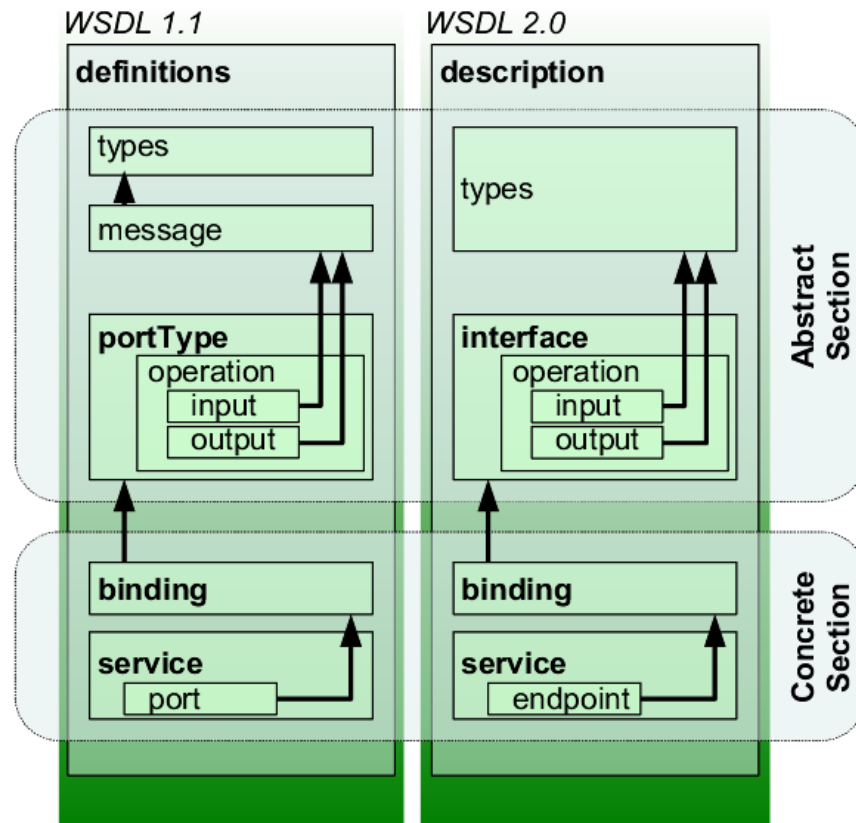


Figure 5: WSDL 1.1 and 2.0 Document Structure[14]

The information described in both is essentially the same [15]:

Table 2: Structure of WSDL 1.1 and WSDL 2.0 files

WSDL 1.0 (WSDL 2.0)	Description
<b>definition (description)</b>	The root element of the WSDL document. It defines the service name, and the namespaces used in the document.
<b>types (types)</b>	The types section describes the input and output parameters of operations in XML Schema format. This is all the type information needed for the information exchange between the service consumer and provider. External XML Schemas (.xsd files) can also be imported.

<b>WSDL 1.0 (WSDL 2.0)</b>	Description
<b>message (N/A)</b>	Messages contain the information required to complete the operation, and contain zero or more message parts, representing parameters. The parts have name attributes, unique within the message, and reference elements in the types section. Messages do not specify direction. In WSDL 2.0, messages are eliminated, and the input and output parameters of the operations simply reference the types directly.
<b>portType (interface)</b>	This defines the Web Service and each operation that the port exposes.
<b>operation (operation)</b>	This defines SOAP actions and the message encoding, as well as the input and output parameters for the operation. If the input is omitted, the operation represents an event. If the output is omitted, no response is expected. If both input and output are specified, the operation is a standard request-response operation.
<b>Binding (binding)</b>	This element specifies the SOAP binding style and transport protocol for the Web Service (portType or interface). The binding style can be either “rpc” or “document.” One binding is specified for each protocol that a Web Service supports.
<b>service (service)</b>	The service element groups related ports together. None of the ports communicate with each other, and if the ports have the same portType and different bindings, then each provides semantically equivalent behavior. A client can choose which port to communicate with based on whatever criteria.
<b>port (endpoint)</b>	This defines the connection point to a Web Service, typically an HTTP URL string.

A WSDL file is a form of service contract. Through its WSDL file, the Web Service is providing an outwardly-descriptive interface, consistent with SOA principles. By locating and parsing a WSDL file, a client has all the syntactic information required to use the operations provided by a Web Service.

## SEMANTIC INFORMATION IN SERVICE CONTRACTS

Still missing, however, is a semantic description of the interface, as well as other non-functional service characteristics, such as Security, Reliability, and Quality of Service(QoS). Traditionally, this is accomplished with a human-readable README file or other type of documentation, which a programmer can use when composing available



services into a business application. Several standards and proposals exist to assist in providing or structuring this additional metadata.

The W3C Recommendation Web Services Policy (WS-Policy) [17] defines a container for specifying a range of policy considerations, but provides no actual semantics for describing policy behavior. These policies can refer to domain-specific capabilities, requirements, and general characteristics of Web Service-based systems.

Web Service Semantics (WSDL-S) is a W3C member submission for annotating WSDL 2.0 documents with semantic information to enable dynamic discovery, composition, and invocation of services [16]. The annotations are defined by a set of WSDL extension elements and attributes, which reference semantic descriptions of operations and their input and output parameters. The Semantic Web and Ontology languages are introduced in a later section. WSDL-S is not tied to any particular ontology representation language.

The submission focuses on the providing annotations for the three abstract constructs defined in WSDL 2.0, namely types, interface, and operation. Service and binding annotations are assumed to be addressed by WS-Policy. Five new extensibility attributes and elements are defined:

*Table 3: WSDL-S extensibility attributes and elements*

<b>Attribute or Element</b>	<b>Description</b>
modelReference	Specifies the association between a WSDL entity and a concept in some semantic model Scope: complex types, element, operation, as well as extension elements precondition, and effect
schemaMapping	Added to XSD elements and complex types to handle structural differences between schema elements and their corresponding semantic concept descriptions
precondition effect	Child elements of the operation element, used mostly in service discovery and composition. The standard does not restrict the semantic description or use of these elements.
category	An extension attribute to the interface element, which can be used for semantic categorization in service registries, such as UDDI.

Each operation, type and message can be linked to a semantic description in a domain ontology description, and the conditions for invoking an operation, as well as the expected result of the operation can also be unambiguously defined.

### 2.2.3 Devices Profile for Web Services

To promote interoperability between resource-constrained devices, the Devices Profile for Web Services (DPWS)[18] defines a minimal set of implementation requirements for dynamic discovery, service description, secure messaging, and events and subscriptions. The goals of the standards are to provide interoperability analogous to Universal Plug and Play (UPnP) for networked devices, fully aligned with Web Services technology. The profile defines constraints on formats and protocols so that Web Services can be implemented on peripheral and consumer electronics-class devices.

The general layout of DPWS clients, devices, and services is as follows:

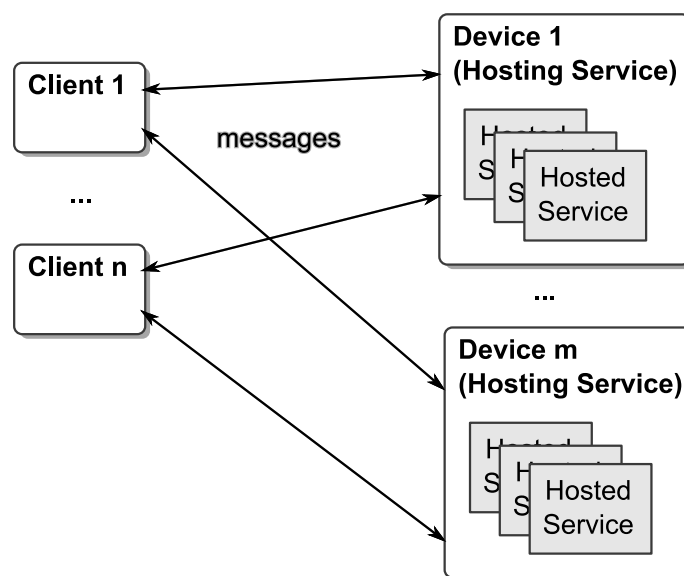


Figure 6: Arrangement of DPWS Clients, Devices, and Services[10]

DPWS defines a service as a software system that exposes its capabilities by receiving and/or sending messages on one of several network endpoints. Messages in DPWS are always transmitted in a SOAP envelope, generally transported via HTTP and TCP/IP, or SOAP-over-UDP in the case of discovery services.

From a SOA perspective, a DPWS-compliant device is a type of service that hosts other services. The device acts primarily as a resource for device-wide metadata, and for metadata about the services it hosts. A hosted service is outwardly visible, not encapsulated by the hosting service, and is addressed separately from the host.

DPWS specifies a set of built-in services that the device must implement:

- Discovery services, for clients to discover devices, and for devices to announce themselves in a network
- Services to retrieve device and hosted service metadata
- Eventing and Subscription Management services, for asynchronous notifications

A client can discover devices in the network that match specified criteria, retrieve and interpret metadata, invoke operations available in the hosted services, and subscribe to and receive notifications.

DPWS assembles a set of core web standards, and extends or constrains them to provide a base set of capabilities for resource-constrained devices:

*Table 4: Specifications in the Devices Profile for Web Services*

<b>SPECIFICATION</b>	<b>DESCRIPTION</b>
WSDL 1.1	The WSDL describes the messages each hosted service is capable of sending and receiving.
SOAP 1.2	All messages are transported in a SOAP envelope, and additional specs make use of SOAP headers.
WS-Addressing [23]	<p>This Specification standardizes endpoint references and message information headers, to convey information typically provided by transport protocols and information systems. A Web Service endpoint is a referenceable entity that can send and receive messages. An endpoint reference can be used to provide information for accessing a Web Service endpoint, or to provide an address for an individual message inside a message information header, along with message characteristics, source and destination addressing, and message identity.</p> <p>DPWS should rely solely on WS-Addressing 1.0, with added restrictions for device identifiers.</p>
WS-Transfer [19]	<p>Specification that defines a mechanism for acquiring XML-based representations of entities using the Web Services infrastructure. It defines two operations for sending and receiving resource representations (Get, Put), and two for creating and deleting (Create, Delete) a resource using a “resource factory”.</p> <p>DPWS uses the WS-Transfer Get operation as a means for the Client to retrieve resource representation data for a device, which includes relationship metadata for itself and hosted services, and addressing data for the hosted services.</p>
WS-MetadataExchange [20]	This specification is intended for the retrieval of Web Service Description Information. It defines an encapsulation format for metadata, and treats the metadata about a web

SPECIFICATION	DESCRIPTION
	<p>service endpoing as a WS-Transfer resource. The specification defines two mechanisms that clients can use to ask a WS-MetadataExchange endpoint for its metadata: GetWSDL/GetWSDLResponse, and GetMetadata/GetMetadataResponse.</p> <p>DPWS uses the GetMetadata operation to retrieve metadata for a hosted service, which includes the WSDL document. The hosted service can return either the WSDL document, or a reference to the document in a GetResponse envelope.</p>
WS-Policy[17]	<p>This specification provides a general framework for specifying a variety of capabilities, requirements, and characteristics of entities in a Web Service-based system. A policy assertion identifies a behavior that is a requirement of a policy subject. A policy alternative is a set of policy assertions. Generally, a policy is used to convey conditions on interaction between two endpoints. A provider exposes a policy to describe the conditions for providing the service. A requester uses policies when deciding whether to use the service.</p> <p>DPWS defines the dpws:Profile policy assertion, indicating that compliance with the profile is required.</p>
WS-PolicyAttachment [21]	<p>This specification provides two generalized mechanism for associating policies with the subjects to which they apply. It also specifies how the mechanisms can be used to associate WS-Policy with WSDL and UDDI descriptions. The global attribute “wsp:PolicyURIs” and child elements “wsp:Policy “ and “wsp:PolicyReference” are defined, so that resources can reference applicable policies.</p> <p>DPWS uses this to attach the dpws:Profile policy assertion to binding in the WSDL file.</p>
WS-Discovery	<p>This specification describes how to announce availability of services, search for services, and locate previously referenced services on a local network using a multicast discovery protocol based on SOAP-over-UDP. Two one-way messages are defined, Hello and Bye, as well as two two-way search messages, Probe and Resolve. Two</p>

SPECIFICATION	DESCRIPTION
	<p>discovery modes are defined.</p> <p>In ad-hoc mode, clients send probes to a multicast group, and target services matching the search criteria send unicast responses back to the client. The specification also defines multicast hello messages that target services send when they join a network.</p> <p>In managed mode, a discovery proxy receive unicast hello messages from target services, and unicast resolve messages from clients.</p> <p>DPWS specifies that devices must be compliant WS-Discovery target services, but hosted services should not. The profile also specifies additional discovery-related behaviors for devices.</p>
WS-Eventing [22]	<p>It is often useful for web services to receive messages when events occur in other services. This specification provides a means to create and delete subscriptions, manage subscription expiry and renewal, and define a preferred delivery mechanism.</p> <p>WS-Eventing provides an extension point called “Delivery Modes,” and defines a default delivery mode called “Push Mode.”</p> <p>DPWS requires full support for WS-Eventing, including Push Mode, where the hosted service pushes Notifications to the Event Sink (the client). It also specifies fault behavior, appropriate for distributed, low-resource devices, and support for event subscription filtering by action.</p>

---

DPWS also recommends a security model based on WS-Security, but support is optional, and other security models are permitted. DPWS also overrides global constants from other specifications to suit devices, such as packet size limits, timeouts, and delays.

The web service specifications and transport protocols that are part of the Devices Profile for Web Services are shown in Figure 7:

SOA application		
WS-Discovery	WS-Eventing	WS-MetadataExchange
WS-Addressing	WS-Security	WS-Policy
SOAP 1.2 WSDL 1.1, XML Schema		
UDP	HTTP 1.1	
	TCP	
IPv4 / IPv6 / IP Multicast		

Figure 7: Network Protocols and Specifications included in the Devices Profile for Web Services[25]

## DPWS IMPLEMENTATIONS

A number of DPWS implementations exist in various languages, for various operating systems.

The Web Service on Devices API (WSDAPI) [18] is a Microsoft implementation of DPWS for several versions of Windows, including Vista, Windows 7, and Windows Server 2008. The WSDAPI omits some parts of the standard, and introduces additional ones. Some of the omissions include:

- Ignoring size restrictions on UDP packets, strings, and URIs
- Devices and services built on WSDAPI do not provide their WSDL in metadata exchange unless extended by the application to provide this information. The WSDAPI Client implementation does not validate WSDL files, or support late binding. By default, WSDL provision is not part of the programming model.
- Discovery proxies are ignored, and additional WS-Discovery functionality is implemented for cross-network discovery [17].

WSDAPI provides generic client and service DPWS stacks, as well as utilities to facilitate application development. The Web Service on Devices Code Generator is used to create both client proxy and service implementation stub code from WSDL service descriptions.

The .NET Micro Framework is designed to simplify development for resource-constrained, embedded devices, by providing developers with a modern programming environment for creating devices and device drivers. The Framework is currently supported for ARM processors, and includes support for common peripherals and interconnects, such as USB as Flash Memory. The Micro Framework also provides a DPWS stack, fully compatible with WSDAPI. The MfSvcUtil tool, similar to WsdCodeGen, is used to create three files from a WSDL file [51]:

- A service contract class, describing data types for requests and responses, and providing code for serializing the data types to XML
- A client proxy, which hides communication with the service under a layer of abstraction
- Stub code for a hosted service, deriving from the `DpwsHostedService` class

European ITEA research project SIRENA [30] had the objective of developing a Service Infrastructure for Real Time Embedded Networked Applications. The outcome of this project was one of the first to apply the SOA paradigm to communication and interoperability at the level of small embedded devices. Two open-source DPWS projects resulted from this project; SOA4D and WS4D.

The SOA4D (Service-Oriented Architecture for Devices) DPWS Core and DPWS4J Core project are other DPWS-compliant Web Services stacks for C and Java, respectively [28]. Plugins for additional WS standards are also available. Schneider Electric maintains this project.

Web Services for Devices (WS4D) is managed by the University of Rostock, University of Dortmund, and MATERNA [58]. The Java Multi-Edition DPWS Stack (JMEDS) is a framework for implementing and running DPWS Services, Devices, and Clients. Multiple versions of the stack support multiple versions of DPWS, and are tested for compatibility with Windows versions. Additional versions are provided, including a Java implementation for Apache Axis, and a C implementation using gSOAP for small devices.

WS4D-uDPWS [27], DPWS for highly resource-constrained devices, was developed to demonstrate the use of DPWS protocols on IPv6 Low Power Wireless Personal Area Networks (6LoWPAN), and is currently supported for two 8-bit platforms. It supports many features of DPWS, with the notable exclusion of WS-Eventing.

The JMEDS client stack automatically discovers devices, and validates WSDL files, making this stack a workable option for late binding, and “DPWS explorer-” type applications. Also, being written in Java, it is cross-platform. Microsoft’s WSDAPI and .NET Micro Framework make developing services and clients for .NET applications relatively simple by hiding all the messaging and implementation detail behind proxy classes, which must be present at compile time. Support for late-binding requires additional programming effort, when compared to JMEDS.

## 2.3 EVOLUTION OF SCADA SYSTEMS

Trends in technology development over time influence the design of Supervisory Control and Data Acquisition (SCADA) Systems. Microcontrollers and CPUs reduce in size, and improve dramatically in processing power, leading to the proliferation of small, intelligent, embedded, networked devices. Field devices can now contain sensing,

control, and decision-making code. Improvements in network infrastructure make transferring data between nodes faster and more reliable.

The trend in SCADA has been toward distributed systems, using open protocols and standard network infrastructure. Three generations have been identified, shown in Figure 8.

1. The first generation SCADA systems featured a monolithic Master Terminal Unit, independent from other systems, communicating via a WAN to dumb RTUs using proprietary protocols, and rarely integrated with other systems. RTUs had little to no autonomy.
2. The second generation saw the emergence of networked devices, connected by LANs, enabling real-time information sharing and distribution of processing. Protocols and data encodings tended to remain proprietary.
3. Today, the third generation favours open communication protocols and architecture standards, and Internet connectivity.

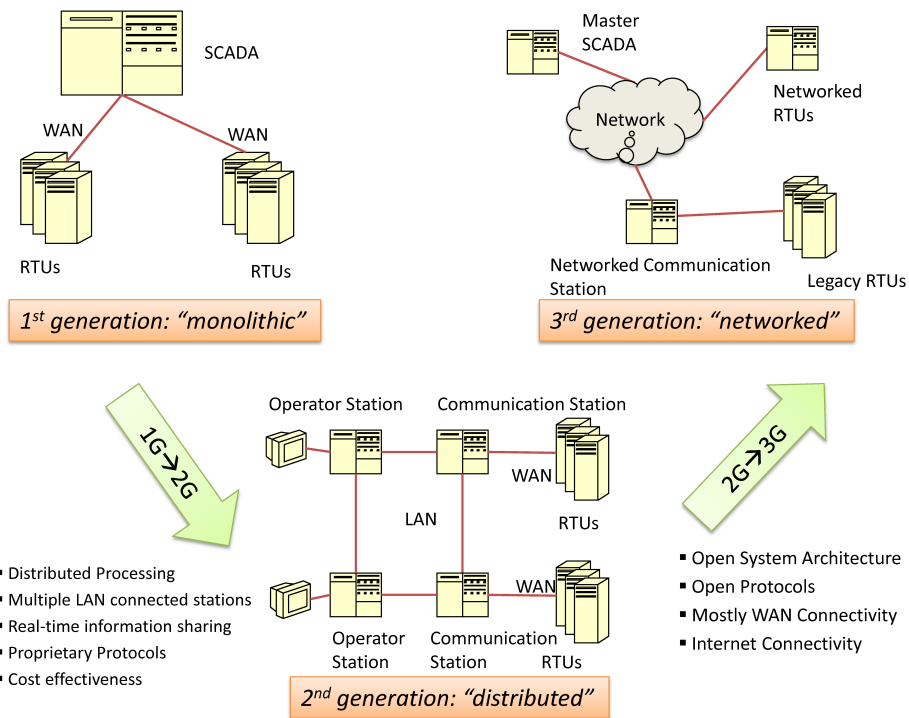


Figure 8: Evolution of SCADA Systems [33]

SCADA systems, traditionally reacting to system faults detected from information gathered from field devices, now behave more proactively, including data processing, security, and prediction based on historical data [62].

Future SCADA systems will likely continue this trend, with globally distributed subsystems becoming more dynamic and collaborative, and real-time information will be shared across all systems at all layers of a manufacturing enterprise. Some



researchers have proposed that SOA and Event-Driven SOA as technologies that can enable this trend to continue.

## 2.4 SOA IN INDUSTRIAL APPLICATIONS

The SOA paradigm was developed as an architecture style for enterprise IT systems, but the guiding principles and best practices are also applicable in industrial domains. A SOA solution for industrial automation and monitoring systems can provide the same advantages over traditional systems that Enterprise SOA provides over other architectural styles.

Future manufacturing systems will be required to be rapidly and cost-effectively integrated, and easily reconfigured in order to cope difficult and dynamic markets [13]:

- Short product lifecycle and quick product introduction
- Demand for mass customization
- Introduction of new processes and machines into the manufacturing workflow
- Scaling production up and down in response to fluctuating demand

The case for applying Service Oriented Architecture (SOA) to industrial control systems has been made in many previous research projects and publications [37]. The benefits that Service Orientation at the device level is expected to bring to industrial applications include:

- **Easier and less costly equipment integration, reusability and reconfigurability:** a substantial portion of the operating cost of a manufacturing plant comes from setup and integration, as well as maintenance downtime. Introducing new equipment requires additional downtime and integration effort. The improved integration capability that SOA promises could minimize costly downtime.
- **Cross-vendor compatibility:** Proprietary standards dominate today's factory floors and enterprise systems. Building systems around open standards reduces a business' reliance on proprietary protocols, systems, and data formats, and can drastically decrease the effort with which systems from multiple vendors can be integrated, allowing factories to choose the best-of-breed components for all systems, without worrying about interoperability.
- **Improved cross-layer integration:** Traditional rigid, hierarchical systems with different communications infrastructure and data formats at all levels are not conducive to full, seamless cross-layer integration. A SOA-style flat architecture could enable Enterprise Resource Planning (ERP) systems to have visibility all the way down to the level of smart embedded devices.
- **Improved business agility:** Service- and event-based systems, coupled with technologies like Complex Event Processing (CEP) can enable calculation of

more complex Key Performance Indicators (KPIs) at the Manufacturing Execution System (MES) level, and more complete information about the factory floor at the ERP level. This allows businesses to adapt more quickly to changing market conditions, and optimize business activities in new ways.

The two most important technologies for creating service-oriented industrial applications are Devices Profile for Web Services (DPWS) and OPC Unified Architecture (OPC-UA). Both solutions are based on Web Services, but follow fundamentally different approaches.

### **2.4.1 Devices Profile for Web Services in Industry**

The ITEA project SIRENA (Service Infrastructure for Real Time Embedded Networked Applications) laid the groundwork for industrial SOA, proving the feasibility of web services at the embedded device level, and produced the SOA4D open-source Devices Profile for Web Services (DPWS) stack, which continues to be maintained. Two projects stemmed from SIRENA, which further developed this idea: ITEA SODA, and FP6 SOCRADES.

ITEA SODA [31] investigated the ecosystem required to implement a SOA system with high-level web service-based communication between devices in several domains, including industry, home, automotive, and telecommunications. They investigated the required development tools, architecture, and methodology for designing, building, deploying, and running a service-oriented application on embedded devices. The SODA project succeeded in embedding web services in very low power, low-cost devices, and promoted the DPWS standard as a platform-neutral integration technology.

FP6 SOCRADES[32] (Service Oriented Cross-Layer Infrastructure For Distributed Smart Embedded Devices) evaluated a variety of SOA solutions applicable at the device level in the manufacturing automation domain, and created a SOA-based infrastructure for manufacturing, where smart embedded devices could interact seamlessly with other service-based components. It also demonstrated how legacy systems could be integrated into a service ecosystem using the gateway or mediator approach, and how manufacturing activities could be automated using service-based orchestration and choreography tools. The project also demonstrated how SOA could enable integration between business-level systems and the factory floor, by providing a flat architecture, applicable to multiple domains.

A DPWS-based solution was demonstrated for electronics assembly. A potential merger between DPWS and OPC Unified Architecture (OPC-UA), another industrial SOA solution, was identified, but not implemented.

SOCRADES did not demonstrate real-time integration of low level devices into high-level applications – the only requirement was the use of Web Services as a communication technology. FP7 IMC-AESOP (ArchitecturE for Service-Oriented Process - Monitoring and Control) is another European Commission-sponsored research

initiative that builds on the results from these previous projects. IMC-AESOP is addressing challenges specific to very large scale distributed systems:

- Distributed monitoring and control of tens of thousands of devices
- Determining what percentage of devices can be service-enabled, considering the performance and real-time considerations
- Managing the large range of plant functionality and dynamic business requirements
- Service lifecycle management for all the autonomous devices operating in a plant
- Defining a transition path for integrating existing devices, manufacturing operations systems, and business systems into the service-based application

The goal of the project is to define a reference SOA architecture for monitoring and control in process industries, and investigate the technology limits for SOA in subsystems, addressing issues such as security, scalability, real-time performance, event-aggregation and filtering, and merging scan-based and event-driven systems.

#### **2.4.2 OPC Unified Architecture**

OPC Unified Architecture (OPC UA) is a relatively new specification from the OPC Foundation for data exchange between systems in industrial applications, based on web-service concepts. OPC UA is designed for accessing large amounts of real-time device data using standard network infrastructure, while maintaining sufficiently high performance. OPC UA specifies a client-server model for information exchange, where a client can access, read, and modify the address space of a server. The specification defines an Object model for information representation on a server, and a pre-defined set of services for browsing, querying, creating, and manipulating Objects in the address space. Information is communicated using OPC UA- and vendor-defined data types, encodings, and transport mappings

OPC UA evolved from classic OPC, which was a data access for Windows-based systems, using Microsoft's OLE, COM, and DCOM technologies. OPC was formerly an acronym for Object Linking and Embedding (OLE) for Process Control, but this acronym has been dropped. The standard was developed to bridge Windows systems and process control devices, and defines standard objects, methods, and interfaces for retrieving field data from devices on the factory floor. A vendor would implement an OPC server for their hardware, which would provide a method for any OPC client to access device data for use in any MES, ERP, HMI/SCADA, or other system.

The OPC UA specification eliminates the reliance on COM/DCOM, and specifies a platform-independent, service-based communication model, and a richer, integrated Address Space Model. In the interest of security and performance, OPC UA defines two data encodings [55]:

- UA Binary: The specification defines a non-portable binary message encoding, optimized for message size, and fast encoding and decoding. The specification relies on a set of primitive data types, for which binary encodings are defined. The encoding excludes type and field name information, because applications are expected to have advance knowledge of the services and data structures being transmitted.
- UA XML: The specification also defines a plain text XML representation of elements in the object model for SOAP/HTTP Web Services. The UA XML encoding uses the formats defined in the W3C XML Schema specification.

Furthermore, two transport mappings are defined [55]:

- UA TCP (UA Native): A TCP-based OPC UA-specific protocol for establishing a full-duplex channel for transmitting binary data between an OPC UA client and server. Unlike HTTP, responses can be returned in any order, and allows responses to be returned on a different socket, if communication failure causes an interruption. OPC TCP is designed to work secure SecureChannel, implemented at a higher layer.
- SOAP/HTTP (Web Service): OPC UA messages are serialized to XML, wrapped in a SOAP envelope, and exchanged using the request-response model defined in the SOAP specification. HTTP or HTTPS transport bindings are used. A message is sent in the body of a POST request, and the response comes in the HTTP response.

The client-server communication paradigm of OPC UA lends itself well to hierarchical application architectures. A higher level client application retrieves data and writes values to a lower-level server. Application layers can be stacked by having an OPC UA Server and Client running on the same component, as shown in Figure 9.

Each component running an OPC UA Server manages its own address space. The server can map nodes in the address space to IOs on devices in a connected fieldbus network or PLC memory, or expose data from a database. A single OPC UA Server integrates data, type definitions, Alarms and Events, and historical data into its Address Space. The Server supports a set of Web Services, which a Client can use to establish a session, browse and query the address space, subscribe to notifications, and invoke object methods.

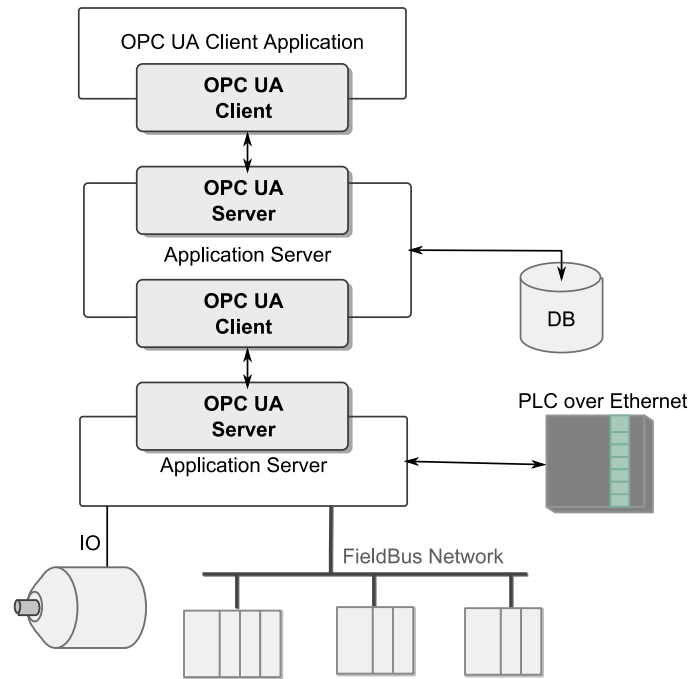


Figure 9: OPC UA Stacked Architecture

## OPC UA DATA MODEL

OPC Unified Architecture is fundamentally about data modelling and transport. In classic OPC, only pure data was provided, such as raw sensor values, with only limited semantic information, such as the tag name and the engineering unit. OPC UA offers more powerful capability for semantic modelling of data.

OPC UA uses object-oriented techniques, including type hierarchies and inheritance, to model information. Type information is stored on Servers and accessed in the same way as instances, similar to relational database systems. The OPC UA Node model allows for information to be connected in various ways, by allowing for hierarchical and non-hierarchical reference type. This facilitates exposing the same information in many ways, depending on the use case. Both the type hierarchies and references types can be extended, allowing information models of existing systems to be exposed natively, without the need for mapping between models. Information models are always contained in an OPC Server, so an OPC Client is not required to have an integrated OPC UA Information model.

The base OPC UA specifications provide the only the infrastructure to model information, and encourage additional, industry specific information model specifications to be defined by vendors and standards organizations. Development has begun on a base model for exposing device information and device types in OPC UA (UA Devices), which can be extended with vendor-specific information. Also, efforts are underway to expose the ISA 95 model in OPC UA to provide information to MES and ERP systems.

### OPC UA Address Space

The objects and related information that an OPC UA Server makes available to a Client comprises its Address Space. The contents of the address space are represented as a set of Nodes, described by Attributes, and interconnected by References.

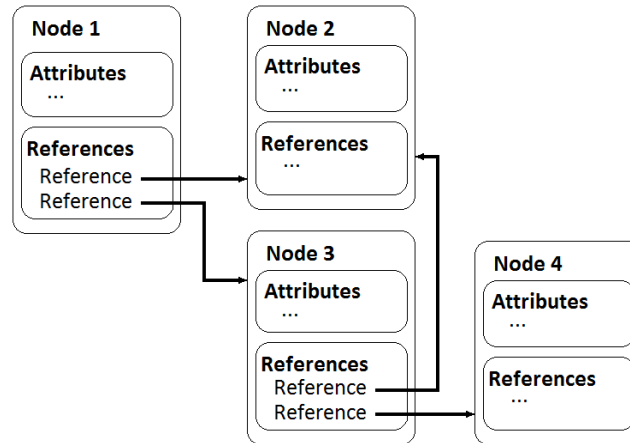


Figure 10: OPC UA Node Model[38]

The AddressSpace is designed hierarchically, and the top levels are the same across all OPC UA Servers to promote interoperability. OPC UA defines a set of Web Services that allow OPC UA Clients to browse and edit objects in the Address Space of an OPC UA Server. The address space model is described in Part 3 of the OPC UA Specification, and summarized here.

Each Node in the AddressSpace is an instance of a NodeClass, which describes the Attributes and References that must be instantiated when a Node is defined in the AddressSpace.

Attributes are the only elements in the OPC UA AddressSpace that have Data values. Attributes are elementary components of nodes, included in NodeClass definitions, and are not themselves represented as nodes in the AddressSpace. Attribute values on a Server can be accessed by a Client using Read, Write, Query, and Subscription/MonitoredItem Services. Attribute definitions consist of the following information

- Attribute id
- Name
- Description
- Data type
- A Mandatory/Optional indicator

NodeClasses define a fixed set of Attributes that cannot be extended by the Client or Server. Additional descriptive information about nodes can be added using Properties.

References describe connections between nodes. They are also elementary components of nodes, not nodes themselves, but differ from Attributes in that they are defined as instances of ReferenceType nodes, described by the ReferenceType NodeClass. References are accessed indirectly using browsing and querying services. The node that contains the Reference is referred to as the SourceNode, and the node being referenced is the TargetNode. All References are typed, and the ReferenceType defines the semantics of the relationship between the Source and Target nodes. The TargetNode can be in the AddressSpace of the same OPC UA Server as the SourceNode, or in the AddressSpace of another OPC UA Server. The specification does not require that the TargetNode exists. References are generally not ordered, but there are ReferenceTypes that define order for References of that type, such as HasOrderedComponent.

All NodeClasses inherit attributes from the BaseNodeClass, which defines the attributes common to all nodes, allowing identification, classification, and naming [39]:

*Table 5: Attributes Common to all OPC UA Nodes*

<b>Attribute</b>	<b>Data Type</b>	
<b>NodeId</b>	NodeId	Uniquely identifies the node in the OPC UA Server, and used to address the node in OPC UA Services
<b>NodeClass</b>	NodeClass	An enumeration identifying the NodeClass of a node
<b>BrowseName</b>	QualifiedName	Identifies the node when browsing the OPC UA Server. Not necessarily unique, and not localized.
<b>DisplayName</b>	LocalizedText	The name that should be used to display the node in a user interface.
<b>Description</b>	LocalizedText	Optional textual description of the node.
<b>WriteMask</b>	UInt32	This optional attribute specifies which attributes of the node are writable by an OPC UA Client.
<b>UserWriteMask</b>	UInt32	This optional attribute specifies which attributes can be modified by a user connected to the server.

NodeClasses make up the metadata of the AddressSpace. Client and Servers may neither extend the NodeClass definitions in the OPC UA Specification, nor define their own.

### OPC UA Object Model

Objects are defined in terms of Properties, Variables and Methods, as well as Events. The OPC UA Object Model is how information is structured and enriched with semantics in the address space, allowing domain-specific relationships between objects to be expressed.

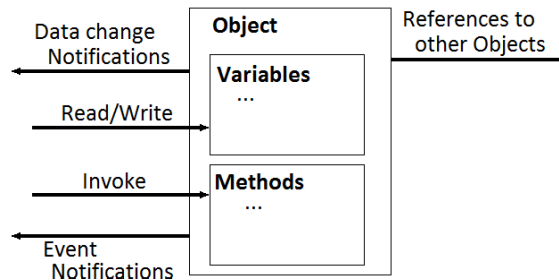


Figure 11: OPC UA Object Model[39]

Objects, Variables, and Methods are represented in the AddressSpace of a server as instances of `ObjectType`, `VariableType`, and `Method NodeClasses`.

### *Variables and Methods*

Variables represent values, and can be either `Properties` or `DataVariables`.

`Properties` are characteristics of `Objects`. `Properties` characterize what a `Node` represents and are server-defined, while `Attributes` define metadata common to all `Nodes` of a `NodeClass`. To prevent recursion, `Properties` may not have sub/`properties` defined for them.

`DataVariables` represent the content of an `Object`. `DataVariables` can have `properties` defined for them, but only complex `DataVariables` can have additional `DataVariables`. A complex `DataVariable` can represent aggregates of other `DataVariables` in the `AddressSpace` by defining a `HasComponent Reference` to each `Node`.

`Methods` are functions, whose scope is bounded by an owning `Object`, similar to static methods of a class. Each method is described by a node of the `Method NodeClass`, which identifies the method's arguments, and describes the behaviour. `Methods` are invoked by `Clients` using the `Method Call` service. `Methods` run to completion, and return the result to the client.

### *Type Definitions*

`OPC Servers` are required to provide type definitions for `Objects` and `Variables`. A `BaseDataVariableType` is defined so a server can use this type if no more specialized type is available. The `HasTypeDefinition` reference links an instance with its type definition, represented by a `TypeDefinitionNode`. `Objects` and `Variables` inherit the `Attributes` described in their `TypeDefinitionNodes`. Industry organizations and standardization groups can define a `TypeDefinitionNode` that is well known in industry, so that `Clients` can interpret it without reading it from the `Server`.

Complex `TypeDefinitionNodes` can define `References` to other `Nodes` as part of the type definition. `TypeDefinitionNodes` reference instances instead of other `TypeDefinitionNodes`, to allow several instances of the same type to have unique `References`, names, and default values. These are called `InstanceDeclarations`. Some instances can be shared, and therefore referenced by `TypeDefinitionNodes`, `InstanceDeclarations`, and instances.



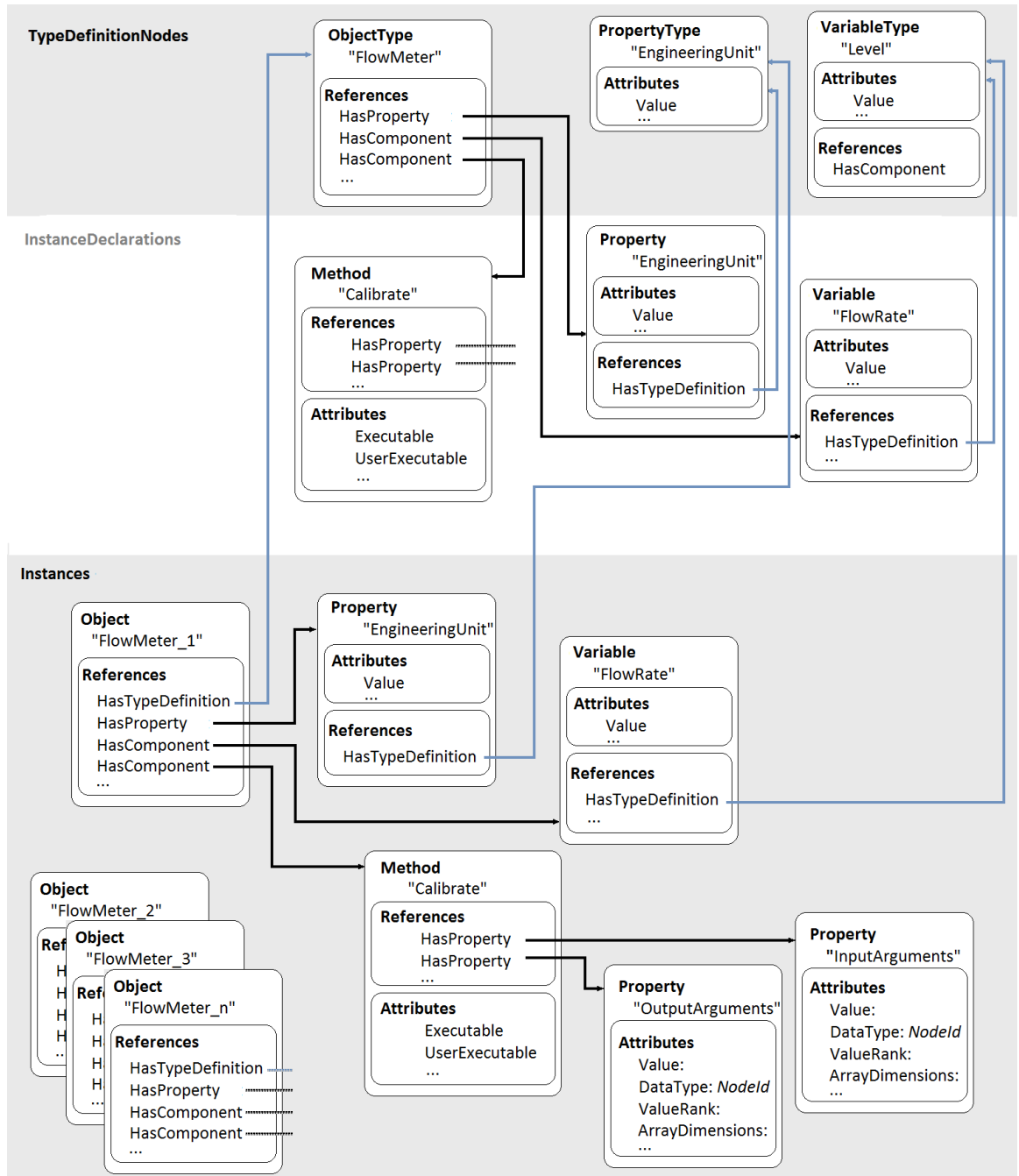


Figure 12: Representation of a Simple Object in the Address Space of an OPC UA Server

Type Definitions can be subtyped to add additional characteristics. This is represented by creating a new Type with the desired characteristics, which is a TargetNode for a HasSubtype Reference from the original VariableType or ObjectType.

Figure 12 shows a possible representation of a simple object, a flow meter with a value variable, a calibration method, and an engineering unit property, on an OPC UA Server.

*OPC UA Information Model*

The OPC-UA Information model describes standardized nodes of a Server’s AddressSpace. The information Model defines the address Space of an empty OPC-UA Server. For compatibility, the top-level nodes of each Server’s address space must look the same.

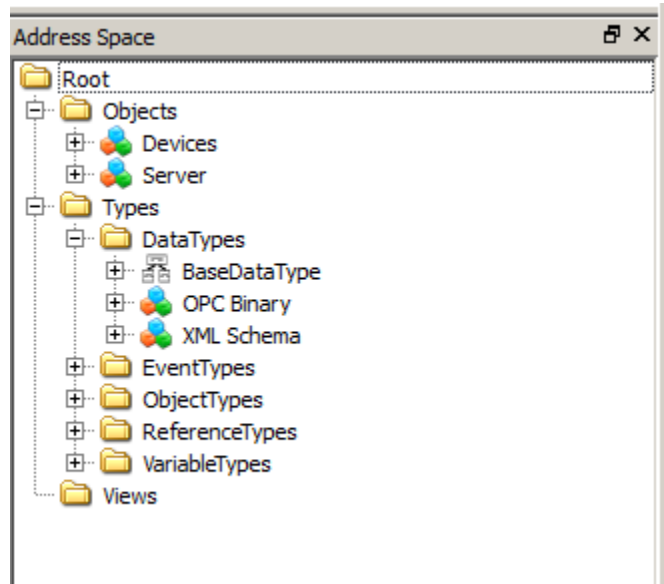


Figure 12: A View of the top-level nodes of an Ignition OPC-UA Server [43], as seen from Unified Automation UaExpert OPC UA Client[42]

The Nodes are standardized types, and as well as standardized instances used for diagnostics or as entry points to server-specific nodes.

**OPC UA SERVICE SETS**

OPC UA defines fixed service sets, which cannot be extended by applications. The service sets allow discovery via a discovery server, secure communication, browsing and querying a server’s address space, and creation of Monitored Items and Subscriptions.

The service sets are described in Table 6:

Table 6: OPC UA Service Sets[44]

Service Set	Service	Description
Discovery		Used to discover <i>Endpoints</i> implemented by a server, and to read the security configuration for the <i>Endpoints</i> . Each <i>Server</i> has a <i>Discovery Endpoint</i> that the <i>Clients</i> can access without establishing a <i>Session</i> .
	FindServers	Returns the <i>Servers</i> known to a <i>Server</i> or <i>Discovery Server</i> . <i>Client</i> can specify filter criteria.  A <i>Server</i> returns only a record that describes itself, while a <i>Gateway Server</i> returns a record

		for each <i>Server</i> that it provides access to, itself included (optionally).
	GetEndpoints	Returns the <i>Endpoints</i> supported by a <i>Server</i> , and all the configuration information required to establish a <i>SecureChannel</i> and <i>Session</i> .
	RegisterServer	A <i>Server</i> registers itself with a <i>Discovery Server</i> . The <i>Server</i> establishes a <i>SecureChannel</i> with the <i>Discovery Server</i> before calling this <i>Service</i> .
SecureChannel	A <i>SecureChannel</i> is a long-running logical connection between a single <i>Client</i> and single <i>Server</i> , which ensures the <i>Confidentiality</i> and <i>Integrity</i> of all <i>Messages</i> exchanged. The channel contains a set of authentication and encryption keys known only to the <i>Client</i> and <i>Server</i> . A <i>SecureChannel</i> is not implemented by the <i>OPC UA Application</i> , but are instead provided by the <i>OPC UA Communication Stack</i> .	
	OpenSecureChannel	Open or renew a <i>SecureChannel</i> .
	Close Secure Channel	Terminate a <i>SecureChannel</i> .
Session	A <i>Session</i> is an application layer connection. <i>Sessions</i> are independent of the underlying communication connection, so they are not immediately terminated if the connection fails. The recovery mechanism depends on the <i>SecureChannel</i> mapping.	
	CreateSession	<i>Client</i> creates a <i>Session</i> with the <i>Server</i> . Returns two values, <i>SessionId</i> and <i>AuthenticationToken</i> , which uniquely identify the <i>Session</i> . A <i>SecureChannel</i> must be opened before a <i>Session</i> is created.
	ActivateSession	<i>Client</i> submits its <i>SoftwareCertificates</i> to the <i>Server</i> for validation, and to identify the user associated with the <i>Session</i> . Must be called before any other <i>Service</i> , or else the <i>Server</i> closes the <i>Session</i> .
	CloseSession	Terminate the <i>Session</i> . Stop accepting requests, return negative responses to all outstanding requests, removes the <i>Client</i> from the <i>SessionDiagnosticArray</i> variable.
	Cancel	<i>Client</i> cancels outstanding service requests.
NodeManagement	Add and delete <i>AddressSpace Nodes</i> and <i>References</i> between them.	
	AddNodes	Add <i>Node(s)</i> to the <i>AddressSpace Heirarchy</i> . Each <i>Node</i> is added as the <i>TargetNode</i> of a <i>HierarchicalReference</i> to ensure that the <i>AddressSpace</i> is fully connected.
	AddReferences	Add <i>Reference(s)</i> to <i>Node(s)</i> .

	DeleteNodees	Delete <i>Node(s)</i> from the <i>AddressSpace</i> .
	DeleteReferences	Delete <i>Reference(s)</i> of a <i>Node</i> . Triggers a <i>ModelChange</i> event.
View	<i>Client</i> uses browse <i>Services</i> to navigate the <i>AddressSpace</i> or <i>View</i> , a subset of the <i>AddressSpace</i> created by a <i>Server</i> .	
	Browse	Discover <i>References</i> of a <i>Node</i> . Can be limited by using a <i>View</i> .
	BrowseNext	Request the next set of <i>Browse</i> or <i>BrowseNext</i> responses that is too large for a single response.
	TranslateBrowse... PathsToNodeIds	Translates browse path(s), consisting of a starting <i>Node</i> and <i>RelativePath</i> , to <i>NodeId(s)</i> .
	RegisterNodes	<i>Clients</i> register <i>Nodes</i> they will access repeatedly, to be used for any potential <i>Server</i> -side, vendor-specific optimization. Useful if the <i>Server</i> doesn't have direct access to the information that it manages.
	UnregisterNodes	Unregister <i>Nodes</i> to free up resources.
Query	Used to <i>Query</i> a <i>Server</i> , used to access a wide variety of <i>OPC UA</i> Data stores and information management systems. A <i>Query</i> permits a <i>Client</i> to access data maintained by a <i>Server</i> , without knowledge of the logical schema used for internal storage. <i>Clients</i> can also <i>Query Views</i> (subsets of the <i>AddressSpace</i> ), and historical data, by specifying a <i>ViewVersion</i> or <i>TimeStamp</i> .	
	QueryFirst	Issue a <i>Query</i> to a <i>Server</i> . Request Data from instances of a <i>TypeDefinitionNode</i> , or request data from instances of related <i>Node</i> types, by specifying a <i>RelativePath</i> .
	QueryNext	Request the next set of <i>QueryFirst</i> or <i>QueryNext</i> response information, if it is too large for a single response.
Attribute	Provides access to <i>Attributes</i> , that are part of <i>Nodes</i> .	
	Read	Read <i>Attribute(s)</i> from <i>Node(s)</i> . Attributes with indexed elements (e.g. arrays) can be read as a composite of a range of indexed values.
	HistoryRead	Read Historical values or <i>Events</i> of <i>Node(s)</i> . Historical values are not visible in the <i>AddressSpace</i> , but can be accessed with this <i>Service</i> .
	Write	Write <i>Attribute(s)</i> to <i>Node(s)</i> .
	HistoryUpdate	Insert, Replace, or Delete historical values or <i>Events</i> of <i>Node(s)</i> .

Method	Represents Function calls of <i>Objects</i> . Methods are invoked and return only after completion.	
	Call	Invoke a list of <i>Methods</i> within the context of an existing <i>Session</i> . This <i>Service</i> provides for passing input and output arguments to/from a <i>Method</i> , defined by a <i>Method's Properties</i> .
MonitoredItem	<i>Clients</i> define <i>MonitoredItems</i> to subscribe to data and <i>Events</i> . <i>MonitoredItems</i> identify the <i>Node Attribute</i> to be monitored, and the <i>Subscription</i> to use to send <i>Notifications</i> . <i>Notifications</i> are data structure that describes the occurrence of data changes and <i>Events</i> .	
	CreateMonitoredItems	Create and add <i>MonitoredItem(s)</i> to a <i>Subscription</i> .
	ModifyMonitoredItems	Modify <i>MonitoredItems(s)</i> of a <i>Subscription</i> .
	SetMonitoringMode	Set the monitoring mode for <i>MonitoredItem(s)</i> of a <i>Subscription</i> . {DISABLED; SAMPLING; REPORTING}
	SetTriggering	Create and delete triggering links for a triggering item. Triggering links are represented by the <i>MonitoredItem</i> id for the item to report, and link triggering items to items.
	DeleteMonitoredItem	Remove <i>MonitoredItem(s)</i> from a <i>Subscription</i> . Also removes its triggered item links.
Subscription	Subscriptions are used to report <i>Notifications</i> to <i>Clients</i> . They have a set of <i>MonitoredItems</i> , assigned by the <i>Client</i> , which attempt to send <i>NotificationMessages</i> , containing <i>Notifications</i> , to the <i>Client</i> at the specified publishing interval. The <i>Subscription</i> periodically sends <i>NotificationMessages</i> at user-specified publishing intervals (interval of 0, if event-based).	
	CreateSubscription	Creates a <i>Subscription</i> , which monitors a set of <i>MonitoredItems</i> for <i>Notifications</i> , which are returned to the <i>Client</i> in response to <i>Publish</i> requests.
	ModifySubscription	Modify a <i>Subscription</i> .
	SetPublishingMode	Enable sending <i>Notification(s)</i> on <i>Subscription(s)</i> .
	Publish	Acknowledge receipt of <i>NotificationMessages</i> for <i>Subscription(s)</i> , or request the <i>Server</i> to return a <i>NotificationMessage</i> or keep-alive <i>Message</i> . <i>Publish</i> requests can be used by any <i>Subscription</i> .

	Republish	Request the <i>Subscription</i> to republish a <i>NotificationMessage</i> from its retransmission queue.
	TransferSubscriptions	Transfer <i>Subscription</i> and its <i>MonitoredItems</i> between two <i>Sessions</i> of a single <i>Client</i> , or from one <i>Client's Session</i> to another <i>Client's Session</i> .
	DeleteSubscriptions	<i>Client</i> deletes one or more <i>Subscriptions</i> that have not been transferred to another <i>Client</i> , or that have been transferred to it.

OPC UA defines two data encodings

- XML/text
- UA Binary

And two transport protocols:

- TCP
- SOAP Web Services over HTTP

The XML Web Services Stack is shown in Figure 13 below:

WS-SecureConversation			WS-SecurityPolicy 1.2
WS-Security 1.1		WS-Trust 1.3	
XML Signature 1.0	XML Encryption 1.0	WS-Addressing 1.0	
SOAP 1.2			
HTTP or HTTPS (SSL/TLS)			

Figure 13: OPC UA Web Services Stack

### 2.4.3 OPC UA Companion Specifications

The OPC UA Data Model is designed to be extended with object and information models from other standards organizations, as OPC UA Companion Specifications. The following standard organizations have been identified as potential candidates for companion specifications to describe how their data is exposed in OPC UA, and some working groups have been formed to

- EDDL, in cooperation with Foundation Fieldbus, Hart, Profibus
- Field Device Integration (FDI)
- ISA 88/95
- MIMOSA
- IEC TC57 WG13

UA Companion Specifications already exist for Devices and PLCopen (IEC 61131-3). The OPC UA for Devices specification defines an information model, providing a

unified view of devices, irrespective of the underlying device protocols. It specifies ObjectTypes used to represent devices and components in the OPC UA Address Space, mainly for device configuration and diagnostics. The standard is general enough to allow any application to access device data. The ObjectTypes defined include:

Table 7: ObjectTypes defined in the OPC UA for Devices Companion Specification

ObjectType	Description
TopologyElementType	Base element in a device topology model, specifying parameters and methods
DeviceType	Supports sub-devices and Blocks
BlockType	Used to organize an address space. Block models can be specified by Fieldbus Organizations
ProtocolType	Represents a specific communication protocol implemented by a TopologyElement, such as PROFIBUS, FFBusType, etc.
ConfigurableObjectType	Used to create modular topology units, used by devices to organize blocks.

The hierarchy of these objects is shown in Figure 14. A device would be represented as an object of type DeviceType, which inherits the properties and attributes of TopologyElementType.

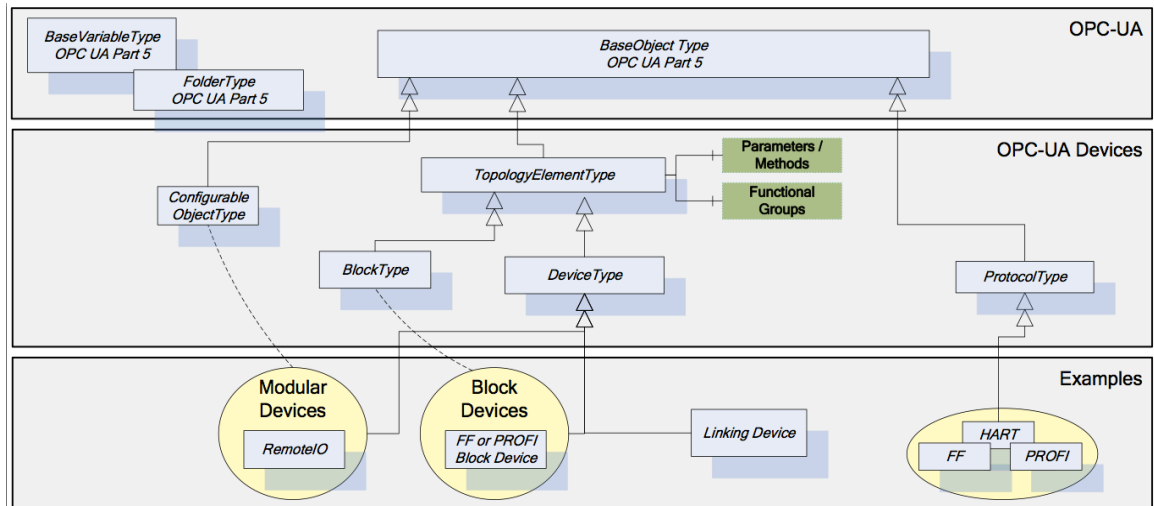


Figure 14: OPC UA Object Types Introduced by OPC UA for Devices Companion Specification [49]

Functional groups can be used to organize parameters and methods inside a Topology Element. A single Parameter or Method can be referenced from multiple

functional groups. Functional Groups can represent interfaces such as Configuration and Process Data, as shown in Figure 15.

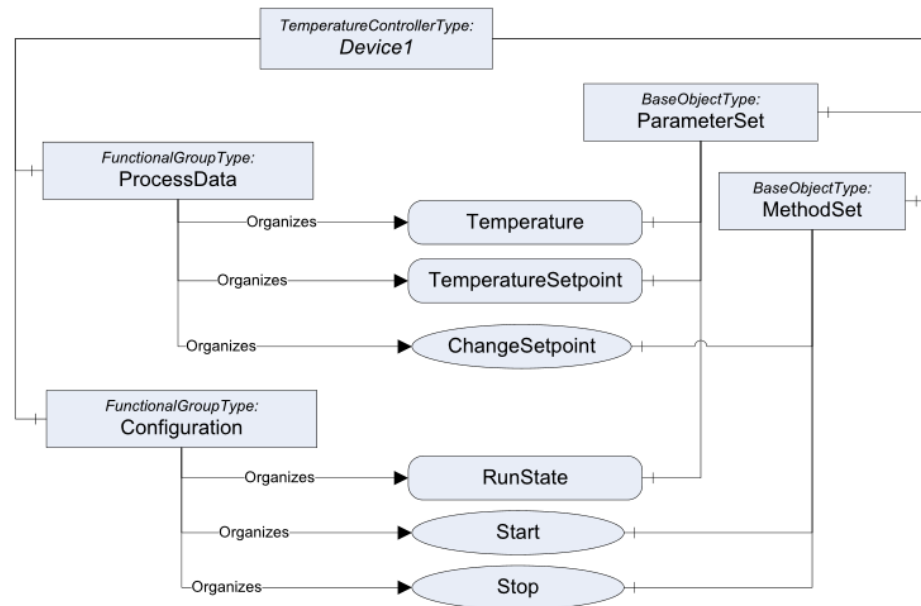


Figure 15: OPC UA Devices Example [48]

OPC UA For Devices 1.00 Companion Specification contains complete descriptions of all blocks, as well as examples.

IEC 61131-3 part of a family of standards, which attempts to standardize programming languages for industrial automation. The PLCOpen OPC UA Information Model extends the OPC UA for Devices models to represent IEC61131-3 elements and programming languages. This specification, create by a joint committee of the OPC Foundation and PLCOpen, defines an information model to represent IEC61131-3 architectural models in the OPC UA Address Space.

The specification document contains examples of representations of Ctrl Configurations, Programs, Function Blocks, and other ObjectTypes. As in the Devices data model, FunctionalGroups can be used to expose groups of parameters and variables as ‘interfaces’ for different use cases.

## 2.5 EVENT-DRIVEN ARCHITECTURE AND EVENT-DRIVEN SOA

The service model described in most SOA literature generally prescribes a synchronous request-response interaction. A device or system exposes its capabilities as services, and another device or system makes use of the capabilities, invoking the exposed operations via SOAP message. The services and operations are discoverable using dynamic discovery mechanisms, or in a service registry. This model creates a degree of dependency between the client and the service, generally not loosely-coupled in practice. Furthermore, following a request-response message exchange pattern results



in information about system state being pulled on request from lower-level systems by higher-level systems.

This model is well suited to composing capabilities of devices and systems into higher-level business processes, but in a manufacturing environment, it is not always optimal. For a large-scale factory or process plant monitoring system, large numbers of intelligent, low-resource embedded devices can be deployed on the shop floor. To achieve a maximally responsive system, one that senses the environment and reacts to changes, notifications of events must be generated immediately. When an alarm is triggered or changes are detected in some monitored value, such as a tank level or flow rate, notification messages can be “pushed” to event subscribers [52]. The result is an extremely loosely-coupled system, where the event source detects an event, publishes a notification, and has no knowledge of the subsequent processing.

In a service-oriented system, service invocation is generally driven in one of two ways[53]. In a composite application, user interaction triggers invocation of one service, or a sequence of services. Alternatively, business processes and events drive service invocations. A service may generate an event, which may indicate a problem, a potential problem, some deviation from normal operation, or a completed milestone. Events are immediately disseminated to all interested systems, which evaluate the event, and, as needed, trigger execution of a business process, invoke a service, or generate another event. An event-generating service or business process can be just one of many event sources in an Event-Driven Architecture (EDA).

David Luckham defines an EDA as “an SOA in which all communication is by events, and all services are reactive event processes (ie. React to input events, and produce output events).” [54]

An event is a notable thing that happens inside our outside a business or system [53]. Each event will typically be specified in business terms, rather than raw data or in application terms, for the event to retain some meaning for interested parties. Event notifications will usually be delivered with a header, and body. The header will typically contain information such as event name, type, timestamp, occurrence number, event source. It may also contain a reference to some semantic information, such as an ontology file containing an event description. The body will contain the actual event data, describing what occurred. In the case of a threshold limit being exceeded, the body may contain the threshold limit level, the measured value, and perhaps the previous value and some severity indicator.

OPC UA and DPWS both support event generation, although the models are quite different. DPWS requires full support for WS-Eventing, including push mode, where the hosted service pushes notifications to the event sink (the client)[22]. This specification provides a means to create and delete subscriptions, manage subscription expiry and renewal, and define a preferred delivery mechanism. OPC UA’s event model defines a general-purpose eventing system, in which a client creates a Subscription object on a server, and collection of MonitoredItems, which use the Subscription to publish EventNotifications. Each Monitored item identifies the item to

monitor, such as Variables, Attributes, EventNotifiers, and generates a notification when they detect a data or status change that match a client-specified filter. A more detailed description of the OPC UA eventing mechanism can be found in Appendix A.

A SOA-based system for shop floor monitoring requires devices capable of acquiring data from sensors and controlling actuators. In many scenarios, such as some pressure or temperature level exceeding a configured threshold, or an access violation in a restricted area, a single sensor value can indicate a fault or emergency situation. However, reading from a single sensor is often not sufficient for detecting an emergency or maintenance situation. Systems should be able to react to situations identified by a sequence or combination of low-level events from one or more sources.

Several styles of event processing can be found in event-driven systems today [53]. A single event notification can communicate valuable information about a specific part of the system at a specific time, but detecting patterns and correlations in events in all sub-systems over an extended time period can be a powerful tool for performance prediction, fault detection and prediction, and complex system-wide situations which can enable companies to make better decisions about future control instructions, maintenance, or production scheduling [50]. Event processing tools can be used to gain greater insight, and perform advanced analytics using simple atomic events generated by individual components in a large system. In the simplest case, Simple Event Processing, an event occurrence initiates some downstream action, driving real-time application flow. Slightly more complex, Stream Event Processing involves analysing a continuous, high-speed stream of time-ordered events, scanning ordinary events for notability, and applying algorithms to the data. Stream event processing facilitates real-time decision-making. Complex Event Processing (CEP) is a more powerful technology for performing analysis on multiple event streams, which has been the subject of much investigation by industrial SOA researchers.

### **2.5.1 Complex Event Processing**

CEP has gained popularity in the domains of network monitoring and Business Process Management (BPM) in recent years [54]. CEP provides tools for handling and analysing events in temporal, combinational, and sequential occurrences, using platform-dependent query language, such as Event Patterning Language (EPL), or Language Integrated Query (LINQ) [50]. The ultimate goal of CEP is to extract high-level knowledge from a cloud of low-level events.

CEP feature sets vary widely, depending on query language and platform. However, capabilities generally include event correlation, composition and aggregation, extraction, parsing, filtering and ordering, semantic matching, structure transformation, and content-based routing. CEP engines can also split, generate, and enrich events, and trigger actions.

Modern processing engines running on adequate hardware can handle on the order of 100,000 events per second, depending on query complexity, but in a very large-scale

system, low-level event sources (devices) themselves should possess some filtering and decision-making capabilities.

Both commercial and open source solutions are available. Microsoft's StreamInsight, CORAL8, and TIBCO Business events are commercial CEP offerings, while Esper (NEsper) and StreamCruncher are free, open-source alternatives.

## 2.6 OWL WEB ONTOLOGY LANGUAGE

The Semantic Web is a concept where human-readable web-content is extended by machine-readable information with explicit meaning, allowing machines to automatically interpret, process, and integrate information on the Web, and intelligently perform tasks. The OWL Web Ontology Language is the top layer in a stack of W3C recommendations related to the Semantic Web[40].

*Table 8: Stack of W3C recommendations related to the Semantic Web*

OWL	OWL adds another layer for richer descriptions of classes, properties, and relationships between classes and properties.
RDF Schema	RDF Schema defines a vocabulary for describing properties and classes of RDF Resources, as well as semantics for generalization hierarchies of these classes and properties.
RDF	Resource Description Framework is a data model for describing objects (“resources”) and relationships between objects. The simple semantics can be represented in XML.
XML Schema	XML Schema extends XML with datatypes, and restricts the structure of XML Documents.
XML	eXtensible Markup Language provides the base syntax for structured information, but does not constrain semantics.

A rich descriptive language for knowledge representation is required for machines to perform useful interpretive tasks on information in documents. The Semantic Web requires structured ontologies.

An ontology defines the terms in some vocabulary, and the relationships between these terms. The ontology represents the area of knowledge (the “domain”), and consists of descriptions of three kinds of concepts:

- Classes (general things) in the domain
- Relationships between things
- Properties or attributes of these things

Ontology languages allow users to write formal descriptions of domain models. Some key requirements of an Ontology Language are:

1. A well-defined syntax

The well-defined syntax is necessary for machine-processing of the information. Although human users will likely be developing their domain ontologies using graphical tools, the basic philosophy of the language should be natural and easily understandable.

## 2. Well-defined semantics

Formal semantics precisely define the meaning of the knowledge, leaving no room for subjective interpretations by different persons or machines. It also allows human reasoning based on the knowledge, and machine reasoning support

## 3. Sufficient expressive power

An Ontology language needs to be more expressive than RDF Schema, supporting ideas such as cardinality restrictions, symmetrical and asymmetrical relationships, disjointness of classes, and defining new classes based on unions, intersections, and complements of classes. A language needs to be able to express wide variety of information, but also allow for reasoning within the information. The expressivity of a language defines what can be represented, and thus determines the reasoning capabilities should be expected from a system that implements it.

## 4. Efficient reasoning support

As the expressive power of the language increases, the reasoning efficiency decreases. Reasoning for ontological knowledge can be about class membership, class equivalence, knowledge consistency, and classification. Machine reasoning support is indispensable when designing large, shared ontologies with multiple authors, or integrating and sharing ontologies between organizations. Machines can rapidly and automatically

- Check for ontology knowledge consistency
- detect unintended relationships
- Automatically classify instances

The OWL Web Ontology Language is designed to meet these requirements. It can be used to explicitly describe the terms and relationships in a domain, for applications where content of information in documents needs to be processed by applications, rather than just presented to a human user. OWL goes beyond XML, Resource Description Framework (RDF), and RDF Schema (RDF-S), to allow greater machine interpretability of web content.

W3C has defined three sub-languages of OWL:

*Table 9: OWL Sub-Languages [40]*

<b>Sub-Language</b>	<b>Description</b>
OWL Lite	OWL supports classification hierarchy and simple constraints. It is designed for easy implementation, and easier understanding. Some concepts, such as disjointness, arbitrary cardinality, and enumerated classes, cannot be expressed in OWL Lite.

**OWL DL** DL is short for Description Logic, which forms the formal foundation for OWL. OWL DL includes all OWL language constructs, but with some restrictions to ensure that the language corresponds to a well-studied description logic. This allows maximum expressiveness, while retaining computational completeness (meaning all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time.) Every OWL DL document is a valid RDF document, but the converse is not necessarily true.

**OWL Full** OWL Full is for applications where maximum expressiveness and syntactic freedom are required, without computational guarantees. It uses all OWL language primitives, with arbitrary combinations of these primitives, including the possibility of changing the meaning of pre-defined vocabulary of RDF and OWL. OWL retains full RDF compatibility.

---

OWL Full can be viewed as an extension of RDF, while OWL Lite and OWL DL are extensions of restricted views of RDF. All OWL sub-languages use RDF syntax, and all instances are declared as in RDF, using RDF descriptions and typing information. All OWL constructors are specializations of RDF counterparts.

### **OWL Language Primitives**

The OWL Language constructs are listed below [41]:

Table 10: OWL Language Constructs

OWL-Lite, DL, and Full			
RDF Schema Features: Class (Thing, Nothing) rdfs:subClassOf rdf:Property rdfs:subPropertyOf rdfs:domain rdfs:range Individual	Property Characteristics: ObjectProperty DatatypeProperty inverseOf TransitiveProperty SymmetricProperty FunctionalProperty InverseFunctionalProperty		Annotation Properties: rdfs:label rdfs:comment rdfs:seeAlso rdfs:isDefinedBy AnnotationProperty OntologyProperty
Datatypes: xsd datatypes	Property Restrictions: Restriction onProperty allValuesFrom someValuesFrom		Versioning: versionInfo priorVersion backwardCompatibleWith incompatibleWith DeprecatedClass DeprecatedProperty
Header Information: Ontology imports			
Boolean Combinations of Class Expressions intersectionOf unionOf complementOf	Cardinality: minCardinality maxCardinality cardinality  only 0 or 1	un-restricted	(In)Equality: sameAs differentFrom AllDifferent distinctMembers equivalentClass equivalentProperty
	Class Axioms: oneOf, dataRange disjointWith equivalentClass (applied to rdfs:subClassOf class expressions)		Filler Information: hasValue
<b>OWL DL and Full Only</b>			

**Structure of an OWL Ontology**

To those familiar with XML syntax, and Object Oriented Programming concepts, the structure and philosophy of an OWL document are easy to understand.

An OWL Ontology uses RDF's XML Syntax. The root of an OWL document is an `rdf:RDF` element, which will contain a number of namespace declarations.

```
<rdf:RDF
  xmlns      ="http://www.tut.fi/fast/2011/aesop-owl-overview#"
  xmlns:ns   ="http://www.tut.fi/fast/2011/aesop-owl-overview#"
  xmlns:base  ="http://www.tut.fi/fast/2011/aesop-owl-overview#"
  xmlns:owl   ="http://www.w3.org/2002/07/owl#"
  xmlns:rdf   ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs  ="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd   ="http://www.w3.org/2001/XMLSchema#">
```

*Sample Code 1: OWL Document Root Element*

Generally, the first two namespaces will be the default namespace, to which all unprefix qualified names will belong, and the second associates a prefix. The third identifies the base URI for the document. The last four define the prefixes needed for the items from the OWL, RDF, RDF Schema, and XML Schema namespaces.

The Ontology will then start with an `owl:Ontology` element, which contains annotations, version information, and imports of other OWL documents. The only information in the `owl:Ontology` element with any logical effect are the `owl:imports` elements, because they indicate other documents that are assumed to be part of the ontology. Typically, there an `owl:import` element for each namespace used.

```
<owl:Ontology rdf:about="">
  <rdfs:comment>Sample OWL ontology</rdfs:comment>
  <owl:priorVersion rdf:resource="http://www.tut.fi/fast/2005/aesop-owl-overview#" />
  <owl:imports rdf:resource="http://www.tut.fi/fast/2011/external-document" />
  <rdfs:label>Sample Ontology</rdfs:label>
  ...
```

*Sample Code 2: owl:ontology Element*

Next, come the definitions that make up the Ontology: Classes and Individuals, Properties, and Property Characteristics and Restrictions.

Classes (objects, things) are defined using `owl:Class` elements, containing Class Axioms, defining relationships to other classes. The `rdfs:subClassOf` relation relates a specific class to a more general one.

```
<owl:Class rdf:ID="temperature">
  <rdfs:subClassOf rdf:resource="#PhysicalProperties" />
  <rdfs:label xml:lang="en">temperature</rdfs:label>
  <rdfs:label xml:lang="fr">température</rdfs:label>
  ...
</owl:Class>

<owl:Class rdf:ID="employee">
  <rdfs:subClassOf rdf:resource="#person" />
  ...
</owl:Class>
```

*Sample Code 3: RDF Subclassing*

Classes can also be declared as boolean combinations of other classes and restrictions on classes.

```

<owl:Class rdf:ID="Actuator">
  <rdfs:comment>
    For simplicity sake, actuators are disjoint with sensors
  </rdfs:comment>
  <owl:disjointWith="#Sensor"/>
</owl:Class>

<owl:Class rdf:ID="Sensor">
  <rdfs:comment>Sensorare devices that measure</rdfs:comment>
  <rdfs:subClassOf rdf:type="#Devices"/>
  <rdfs:subClassOf>
  <owl:Restriction>
  <owl:onProperty rdf:resource="#measures"/>
  <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
    1
  </owl:cardinality >
  </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

```

*Sample Code 4: Boolean Combination*

Describing instances, or individuals, is done by declaring it as a member of a class in one of two ways:

```

<Sensor rdf:ID="thermometer" />
or
<owl:Thing rdf:ID="thermometer" />
<owl:Thing rdf:about="#thermometer">
  <rdf:type rdf:resource="#Sensor"/>
</owl:Thing>

```

*Sample Code 5: Instance declaration*

Properties can be either ObjectProperties, relating Objects to other Objects, or DatatypeProperties, which relate Objects to data values. OWL uses XML Schema data types, and does not specify any restrictions on these types. ObjectProperties can restrict the relation they represent by, for example, specifying domain and range. These values can also be inherited from inverse properties by interchanging domain with range. When multiple domains are declared, the domain is taken as the intersection of all declared domains. The same applies to ranges.

```

<owl:ObjectProperty rdf:ID="measuredBy">
  <rdfs:domain rdf:resource="#PhysicalProperties"/>
  <rdfs:range rdf:resource="#Sensor"/>
  <owl:inverseOf rdf:resource="#measures"/>
</owl:ObjectProperty>

```

*Sample Code 6: Domain and Range*

OWL instances are declared with their properties



```

<owl:Class rdf:ID="SR-20">
  <rdfs:comment>
    Kytola SR-20 Oval Gear FlowMeters have max flow rate of 20 litres per
    minute
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#FlowMeter"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#maxFlowRate"/>
      <owl:hasValue>
        <xsd:double rdf:value="20"/>
      </owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#manufactured-by"/>
      <owl:hasValue>
        <xsd:string rdf:value="Kytola"/>
      </owl:hasValue>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

```

*Sample Code 7: Instance declaration with properties*

Properties can also be extended using the `rdfs:subPropertyOf` relation:

```

<owl:ObjectProperty rdf:ID="hasMaxOperatingTemperature">
  <rdfs:subPropertyOf rdf:resource="#hasPerformanceParameter" />
  <rdfs:range rdf:resource="#Sensor" />
  ...
</owl:ObjectProperty>

```

*Sample Code 8: Extending properties*

DatatypeProperties contain data in any of the XML Schema data types:

```

<owl:DatatypeProperty rdf:ID="manufactured-by">
  <rdfs:domain rdf:resource="#product"/>
  <rdfs:range rdf:resource="xsd:string"/>
</owl:DatatypeProperty>

```

*Sample Code 9: DatatypeProperty*

A more complete treatment can be found in the OWL Web Ontology Language Specifications online from the World Wide Web Consortium.

There are many ways to construct an ontology. Choosing the best approach for a single application depends on the view that the ontology designer takes of the system.

### OWL and the OPC-UA Address Space

The OWL Web Ontology Language and the OPC-UA AddressSpace have some fundamental similarities. Objects, Variables, and References in the OPC UA Object Model are conceptually similar to Class instances, ObjectProperties, and DatatypeProperties in OWL. The OPC UA Specification states that the object model

was intended to extend classic OPC's Data Access specification with more semantic capability. Some further analysis can be done to determine which OWL constructs are expressible in the OPC UA Address Space, and whether any use cases exist that would make such a mapping worthwhile.

## **3 APPROACH AND METHODOLOGY**

### **3.1 MERGING OPC-UA AND DPWS**

#### **3.1.1 Comparison Between the Technologies**

OPC UA and DPWS have both been proposed as solutions for industrial SOA. Although there is considerable overlap between the core Web Service standards that comprise DPWS and OPC UA, the philosophies behind these specifications are fundamentally different. DPWS devices have a set of built in services for discovering and providing interface descriptions for hosted services, which expose the functionality of the device as custom services. OPC UA servers allow OPC UA clients to access and edit nodes in their AddressSpace using a fixed set of services. DPWS is “action,” or “verb”-oriented, while OPC UA is more “object,” or “noun” –oriented, with optional support for methods associated with objects.

Although communication in OPC UA is based on services, a common criticism of OPC UA is that it is not compliant with SOA principles [46]:

- It makes only restricted use of services; fixed service sets are specified, which the user cannot extend
- It is based on a traditional object-oriented data model and does not inherently allow systems or resources to be represented in terms of their capabilities
- It requires a connection to be established, and depends on stateful message exchange patterns.

DPWS, on the other hand, has none of these restrictions. Services can be discovered, operations invoked, and subscriptions established by posting the appropriate SOAP message to the endpoint.

DPWS devices support asynchronous eventing, delivering custom event messages to subscribers using “push mode,” defined in WS-Eventing. Events are described in the WSDL file of the hosted service on the device. A client sends a subscription request with the event name and duration, and thereafter, notifications are delivered asynchronously, with no response required. DPWS does not include a mechanism for enabling clients to specify conditions and custom notifications. OPC UA has a more complex eventing mechanism, allowing clients to create “MonitoredItems” on a server to sample Node attribute values in the address space and server-created “EventNotifiers,” and generate Notifications via Subscriptions when the client-specified conditions are met. Notifications are delivered as NotificationMessages if and when a publish request has been received from the client. Event notifications delivery still

requires that a session be established. A more detailed description of the OPC UA eventing mechanism is presented in Appendix A.

OPC UA defines a rich data model, with additional companion specifications for domain specific information models, such as ISA95/ISA88 plant hierarchy and batch control models, and IEC 61131-3 PLC models. All nodes in the AddressSpace of an OPC UA Server must be an instance of a one of a fixed set of NodeClasses defined in the standard. DPWS uses XML Schema types, and does not attempt to specify any other information model or data meta-model.

OPC UA has built-in security on multiple levels, requiring that clients establish a Session and Secure Channel to browse and access nodes in the AddressSpace. The specification includes WS-Security, WS-SecureConversation, and WS-Trust [56]. The standard also defines a binary message encoding (UA Binary) with a TCP (non-HTTP) transport mapping to avoid transmitting any messages in plain text. DPWS specifies an optional security model based on WS-Security, but can be extended with any other security model.

### 3.1.2 Adoption in Industry

OPC UA is being adopted at a much faster rate than DPWS in industrial applications. This could be due to a number of factors:

- The OPC UA specification is more complete, which makes interoperability easier to achieve. Integrators must only specify a plant-specific information model, and all layers below (data meta-model, encoding, transport, physical.) In a DPWS-based solution, interoperating devices must also have common security and information models, and they must understand the semantics of the services they are interoperating with.
- The OPC Foundation has stronger support and involvement from groups in the process, manufacturing, and automation industries, and was designed from the ground up for process control. DPWS originated from the high-level IT field, backed primarily by OASIS and Microsoft, with the goal of providing UPnP-like behaviour for networked devices using Web Service standards. Although much research has been done to assess the applicability of DPWS as a solution for Industrial SOA, most of the commercial implementations have been focused on home automation.
- OPC UA does not represent a major paradigm shift from traditional plant control and monitoring systems. It is an integration technology, which can be used to access data on existing devices, using a model that is compatible with existing fieldbuses: reading and writing nodes in some address space that correspond to physical inputs and outputs, or memory locations on devices. A full DPWS implementation, on the other hand, requires a departure from traditional thinking and substantial changes in overall

application architecture, because equipment must expose its capabilities as abstracted, composable services, rather than its raw memory and data values.

- In many industries, some Process Industries, for example, the dynamic discovery, reusability, and asynchronous eventing features of DPWS are either not desirable, or not worth the performance cost and uncertainty associated with verbose XML/SOAP message encoding, and non-deterministic transport protocols. In factories, fieldbuses, traditional PLCs, and explicit, cyclic access to raw data are desirable, and interoperability at Level 3 and above is sufficient.
- A typical plant may have tens or hundreds of OPC UA servers, serving as gateways to existing systems, controllers, fieldbuses, and devices. The same plant, on the other hand, may require tens of thousands of DPWS-compatible devices, along with a set of design, deployment, orchestration, management, and semantic mapping tools that do not currently exist.

For these reasons, DPWS adoption in industrial applications is less prevalent, and any existing non-OPC UA service-based offerings have proprietary, vendor-specific service semantics, and security and data models.

The advantage of DPWS is at the level of intelligent low-resource devices. While OPC UA is typically only found in Level 3 networks and higher, DPWS is a lighter-weight technology, and is well suited to being deployed on small, resource-constrained devices. A small, intelligent device with a fine-grained piece of functionality can expose a DPWS-compliant service interface, but it is unlikely to host a full OPC UA server. DPWS also supports dynamic discovery, and is a better fit for managing mobile or intermittently connected devices, which can join and leave networks, or migrate from one sub-network to another.

By merging these two technologies, a system can leverage the respective strengths of each, and create a true Service Oriented Architecture at the device-level. From DPWS, lightweight, simple, discoverable services at the device level. From OPC UA, the rich data model, security, and market momentum.

### 3.1.3 Technology Merging Strategies

A number of potential merging strategies have been proposed. These strategies can be roughly categorized as follows:

- OPC UA over DPWS: Implementing the OPC UA Service Sets in a DPWS-compliant client or server, including all session and security specifications, and implementing a UA-Binary mapping and OPC UA TCP transport binding.
- DPWS over OPC UA: Implement a DPWS Client extension to an OPC UA server, which can discover DPWS devices in a sub-network and map them into the server's AddressSpace

- Merging at the protocol-level: Implement a Web Services protocol stack that supports both OPC UA and DPWS
- Merger through another technology: Creating OPC UA and DPWS input and output adapters for some other application, such as a Complex Event Processing (CEP) Engine or Database.

Some examples are described in brief below:

Candido et.al. extend the SOCRADES proposal [46] by starting with DPWS and WS-Management for a generic device-level framework. DPWS provides a general purpose, extensible architecture for WS-based interoperability at the device-level, while WS-Management provides a standard for managing resources (servers, devices, information). They then suggest extending it with support for the additional protocols required by OPC-UA, including service sets, security specifications, UA Binary mapping, and UA Native (TCP) transport protocol. Furthermore, they advocate defining a two-way mapping between device data and the OPC UA Object Model, and a low-resource version of the combined protocol.

Bony et al. propose another approach to convergence, by implementing the following components [47]:

- A Node Manager containing a cross-layer domain data model and address space, and exposes an interface to handles access to data in the address space.
- The application (includes devices) Gets and Sets information via the node manager interface
- Client is connected to the network, requests data from server, and sets parameters

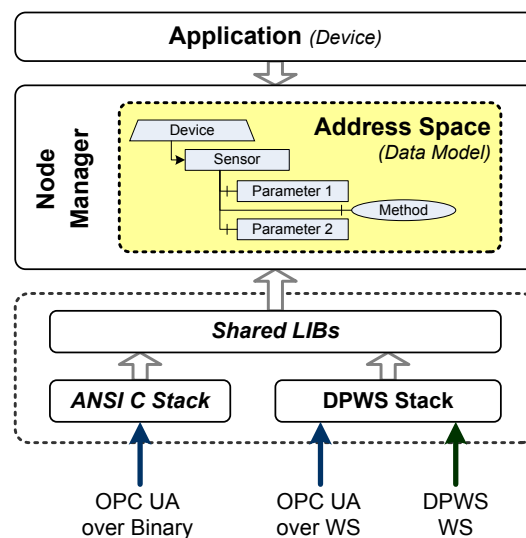


Figure 16: OPC UA - DPWS Convergence Prototype [47]

For the communication stack, they propose two parallel stacks at the encoding and transport levels: an ANSI C Stack for UA Binary over TCP-based UA Native, and a combined Web Services stack (SOAP/XML over HTTP/HTTPS) for OPC WSs and DPWS WSs. Both stacks would access shared libraries, which manage endpoints, certificates, sessions, and binding to the node manager. The combined stack is shown in

Figure 16.

In addition, they present some suggestions for merging various parts of the standards:

- Extend OPC UA with dynamic discovering using WS-Discovery, implemented only in OPC UA Discovery servers, or in all OPC UA servers.
- OPC UA as a Middleware Server: An OPC UA Server maintains a virtual instance of every DPWS device discovered in the address space, dynamically WS-Discovery mechanisms, such as discovery probes and Hello/Goodbye messages. Vendor-defined hosted web services could be exposed as OPC UA methods, invoked by clients using the appropriate OPC UA service set, or the Server would read and store all parameter values of a device periodically.

Garcia et al. [50] proposed a convergence of DPWS and OPC UA eventing mechanisms by mapping OPC UA Notifications and DPWS Event messages into a common Complex Event Processing (CEP) engine event format. An Event Translator component has an integrated DPWS client stack for discovering devices and subscribing to WS-Eventing push events, and an OPC UA client for connecting to an OPC UA Server to configure Monitored Items and receive responses to Publish requests. This middleware component converts incoming DPWS and OPC UA into a generic event stream, consumed by the CEP engine.

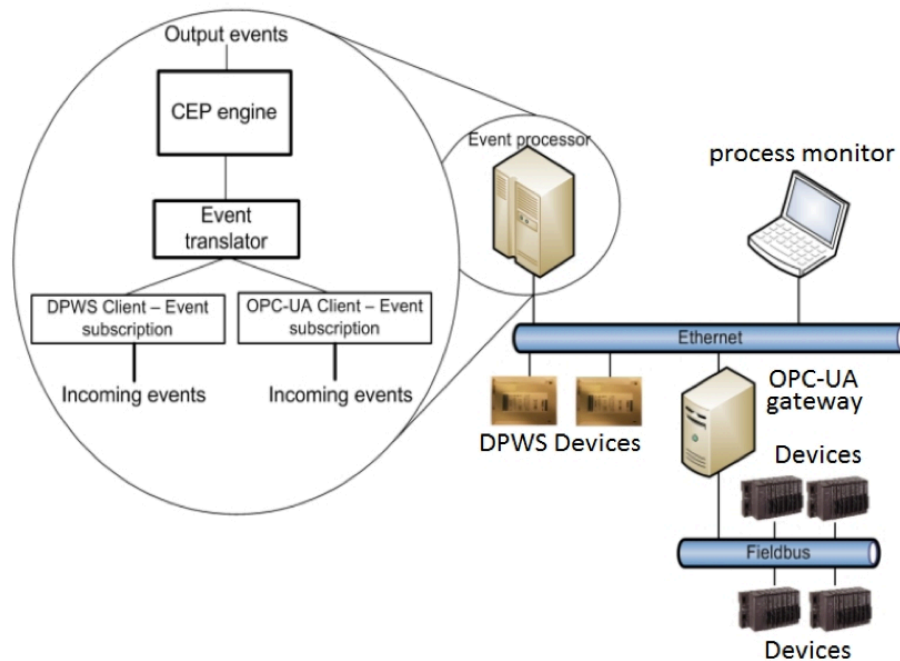


Figure 17: DPWS/OPC UA Architecture for Event Processing

### 3.1.4 Chosen Approach

All of the OPC UA/DPWS convergence strategies described previously have their merits. However, given the realities of today's market, and the comparative advantages and disadvantages of each technology, an alternative approach is presented:

- Define a representation of DPWS devices using the OPC UA Object model. This representation must be based on:
  - WSDL descriptions of DPWS hosted services
  - OPC UA Device Information Model Specifications
- Implement an OPC UA gateway to DPWS, consisting of
  - A DPWS Client, which discovers devices, creates and maintains a representation of the device, its hosted services, and input and output parameters in an OPC UA AddressSpace, and manages service invocations and event subscriptions.
  - An OPC UA Server, which enables OPC UA clients to browse the representations of all devices, and invoke and subscribe to the available operations and events.

This approach has the following advantages:

- Light-weight DPWS is used at ISA 95 Levels 2 and 1, where it performs best. Devices can be discovered, and forgotten. Once devices are discovered, OPC UA's built-in status and data quality attributes for objects can carry current information about connectivity, device health, most recent invocation



or time since last event. This is especially advantageous for persistent representations of intermittently connected devices.

- OPC UA is used for interoperability at higher levels, taking advantage of its data model, security mechanisms, and industry presence. DPWS can be used with minimal security overhead in local subnets, and cross-network secure transactions can take place via OPC UA without the need for establishing new security specifications.
- Device functionality is represented in the Object Model, using a WSDL-based service representation. Device memory or physical IOs are not exposed.
- Mature DPWS and OPC UA stacks already exist.

Following this approach, a prototype OPC UA Gateway to DPWS devices is implemented.

## 3.2 COMPONENT SELECTION

### 3.2.1 OPC UA Client and Server SDKs

There are few free or open-source OPC UA stacks currently available for this style of development. The OPC Foundation maintains an official OPC UA SDK for both ANSI C and .NET, available only to corporate members. Several commercial offerings are available:

*Table 11: Commercial OPC UA Client and Server SDKs*

Company	Name	Description
<b>Embedded Labs</b> www.embeddedlabs.com	OPC UA Device Server SDK	This SDK is designed to allow embedded developers to easily add an OPC UA Server to their resource-constricted microcontroller- or microprocessor-based product.
<b>Inductive Automation</b> www.inductiveautomation.com	Ignition OPC UA Server	Ignition is a mature, web-based, industrial application server, with HMI, SCADA, and MES capabilities. The Ignition OPC UA Server is a module for Ignition, which can serve as a standalone OPC UA server. Ignition features a free driver API and SDK for developing custom modules for

		interfacing with all parts of Ignition, including the OPC UA server.
<b>Prosys</b> www.prosysopc.com	OPC UA Java SDK	This commercial SDK includes OPC UA client and server stacks that can be integrated royalty-free into custom applications. A free evaluation version can be requested.
<b>Softing</b> www.softing.com	OPC Toolbox UA	Another commercial SDK for easily incorporating full OPC UA functionality into client and server applications. A free downloadable demo is available
<b>Unified Automation</b> www.unified-automation.com	C++ and ANSI C OPC UA Client and Server SDKs	Server and Client SDK/Toolkit includes precompiled libraries and header files, documentation and samples, as well as utilities for designing information models and generating code.

---

Of these options, the Ignition OPC UA Server from Inductive Automation is ideal, for the following reasons:

1. The OPC UA Server is integrated into a larger system designed for industrial control systems.
2. The Ignition OPC UA Server license is free for end-users, and the licensing is favourable for developing experimental modules and drivers using the Ignition Developer SDK
3. Ignition Server is written in Java, so it can be run on any platform that supports Java (Windows, OS X, Linux, etc.), and an existing DPWS stack implemented in Java can be integrated into Ignition Modules
4. An infinitely restartable two-hour trial of the integrated web-based HMI, and ability to create custom visual components for the HMI

### 3.2.2 Ignition OPC UA Server

Ignition is an industrial application server from Inductive Automation[45], used to create HMI, SCADA, and MES systems. Ignition (formerly FactorySQL and FactoryPMI) is a mature, well-tested application. The feature list includes

- web-based gateway configuration and HMI drag-and-drop editor
- a rich set of visual components

- database-centric architecture
- advanced reporting and alerting mechanisms
- designed from the ground-up for scalability.
- implemented in Java, so it can be run on a wide range of platforms, on all major operating systems.
- Control system access on mobile devices

The OPC UA Specification has gained significant industry traction since its release, and Ignition has gained significant popularity over the last few years resulting from the addition of a built-in OPC UA Module, using a custom OPC UA stack, to its feature set. The OPC UA Module can act as a communication layer for other modules, or as a standalone OPC UA Server for third party OPC UA clients. Currently, drivers are available for Modbus Ethernet, as well as Allen Bradley and Siemens S7 PLCs, but more are in development. In the interest of performance, Ignition OPC UA supports only the UA-Binary/TCP transport encoding and protocol, not the XML/text mapping via SOAP-over-HTTP.

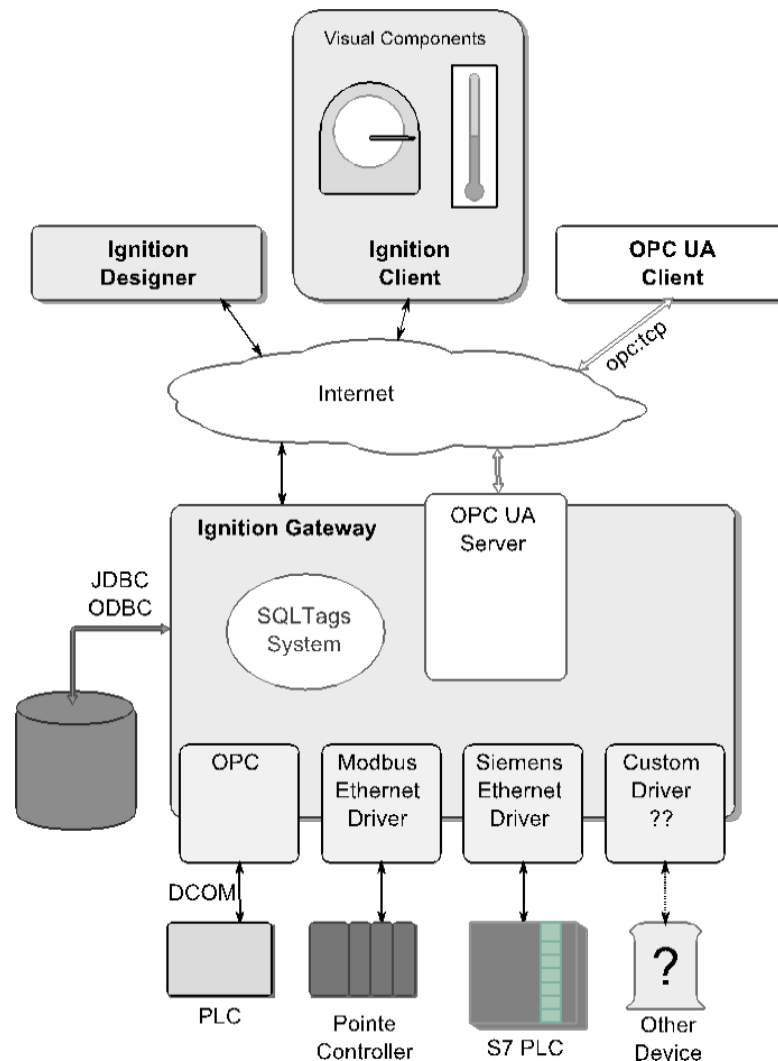


Figure 18: Ignition OPC UA Server Connections

Ignition also features an open driver API, allowing development of OPC UA driver modules, and general Ignition modules in Java.

An example of a system built around an Ignition OPC UA server is shown in Figure 18. The core of an Ignition-based application is the Ignition gateway. The gateway can be distributed over multiple hardware devices for scalability or redundancy. The gateway has an HTML configuration interface that can be accessed through a browser. Modules can add additional communication capabilities to the Gateway, and expose memory or information as nodes in the OPC UA Address Space. For example, the Modbus Over Ethernet Driver for Ignition is a module that allows the addressable memory on a Modbus device to be exposed as nodes.

The Ignition Designer is launched from the Gateway Configuration interface in a browser. The Java-based designer allows the user to link SQLTags to OPC UA nodes, SQL queries in a database, or functions of other SQLTags, expressed in Python. Client interface windows can be designed by adding visual components to display and set the information contained in SQLTags. The client programs can then be launched from the gateway configuration page.

The Ignition OPC UA Server has some disadvantages. While the SDK does not explicitly exclude objects, it is suited best to a simplified view of the OPC UA Object Model, which corresponds one to one with the SQLTags system that the Ignition Gateway is built around. This simplified view consists only of FolderNodes and DataVariableNodes. Functionally, a system built with these restrictions would be identical to a full implementation, but the potential for enriching data with semantic information by structuring the data using the object model is reduced.

OPC UA Methods are not supported in the Ignition Client or Ignition Designer. The OPC UA Server can expose methods, but to link Ignition Visual Components to operations or event output parameters, another approach is required.

### 3.2.3 The Ignition SDK

The Ignition SDK is a collection of libraries and sample code for creating custom Ignition Modules in Eclipse IDE. Three types of modules can be created:

- Gateway Module: These modules can provide new communication interfaces, manipulate a database, provide SQLTags to the Ignition System, or add Nodes to the OPC UA Address Space.
- Designer Module: Programming for the Ignition Designer. This can include making custom menus, buttons, and toolbars, or adding custom client-creation utilities.
- Visual Components: Custom visual components, written in Java, similar to any 2D or 3D Java Swing or AWT or user interface. Accessing custom visual components requires a custom designer module, which creates a new component pallet, and adds the custom components.

A new gateway module can be created, containing a DPWS client stack. This client stack will discover DPWS devices, and create representations of the devices as nodes in the OPC UA Address Space. Any OPC UA client connecting to the Ignition OPC UA server must be able to invoke operations, and subscribe to events on the discovered DPWS devices, using the OPC UA service sets.

### **3.2.4 JMEDS WS4D DPWS Java Stack**

Custom Ignition Gateway Modules are written in Java. The WS4D JMEDS Java Stack is a full open-source DPWS implementation, with two versions: Version 0.9.7 supports DPWS v1.0, while v2.3.7 supports DPWS v1.1. The stack is used to discover devices, and parse device and hosted service metadata to create a java object structure representing the device, along with all services, operations, and events, including input and output parameters.

JMEDS is available for download at <http://ws4d.e-technik.uni-rostock.de/jmeds/>, under the Eclipse Public License[58].

### **3.2.5 InicoTech S1000 Smart RTU**

The S1000 from InicoTech is a compact controller with a DPWS communication stack. The device executes program code written in Structured Text (ST), an IEC 61131-3 programming language, and can be configured with custom Web Services. The device can host operations, and publish events using WS-Eventing push mode. A device can also invoke operations on other WS-enabled devices, provided that the request and response messages are pre-configured.

The S1000 comes standard with eight digital inputs, eight digital outputs, and an RS-232 serial communication port. The version used in this project also features four analog inputs. Other IO expansion modules are available, including additional digital or analog IO, high-speed pulse counter, and three-phase power meter.

The device supports WS-Discovery, including listening on UDP port 3702 for discovery probes, and broadcasting Hello/Goodbye messages when joining and leaving a network. The device will supply references to WSDL files for each hosted service in response to a GetMetadata Request message, which describes the action identifiers, and input and output message format v the operations and events for the hosted service.

These devices are selected, because they are uniquely well suited to this type of research activity, no other programmable DPWS devices are commercially available at time of writing, and deployment and testing with a large number of devices is relatively painless.

### 3.3 PROPOSED INTEGRATION APPROACH

To create representations of discovered DPWS devices in the AddressSpace of an OPC UA Server, we can use the device metadata, and the WSDL descriptions of hosted services.

As discussed in Section 2.2.2, the WSDL file describes the web service interface, including all operations, events, and input and output message format. The first step is to establish a mapping from the WSDL file to the OPC UA Node model.

#### 3.3.1 Mapping WSDL to OPC UA Address Space

In Section 2.4.3, the ObjectTypes introduced in the OPC UA Companion Specification “OPC UA for Devices [49]” are described. This can be used as a starting point for representing DPWS devices in the OPC UA Object Model:

- The DeviceType object type, extending the TopologyElement type, can be used to represent the Hosting Service.
- A special functional group called “Identification” is specified, for organizing metadata for Topology elements, stored as Parameters in the TopologyElement’s ParameterSet. Alternatively, the DeviceType can be extended to include attributes for representing DPWS-specific device metadata, returned in response to a WS-Transfer “Get” request.
- In the simplest case, each Hosted Service can be represented by a Functional Group, which organizes a set of methods and parameters.
- Alternatively, if a more complex structure is required, the Hosted Services could themselves be represented as extensions of the TopologyElement type, containing FunctionalGroups to organize the operations.
- For including semantic data, new types can be defined, extending the existing MethodTypes and ParameterTypes, to include the extension attributes and elements defined in Web Service Semantics (WSDL-S) [16]

A generic mapping is shown in Figure 19. The MethodSet gathers all the methods that are exposed to the client, and the ParameterSet gathers all parameters that the device has. The FunctionalGroups representing the hosted services organize the methods and parameters of the device. Multiple FunctionalGroups can reference the same methods and parameters.

Asynchronous push-mode events defined in WS-Eventing do not clearly fit into the OPC UA for Devices Object Model. One approach is shown in Figure 19, although many different approaches could be designed. Events are grouped in a separate functional group, nested within the Hosted Service, with the appropriate output parameters, and a method for subscribing and unsubscribing to each event.

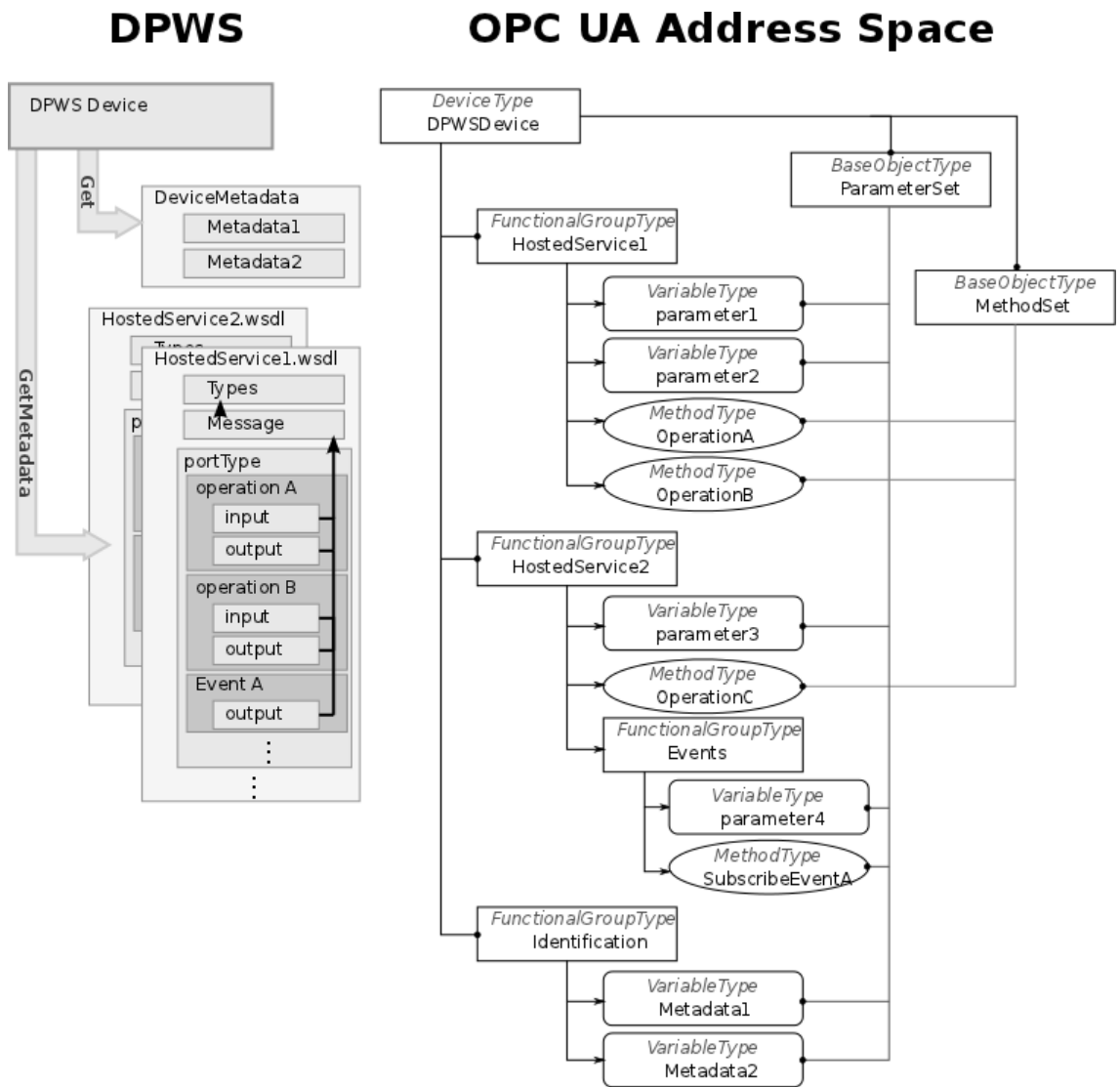


Figure 19: Generic Mapping from DPWS to the OPC UA for Devices Object Model

The data values themselves must be represented in the parameters. DPWS uses XML Schema types, whereas OPC UA defines a fixed set of primitive types that cannot be extended. Fortunately, Part 6 of the OPC UA Specification defines mappings from OPC UA primitive types to XML Schema types. These are summarized in Table 12. When mapping values between OPC UA Nodes and DPWS messages, this table is followed.

Table 12: Mapping between OPC UA Primitive Types and XML Schema Types [55]

OPC UA Primitive Type	XML Schema Types
<b>Integer</b>	
SByte	xsd:byte
Byte	xsd:unsignedByte
Int16	xsd:short
UInt16	xsd:unsignedShort

OPC UA Primitive Type	XML Schema Types
Int32	xsd:int
UInt32	xsd:unsignedInt
Int64	xsd:long
UInt64	xsd:unsignedLong
<b>Floating Point</b>	
Float	xsd:float
Double	xsd:double
<b>Other</b>	
String	xsd:string
DateTime	xsd:dateTime
Guid	xsd:string
ByteString	xsd:base64Binary
XML Element	xsd:complexType
NodeId,	xsd:string
ExpandedNodeId,	xsd:string
StatusCode,	xsd:unsignedInt
DiagnosticInfo, QualifiedName,, LocalizedText, ExtensionObject, Variant, DataValue, Enumerations, Arrays, Structures, Messages	Some xs:complexType. See OPC UA Specification Part 6: Mappings

Simple types can be represented as simply as named parameters of the appropriate OPC UA primitive type. Complex Types can also be easily represented using an appropriately structured object.

Now that the DPWS device, service, and operation representation in the OPC UA Address Space has been established, a proof of concept system can be implemented, following the integration approach proposed in Section 3.1.4.



## 4 IMPLEMENTATION

### 4.1 SYSTEM OVERVIEW

In Section 3.2, components were selected for implementing a system to demonstrate a merger between OPC UA and DPWS. The following components were selected:

- **OPC UA Server:** Ignition OPC UA Server from Inductive Automation, and the Ignition Java SDK [45]
- **DPWS Stack:** Java Multi-Edition DPWS Stack (JMEDS) from WS4D.org [58]
- **DPWS-enabled Devices:** InicoTech S1000 Smart RTU [59], and virtual devices, implemented with WS4D's JMEDS [58].

The system diagram is shown in Figure 20. The core of the system is the Ignition Gateway, hosting the OPC UA Server. The Ignition Gateway has a browser interface, which is used for gateway configuration, connecting databases, installing and configuring devices and additional modules, and launching the Ignition Designer and Client applications. The Ignition Project Designer is used for creating graphical user interfaces (SCADA HMIs) for Client applications, driven by the SQLTags system.

### 4.2 DPWS MODULE FOR IGNITION

The DPWS Driver is written using the Ignition SDK, and includes JMEDS. The DPWS Driver module:

- Uses WS-Discovery to dynamically discover all DPWS devices in the network
- Creates, manages, updates, and deletes representations of the devices in the OPC UA Address Space, and in the Ignition SQLTags system, and maintains consistency between the two representations
- Connects the actual device with its representation in the OPC UA Address Space
- Handles communication with the discovered devices, including subscription management, receiving events, and operation invocations and responses

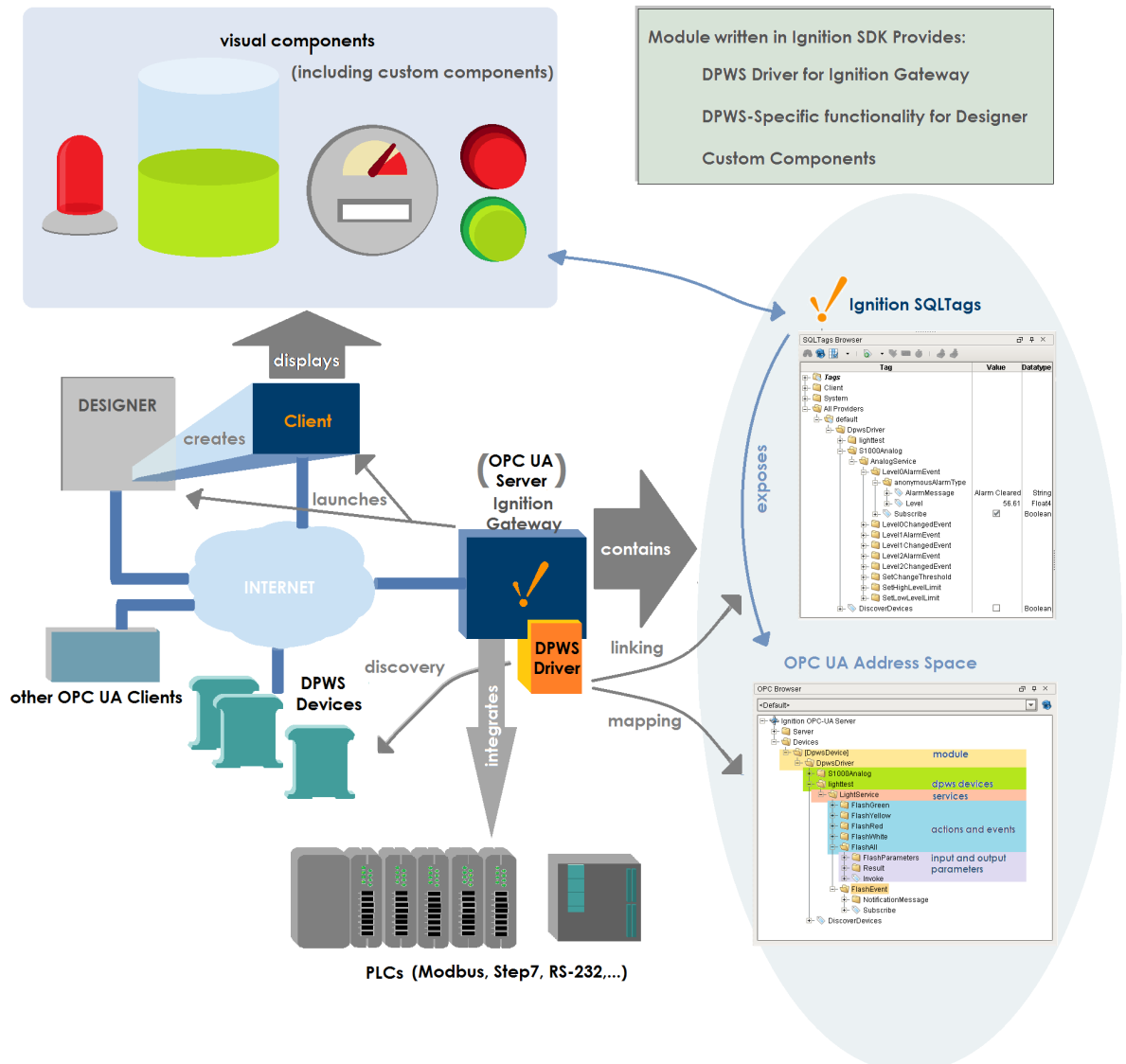


Figure 20: Ignition DPWS Module: System Overview

As of version 7.2.8 of Ignition Server and the Ignition SDK, three different approaches exist for adding nodes to the OPC UA address space [60]:

1. Using The OPC UA Server Connection Extension Point: Create a custom implementation of an OPC UA Server, and register it with the system through the OPCManager
2. Create and manage nodes on the default Ignition OPC UA Server using the interface provided.
3. Implement a NodeMapDriver, which is added as an OPC UA Device in the gateway configuration

To use the Ignition Vision Module for creating and displaying graphical HMIs, SQLTags must be created, linked to the appropriate DataVariableNode in the OPC UA Address Space. Tags can be created and managed using the API to the system default tag provider, or a custom tag provider can be written and registered.

For the DPWS Module, implementing a NodeMapDriver for handling the OPC UA nodes, and a custom Tag Provider for creating and updating the SQLTags appears to lead to the optimal balance between performance and maximizing reuse of existing utility classes and object implementations in the SDK.

DataVariableNodes and SQLTags that correspond to parameters in an service response message, or an event message are defined as “Read Only,” so that only the back-end DPWS Module code can change these values.

As discussed in Section 3.2.2, the NodeMapDriver interface and the OPC Browser in the Ignition designer present a simplified view of an OPC UA Address space, limited to FolderNodes and DataVariableNodes. The representation of a WSDL service interface description in an OPC UA address space proposed in Section 3.3 is modified to meet these restrictions. The representation, as implemented, is shown in Figure 21.

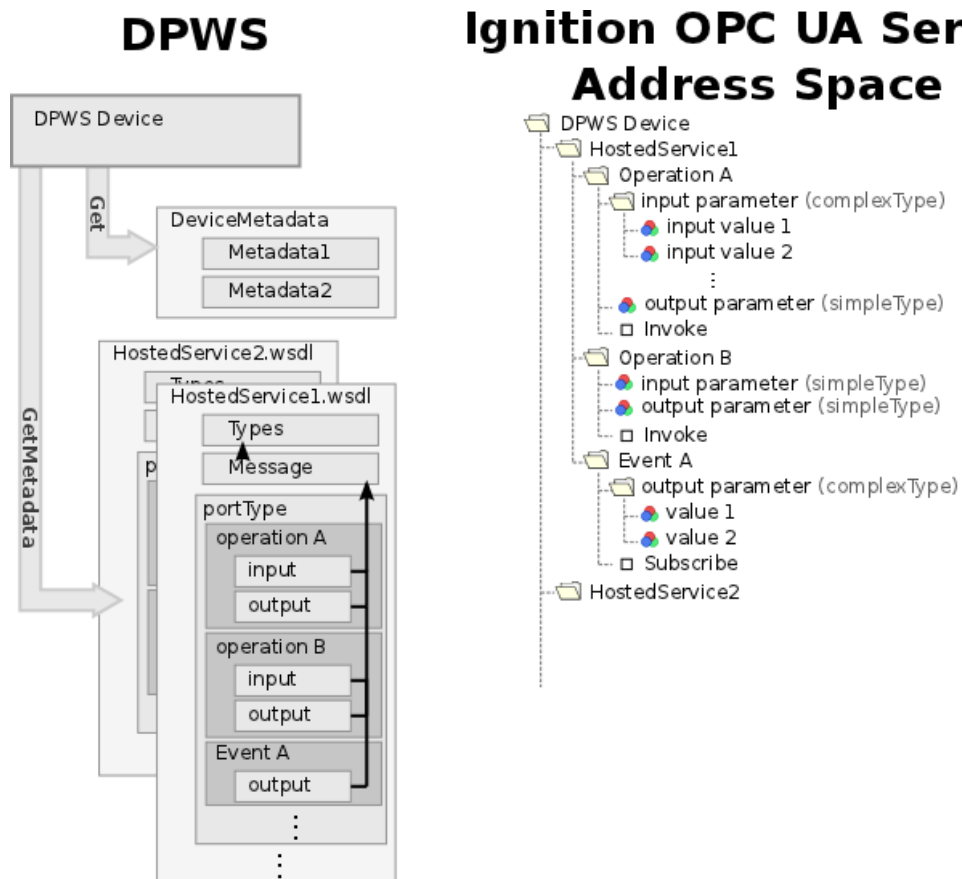


Figure 21: Simplified DPWS to OPC UA Device Representation for Ignition OPC UA Server

Functionally, this implementation would be identical to a solution using a more object-oriented representation, but the semantic and organizational strengths of the OPC UA data model are not being leveraged.

### 4.2.1 Ignition Designer Interface

When the designer is launched, the user creates a new project. An annotated view of the Ignition Designer is shown in Figure 22. Various browsers appear down the left side, including:

- Project Browser, for managing project resources, and creating HMI Windows for editing
- SQLTags Browser, for creating and managing SQLTags, which can be linked to OPC UA Data Variable Nodes, historical data in an attached Database, functions of other SQL Tags, or just used as variables. Ignition Visual Components use SQL Tags for input and output.
- OPC Browser, for viewing the OPC UA Address Space. This interface only shows the nodes themselves, not the values of any attributes. When an SQL Tag is created and linked to an OPC UA Node, changes in the value of one are reflected in the other, after some delay configured in the scan class associated with the tag.

For building HMIs, Ignition provides a large set of visual components, including text and numeric input and output, tanks, and gauges. Custom components can also be designed. A simple example of a custom component was created, to show the state of the digital inputs for the S1000, taking change-based events as input.

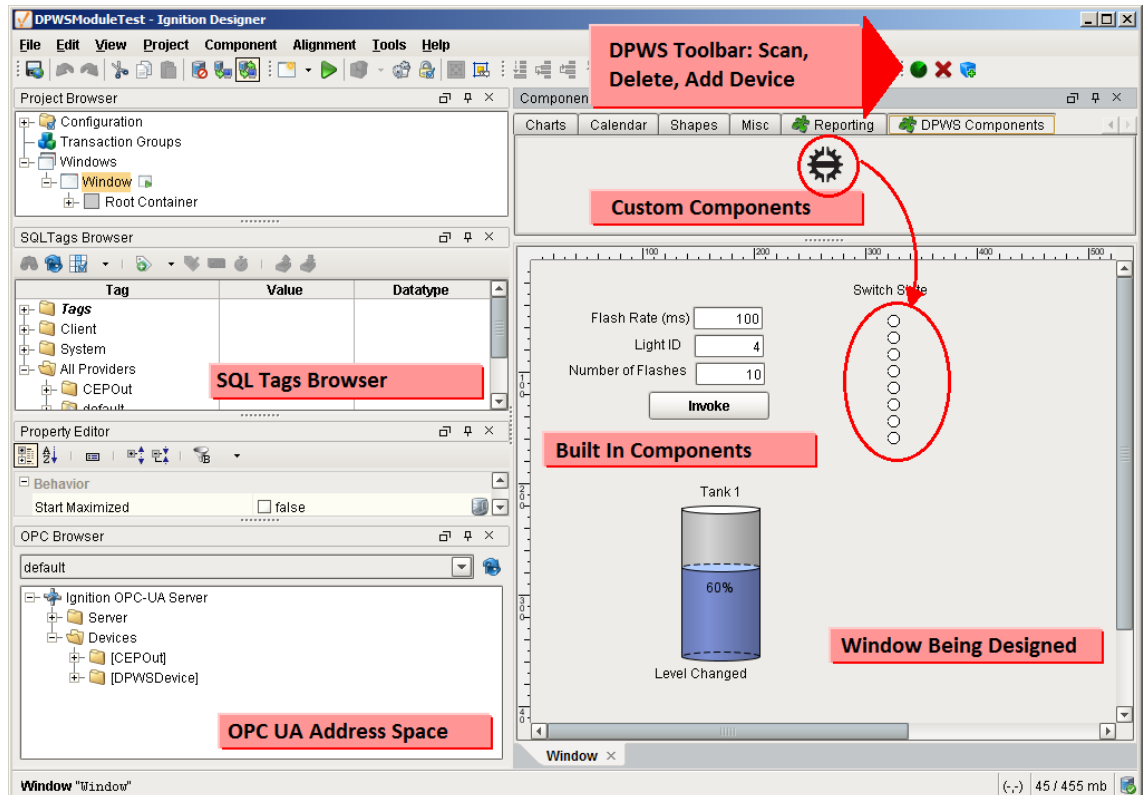


Figure 22: DPWS Module for Ignition Server - View from Designer

A DPWS toolbar was added to the designer for discovering devices in the local network, removing discovered devices, including ending all active subscriptions, deleting all OPC UA nodes and SQL tags created when the device was discovered.

When a device is discovered, a representation is created in the OPC UA address space as described in Figure 21. A single device, hosting a single service with several events and operations, is shown in the OPC Browser and the SQL Tag Browser in Figure 22. The organization of the SQL Tags is changed slightly for convenience.

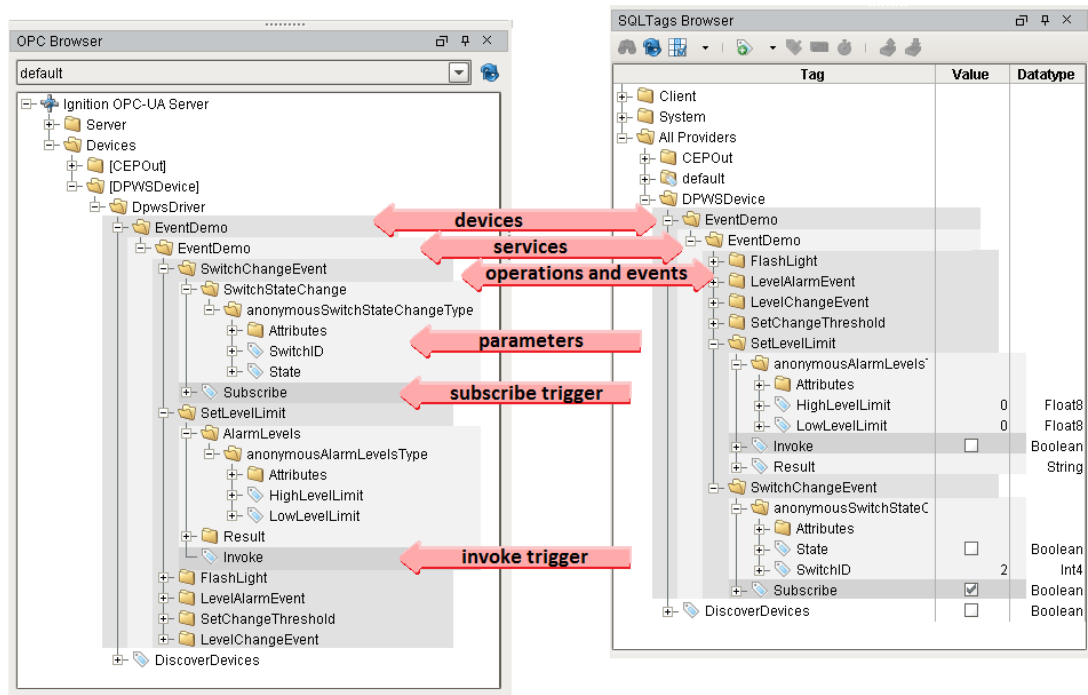


Figure 23: OPC UA Address Space and SQLTags Browser panels in Ignition Designer

When the HMI is created and published, the HMI application can be launched from the front page of the Ignition Gateway browser interface. The Client is shown in Figure 24.

To improve the responsiveness when receiving DPWS events, the custom Tag provider and the Node Map Driver are connected using an asynchronous eventing mechanism to minimize the delay between the time that the DPWS client receives the event, and the time that the values received are reflected in the OPC UA address space, and SQL Tags. The default scan class detects changes every 0.5 seconds, and reducing this time caused performance to degrade as the number of devices and nodes increased.

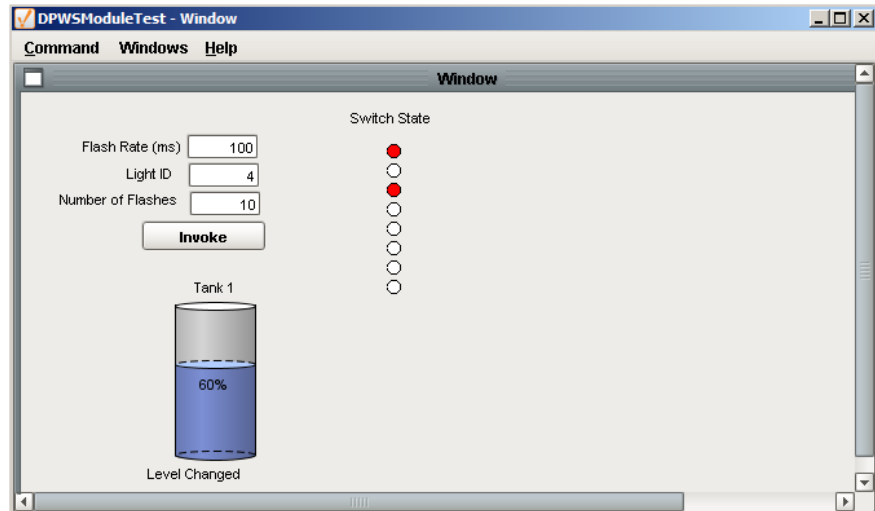


Figure 24: Ignition Client HMI

The OPC UA address space of the Ignition Server can be navigated and manipulated using any third party OPC UA Client. For example, a free client available from Unified Automation is used to subscribe to an event and monitor the contents of the event message in Figure 25.

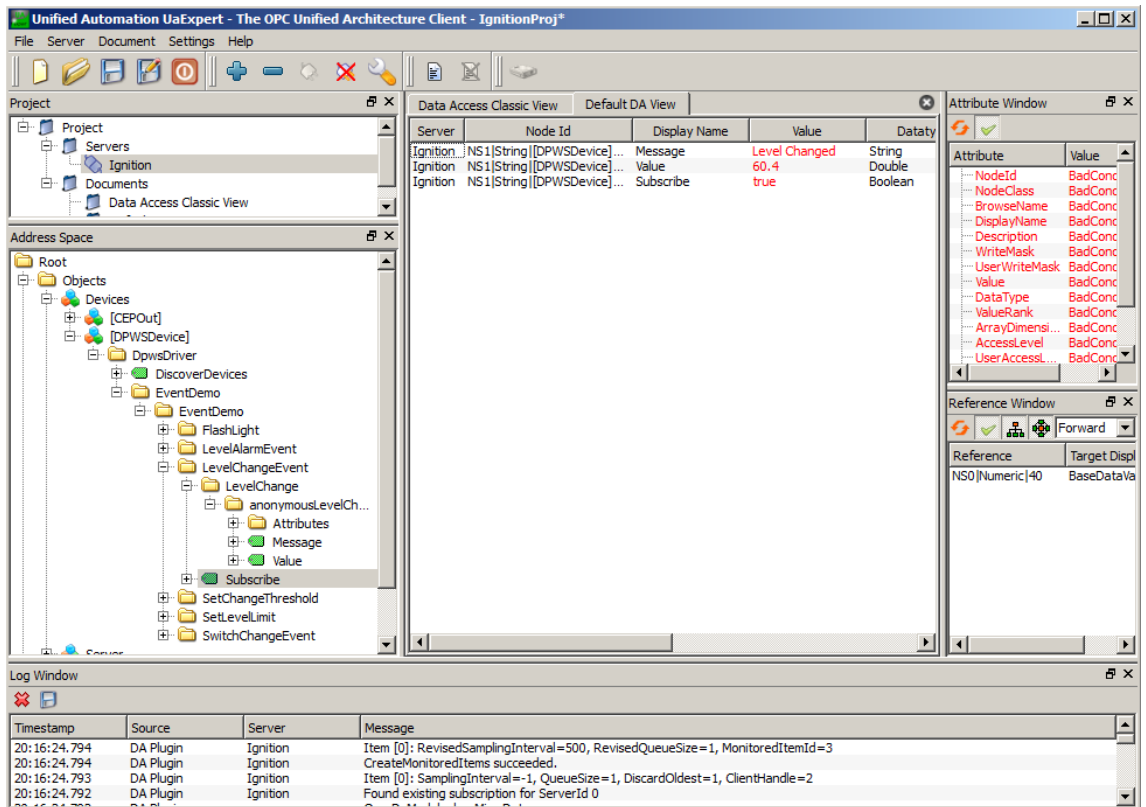


Figure 25; View of Address Space of Ignition OPC UA Server from a third party OPC UA Client

### 4.3 MICROSOFT STREAMINSIGHT COMPONENT

As discussed in Section 2.5, in an Event Driven SOA, communication is in the form of passing asynchronous event notification messages, and service invocations are triggered by internal or external events. From a manufacturing perspective, events can include equipment state changes, customer orders, changes in market conditions, or changes in measured values. Event-driven applications can have very high event rates, strict latency requirements on the order of milliseconds, and a need for systems processing continuous queries on incoming event streams.

Complex Event Processing (CEP) engines are a powerful tool for deducing high-level information about overall system state or patterns. StreamInsight is a CEP engine from Microsoft, included as a component of SQL Server 2008 R2. To enhance the proof-of-concept system, a component with StreamInsight, with Web Service input and output adapters was created to demonstrate integrating a CEP engine executing continuous queries into a DPWS and OPC UA based monitoring and supervisory control system. A rough system overview is shown in Figure 26.

This component is not described exhaustively, because the focus of the thesis is the DPWS and OPC UA integration. It is, however, relevant when discussing event-driven SOA, and a CEP engine is a significant part of EDA.

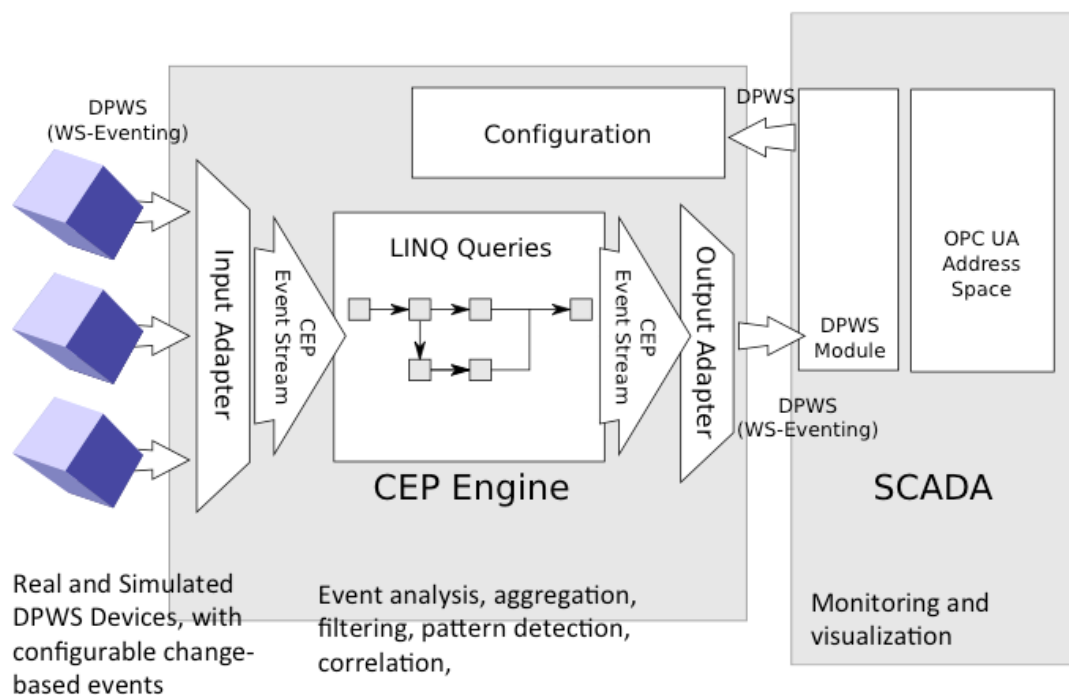


Figure 26: Diagram of StreamInsight CEP Component performing DPWS event filtering

Development for StreamInsight applications is done in C#. As indicated in Section 2.2.3, Microsoft provides two DPWS implementations: The .NET MicroFramework,

and WSDAPI. To accelerate prototype development, the subscription model from WS-Eventing was not used. Instead, devices publish event notifications to a well-known web service endpoint, exposed by the StreamInsight input adapter. Similarly, the output adapter publishes complex events, the output of queries executing continuously on a stream of input events, to a well-known endpoint exposed by another module running in Ignition Server. The Ignition CEP Output Adapter Sink Module creates a service and operation, with a representation in the OPC UA address space designed according to the same approach as the Ignition DPWS Module.

For demonstration purposes, the low-level events are generated by an S1000 from InicoTech. An event is generated when the state of a digital input changes, and a notification message is to the StreamInsight input adapter, containing the number of the input (1 to 8), and the new state (TRUE/FALSE). The input adapter transforms the incoming SOAP message into a CepStream event type, and the CEP executes the queries, defined in Language Integrated Query (LINQ). A sample two-stage query is shown below:

```
var Q1 = from m in input.TumblingWindow(TimeSpan.FromSeconds(1),
    HoppingWindowOutputPolicy.ClipToWindowEnd)
    select new EventType{
        ID = 12,
        State = m.OneAndTwo()
    };

var Q2 = from e in Q1
    where e.State == true
    select e;
```

*Sample Code 10: Code for defining sample query in C# for StreamInsight*

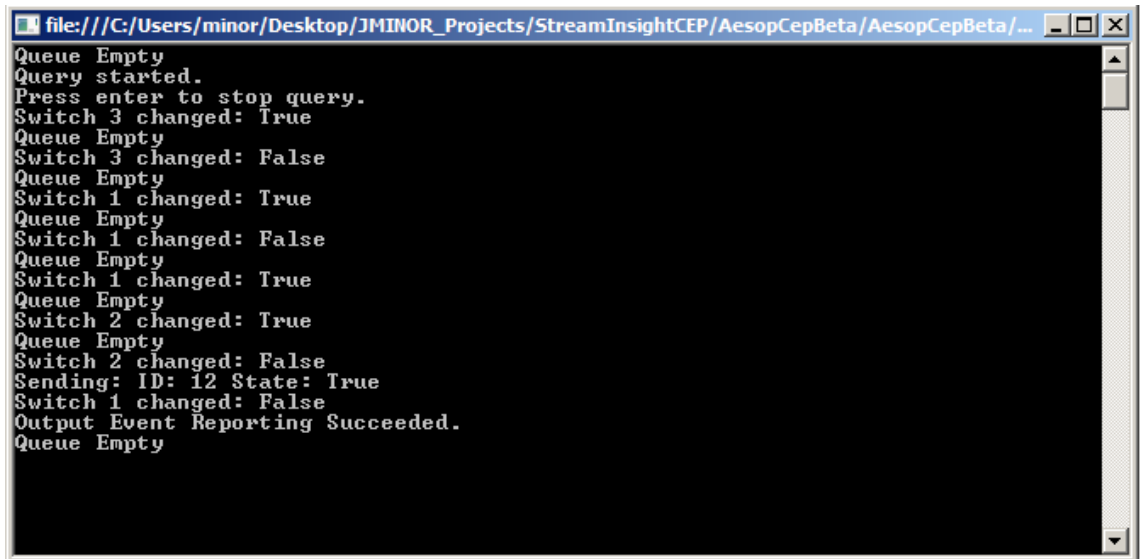
In this particular query, an output event is generated when the first and second inputs change state to TRUE within a tumbling 1-second window. The first query, Q1, creates an output event, with an ID, and a Boolean state, which is some function of all the incoming events in the previous 1 second window. Query Q2 takes the output of Q1, and generates an output event based if the State variable of the Q1 output event is true.

It should be noted that designing complex LINQ queries that apply to real use cases is not the focus of this thesis.

Figure 27 shows the output to the console window from the StreamInsight program, reacting to arbitrary manual switch changes. The event rate in the demo is low, so the input and output event queues never contain more than a single event.

When switches 1 and 2 are set to the high position in short succession, this is detected at the end of the 1s window, and an output event is generated, and the code '12' is reported to the CEP Output Sink module in Ignition.





```

file:///C:/Users/minor/Desktop/JMINOR_Projects/StreamInsightCEP/AesopCepBeta/AesopCepBeta/...
Queue Empty
Query started.
Press enter to stop query.
Switch 3 changed: True
Queue Empty
Switch 3 changed: False
Queue Empty
Switch 1 changed: True
Queue Empty
Switch 1 changed: False
Queue Empty
Switch 1 changed: True
Queue Empty
Switch 2 changed: True
Queue Empty
Switch 2 changed: False
Sending: ID: 12 State: True
Switch 1 changed: False
Output Event Reporting Succeeded.
Queue Empty

```

Figure 27: Console application executing StreamInsight Queries

The Address Space of the Ignition OPC UA server, as seen from the UAExpert third party OPC UA Client browser can be seen in Figure 28. An output event has been successfully received, and this is reflected in the OPC UA's representation of the Service.

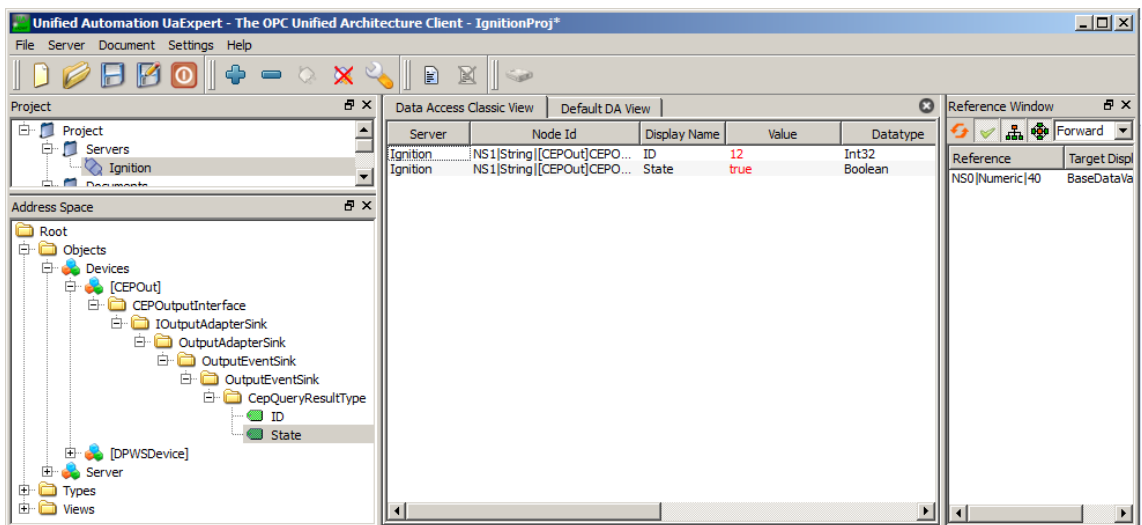


Figure 28: Ignition OPC UA Address Space, as seen in UAExpert

These OPC UA Nodes have corresponding SQL Tags, similar to the discovered devices and services in Figure 23, and can be connected to visual components, or logged in a database, using Ignition-provided functionality.

#### 4.4 OVERALL SYSTEM STRUCTURE

A diagram of the complete prototype system is shown in Figure 29, showing the key classes in the DPWS Module for Ignition, and the interaction points between various parts of the system.

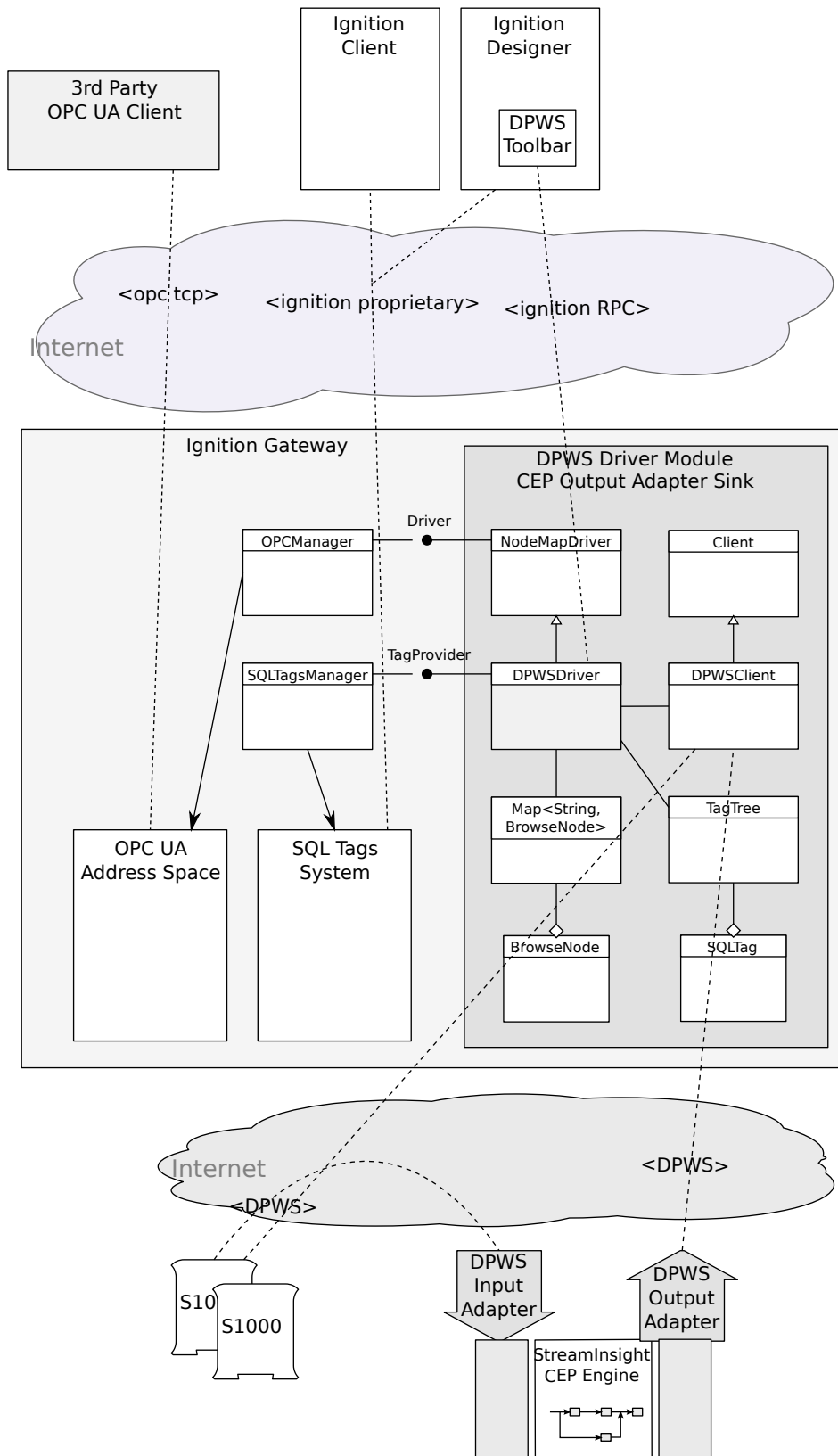


Figure 29: Diagram of complete prototype system, showing interactions and important classes

Some of the important classes were introduced previously, but they are summarized here. The DPWSDriver class extends NodeMapDriver, which already implements some of the OPC UA Node searching, reading, and querying functionality. Some of this functionality is overridden to minimize time delay between SQLTag and OPC UA node updates.

The DPWSDriver also implements the TagProvider interface, and registers itself with the SQLTagsManager on startup. In the SQLTags Browser in Ignition Designer, the DPWSDriver module shows up as a separate Tag provider. The class also contains a TagTree, a data structure for managing SQLTags, and a Map structure, for mapping node addresses to the actual OPC UA Nodes.

The DPWS module also implements some custom BrowseNode and SQLTag types, optimized for module-specific tag types, such as subscription tags and operation invoke triggers.

The DPWSDriver class also creates an instance of DPWSClient, an extension of the Client class in the WS4D Stack. This class handles discovery, subscribing and receiving events, and invoking operations.

Two most important classes provided by the Ignition Gateway for this application are the OPCManager, and SQLTagsManager. These are the major interfaces between the module and the rest of the system, including the OPC UA AddressSpace of the server, and the SQLTags system. External OPC UA Clients can access the nodes provided by the DPWS Module through the address space, and the Ignition Designer and Client access the Tags provided by the module through the Ignition SQLTags system.

A DPWS Toolbar was added to the Ignition Designer. For discovery triggering and device management, a Remote Procedure Call (RPC) interface was established between the Ignition Designer and the DPWSDriver Module. In future implementations, this devices could be managed via OPC UA, and this RPC interface would be unnecessary.

The design of the CEP Output Adapter Sink module and DPWSDriver modules are similar, and were only separated for the purposes of rapid prototyping. The DPWS devices send notifications to the DPWS Input Adapter for StreamInsight, and when the continuously executing queries produce an output event, the DPWS output adapter forwards this notification to the module. The code for handling tags and OPC UA nodes is identical.

## 5 DISCUSSION OF RESULTS

The result of this project was a working prototype of an industrial SCADA system for monitoring and control, bridging two web service-based integration specifications: OPC UA and DPWS. The general approach was to define a representation for DPWS devices in the OPC UA address space, using existing specifications as a guide, and implement a system prototype. To demonstrate the feasibility of adding CEP functionality to the system for deducing higher-level information from low-level system events, a CEP component was also developed, and integrated with the system.

### 5.1 TESTING

Ignition is a fairly mature product, and is presumably well-tested and reliable. The application server is distributable, and scalable, and there is no particular limit on the size of the address space. Any performance limitations or errors would likely come from the JMEDS stack from WS4D, or from programming errors in the Ignition DPWS Module, developed using the Ignition SDK.

The system was tested with up to four InicoTech S1000 DPWS devices simultaneously, with no detectable performance shortcomings. The JMEDS stack was tested independently with upwards of 30 devices, and performance problems, such as dropped events or failed service invocations were rare. Occasionally, when large numbers of devices return discovery probe matches simultaneously, some may be lost. Sending a second discovery probe and ignoring duplicate responses was a suitable solution in these cases.

Functionally, the system performs reasonably well. Discovered DPWS devices appear in the OPC UA Address Space, and parameters can be set and read, operations can be invoked, and events can be subscribed to. The typical workflow being tested would be as follows:

1. Launch the Ignition Designer, and create a new Project. Put the Ignition Designer in “Gateway Read/Write” mode in the toolbar.
2. Click the “Discover Devices” button in the DPWS Toolbar.
3. Refresh the OPC Browser and SQL Tag Browsers to view the discovered devices.
4. Create an HMI, such as the one in Figure 30: Create a new window in the Project Browser, drag tags from the SQL Tags Browser onto the window and select a visual component from the context menu, or add components from the component pallets and link their properties to SQL Tags in the Property Editor.
5. Ensure that all events of interest have the “Subscribe” Node set to TRUE.

6. Save and publish the project.
7. Launch the client project from the Ignition Gateway Browser Interface
8. Test setting parameters, invoking operations, and viewing incoming events.

An important part of the testing also included setting parameters and invoking operations via the OPC UA address space, and viewing the output of event messages using a third party OPC UA Client in a local network.

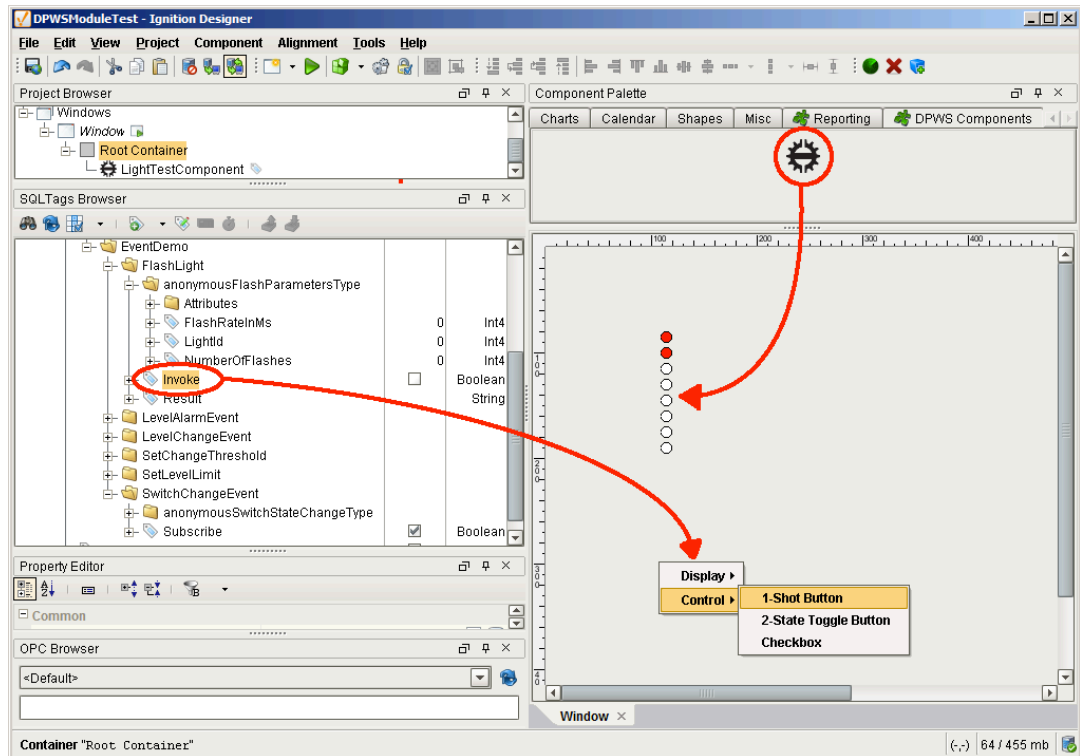


Figure 30: Creating an HMI in the Ignition Designer

Testing of the StreamInsight component, and its Web Service input and output adapters, was limited to functional testing using a single S1000 DPWS-enabled device, and adding components on an existing HMI for viewing the incoming messages from the StreamInsight output adapter.

## 5.2 ASSESSMENT

The system performed well under testing, and could prove quite useful for quickly designing and deploying high-quality HMIs for a set of DPWS devices in a real application. The feasibility of integrating low-level DPWS devices into an OPC UA-based infrastructure was demonstrated satisfactorily.

Also, an approach for integrating a CEP engine into such an architecture was also demonstrated. Although the queries tested were simple, and the implementation did not

use WS-Eventing, this implementation could easily be expanded for a more complex use case.

This system will be used as the base for one of the demonstrators in the European Commission's Framework Package 7 Project, Architecture for Service-Oriented Process Monitoring and Control (IMC-AESOP)[61], which aims to develop a system of systems approach for process monitoring and control based on SOA for very large distributed systems.

### **5.3 NEXT STEPS**

Although the results are promising, several areas for potential improvement have been identified.

As a first version prototype, there is always room for optimization and improvement in reliability. In order for the module to be truly useful, additional tools for managing discovered devices and discovering devices outside local networks would need to be added. Currently, it is possible to conditionally trigger operation invocations based on some arbitrary criteria using the Python scripting functionality of the SQL Tags system. This functionality should be further explored and exploited, and perhaps wrapped in a clearer interface.

In future versions of the Ignition SDK, it may be easier to create the more object-oriented device representation described in Section 3.3.1, rather than the simplified Address Space view used in this implementation, consisting of just Folder Nodes and Data Variable nodes. This could even allow linking objects representing entire services, operations, or events to custom components.

Designing HMIs is a very manual process, that doesn't take advantage of any of the semantic information in the OPC UA address space or WSDL file. Using semantic information linked in a WSDL-S file, many tedious aspects of user interface creation could be automated. For example, a value identified as a tank level or could automatically be given an appropriate representation, or a custom component designed for a specific hosted service, or with multiple input properties, could be automatically linked to the relevant SQL Tags.

The StreamInsight component should be enhanced with a full DPWS client implementation in the Input Adapter, a full WS-Eventing implementation in the Output Adapter, and a web service interface for managing and defining queries.

## 6 CONCLUSION

Various researchers and industry groups have proposed Service Oriented Architecture (SOA) and Event-Driven SOA as solutions for providing interoperability, vendor independence, cross-layer integration, and real-time visibility across all levels of a globally distributed manufacturing enterprise. OPC Unified Architecture (OPC UA) and Devices Profile for Web Services (DPWS) have both been proposed as possible enabling technologies for industrial SOA. Although both specifications are based on a Web-Service communication model, they differ substantially in terms of philosophy, design, completeness, and adoption rate in industry.

The goal of this research was to propose and demonstrate an approach for merging these two web service-based technologies, OPC UA and DPWS, in a way that is applicable in existing industrial applications.

Using Ignition, a commercial HMI/SCADA and MES system with an integrated OPC UA Server, and JMEDS from WS4D.org, an open source DPWS stack, a system was designed and implemented to leverage the complementary strengths of both technologies. A representation of DPWS devices in the OPC UA Address Space was defined, and a plugin module was created for Ignition OPC UA Server to discover DPWS devices, create and maintain representations in the address space, and link them to the physical device. The module supports events and operations. To further prove the concepts of Event-Driven SOA for monitoring and control systems, web service input and output adapters were created for StreamInsight, a commercial CEP Engine from Microsoft, to demonstrate an approach for deducing high-level information from a cloud of low-level events.

This work hopefully contributes something useful to the body of knowledge related to Event-Driven SOA for industrial applications, and helps further the progress toward future event-driven industrial SOA applications, with the inherent modularity, agility, and interoperability envisioned by researchers today.

## 7 REFERENCES

- [1] HAAS, H., AND BROWN, A., "WEB SERVICES GLOSSARY, WORLD WIDE WEB CONSORTIUM (W3C) WORKING GROUP NOTE, FEBRUARY 2004; [HTTP://WWW.W3.ORG/TR/WS-GLOSS/](http://www.w3.org/TR/ws-gloss/)
- [2] GUDGIN, M. ET AL., "SOAP VERSION 1.2 PART 1: MESSAGING FRAMEWORK (SECOND EDITION)", WORLD WIDE WEB CONSORTIUM (W3C) RECOMMENDATION, APRIL 2007; [HTTP://WWW.W3.ORG/TR/SOAP12-PART1/](http://www.w3.org/TR/soap12-part1/)
- [3] MITRA, N. AND LAFON, Y., "SOAP VERSION 1.2 PART 0: PRIMER (SECOND EDITION)", WORLD WIDE WEB CONSORTIUM (W3C) RECOMMENDATION, APRIL 2007; [HTTP://WWW.W3.ORG/TR/SOAP12-PART0/](http://www.w3.org/TR/soap12-part0/)
- [4] GUDGIN, M. ET AL., "SOAP VERSION 1.2 PART 2: ADJUNCTS (SECOND EDITION)", WORLD WIDE WEB CONSORTIUM (W3C) RECOMMENDATION, APRIL 2007; [HTTP://WWW.W3.ORG/TR/SOAP12-PART2/](http://www.w3.org/TR/soap12-part2/)
- [5] DELAMER, I.M., MARTINEZ LASTRA, J.L. AND CAVIA SOTO, M.A. (2007) "AN EVENT-BASED SERVICE-ORIENTED INFRASTRUCTURE FOR RECONFIGURABLE MANUFACTURING SYSTEMS", INT. J. MANUFACTURING RESEARCH, VOL. 2, No. 1, pp.21–50.
- [6] "SOAP-OVER-UDP VERSION 1.1", OASIS STANDARD, JULY 2009; [HTTP://DOCS.OASIS-OPEN.ORG/WS-DD/SOAPOVERUDP/1.1/WSDD-SOAPOVERUDP-1.1-SPEC.HTML](http://docs.oasis-open.org/ws-dd/soapoverudp/1.1/wsdd-soapoverudp-1.1-spec.html)
- [7] JAMMES, F.; SMIT, H.; LASTRA, J.L.M.; DELAMER, I.M.; , "ORCHESTRATION OF SERVICE-ORIENTED MANUFACTURING PROCESSES," EMERGING TECHNOLOGIES AND FACTORY AUTOMATION, 2005. ETFA 2005. 10TH IEEE CONFERENCE ON , VOL.1, NO., PP.8 PP.-624, 19-22 SEPT. 2005
- [8] DELAMER, I.M.; LASTRA, J.L.M.; , "SELF-ORCHESTRATION AND CHOREOGRAPHY: TOWARDS ARCHITECTURE-AGNOSTIC MANUFACTURING SYSTEMS," ADVANCED INFORMATION NETWORKING AND APPLICATIONS, 2006. AINA 2006. 20TH INTERNATIONAL CONFERENCE ON , VOL.2, NO., PP. 5 PP., 18-20 APRIL 2006
- [9] PUTTONEN, J., LOBOV, A., SOTO, M., AND LASTRA, J.L.M. 'A SEMANTIC WEB SERVICES-BASED APPROACH FOR PRODUCTION SYSTEMS CONTROL', ADVANCED ENGINEERING INFORMATICS, VOL.24, No. 3, AUGUST 2010.
- [10] LASTRA, J.L.M.; DELAMER, M.; , "SEMANTIC WEB SERVICES IN FACTORY AUTOMATION: FUNDAMENTAL INSIGHTS AND RESEARCH ROADMAP," INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON , VOL.2, NO.1, PP. 1- 11, FEB. 2006
- [11] LOBOV, A.; LOPEZ, F.U.; HERRERA, V.V.; PUTTONEN, J.; LASTRA, J.; , "SEMANTIC WEB SERVICES FRAMEWORK FOR MANUFACTURING INDUSTRIES," ROBOTICS AND



- BIOMIMETICS, 2008. ROBIO 2008. IEEE INTERNATIONAL CONFERENCE ON , VOL., NO., PP.2104-2108, 22-25 FEB. 2009
- [12] LOBOV, A.; PUTTONEN, J.; HERRERA, V.V.; ANDIAPPAN, R.; LASTRA, J.; , "SERVICE ORIENTED ARCHITECTURE IN DEVELOPING OF LOOSELY-COUPLED MANUFACTURING SYSTEMS," INDUSTRIAL INFORMATICS, 2008. INDIN 2008. 6TH IEEE INTERNATIONAL CONFERENCE ON , VOL., NO., PP.791-796, 13-16 JULY 2008
- [13] DELAMER, I.M.; LASTRA, J.L.M.; , "LOOSELY-COUPLED AUTOMATION SYSTEMS USING DEVICE-LEVEL SOA," *INDUSTRIAL INFORMATICS, 2007 5TH IEEE INTERNATIONAL CONFERENCE ON* , VOL.2, NO., PP.743-748, 23-27 JUNE 2007
- [14] CRISTCOST, WIKIMEDIA COMMONS IMAGE, DECEMBER 2007; [HTTP://EN.WIKIPEDIA.ORG/WIKI/FILE:WSDL\\_11VS20.PNG](http://en.wikipedia.org/wiki/File:WSDL_11vs20.png)
- [15] CHRISTENSEN, E. ET AL., "WEB SERVICES DESCRIPTION LANGUAGE (WSDL) 1.1", WORLD WIDE WEB CONSORTIUM (W3C) NOTE, MARCH 2001; [HTTP://WWW.W3.ORG/TR/WSDL](http://www.w3.org/TR/WSDL)
- [16] AKKIRAJU, R. ET AL., "WEB SERVICE SEMANTICS - WSDL-S", WORLD WIDE WEB CONSORTIUM (W3C) SUBMISSION, NOVEMBER 2005; [HTTP://WWW.W3.ORG/SUBMISSION/WSDL-S/](http://www.w3.org/submission/WSDL-S/)
- [17] VEDAMUTHU, A.S. ET AL., "WEB SERVICES POLICY 1.5 – FRAMEWORK", WORLD WIDE WEB CONSORTIUM (W3C) RECOMMENDATION, SEPTEMBER 2007; [HTTP://WWW.W3.ORG/TR/WS-POLICY/](http://www.w3.org/TR/WS-POLICY/)
- [18] "DEVICES PROFILE FOR WEB SERVICES (DPWS)", OASIS STANDARD, JULY 2009; [HTTP://DOCS.OASIS-OPEN.ORG/WS-DD/NS/DPWS/2009/01](http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01)
- [19] ALEXANDER, J. ET AL., "WEB SERVICES TRANSFER (WS-TRANSFER)", WORLD WIDE WEB CONSORTIUM (W3C) SUBMISSION, SEPTEMBER 2006; [HTTP://WWW.W3.ORG/SUBMISSION/WS-TRANSFER/](http://www.w3.org/submission/WS-TRANSFER/)
- [20] DAVES, D. ET AL., "WEB SERVICES METADATA EXCHANGE (WS-METADATAEXCHANGE)", WORLD WIDE WEB CONSORTIUM (W3C) PROPOSED RECOMMENDATION, SEPTEMBER 2011; [HTTP://WWW.W3.ORG/TR/WS-METADATA-EXCHANGE/](http://www.w3.org/TR/WS-METADATA-EXCHANGE/)
- [21] BAJAJ, S. ET AL., "WEB SERVICES POLICY 1.2 - ATTACHMENT (WS-POLICYATTACHMENT)", WORLD WIDE WEB CONSORTIUM (W3C) SUBMISSION, APRIL 2006; [HTTP://WWW.W3.ORG/SUBMISSION/WS-POLICYATTACHMENT/](http://www.w3.org/submission/WS-POLICYATTACHMENT/)
- [22] BOX, D. ET AL., "WEB SERVICES EVENTING (WS-EVENTING)", WORLD WIDE WEB CONSORTIUM (W3C) SUBMISSION, MARCH 2006; [HTTP://WWW.W3.ORG/SUBMISSION/WS-EVENTING/](http://www.w3.org/submission/WS-EVENTING/)
- [23] BOX, D. ET AL., "WEB SERVICES ADDRESSING (WS-ADDRESSING)", WORLD WIDE WEB CONSORTIUM (W3C) SUBMISSION, AUGUST 2004; [HTTP://WWW.W3.ORG/SUBMISSION/WS-ADDRESSING/](http://www.w3.org/submission/WS-ADDRESSING/)
- [24] "WEB SERVICES DYNAMIC DISCOVERY (WS-DISCOVERY)", OASIS STANDARD, JULY 2009; [HTTP://DOCS.OASIS-OPEN.ORG/WS-DD/NS/DISCOVERY/2009/01](http://docs.oasis-open.org/ws-dd/ns/discovery/2009/01)

- [25] "ADDITIONAL WS-DISCOVERY FUNCTIONALITY", MSDN LIBRARY, JULY 2009; [HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/BB736556\(V=VS.85\).ASPX](http://msdn.microsoft.com/en-us/library/bb736556(v=VS.85).aspx)
- [26] "ABOUT WEB SERVICES ON DEVICES", MSDN LIBRARY, JULY 2009; [HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/AA385800\(V=VS.85\).ASPX](http://msdn.microsoft.com/en-us/library/aa385800(v=VS.85).aspx)
- [27] "WS4D-UDPWS: DPWS FOR HIGHLY RESOURCE-CONSTRAINED DEVICES", GOOGLE CODE OPEN SOURCE PROJECT, AUGUST 2010; [HTTP://CODE.GOOGLE.COM/P/UDPWS/](http://code.google.com/p/udpws/)
- [28] "SOA4D SERVICE-ORIENTED ARCHITECTURE FOR DEVICES", OPEN SOURCE PROJECT, JUNE 2007; [HTTPS://FORGE.SOA4D.ORG/](https://forge.soa4d.org/)
- [29] A. MENSCH AND S. ROUGES, DPWS CORE VERSION 2.1 USER GUIDE, APRIL 2009; [HTTPS://FORGE.SOA4D.ORG/DOCMAN/VIEW.PHP/8/45/DPWSCORE+USER+GUIDE.PDF](https://forge.soa4d.org/docman/view.php/8/45/DPWSCore+User+Guide.pdf)
- [30] ITEA SIRENA CONSORTIUM; 2003-2005; [HTTP://WWW.SIRENA-ITEA.ORG/](http://www.sirena-itea.org/)
- [31] ITEA SODA CONSORTIUM; 2006-2008; [HTTP://WWW.SODA-ITEA.ORG](http://www.soda-itea.org)
- [32] FP6 SOCRADES CONSORTIUM; 2007-2009; [HTTP://WWW.SOCRADES.EU](http://www.socrales.eu)
- [33] FP7 IMC-AESOP CONSORTIUM, D1.1A: STATE-OF-THE-ART REPORT; FEBRUARY 2011.
- [34] ARSANJANI, A. ET AL., "SOMA: A METHOD FOR DEVELOPING SERVICE-ORIENTED SOLUTIONS," IBM SYSTEMS JOURNAL , VOL.47, NO.3, PP.377-396, 2008
- [35] MACKENZIE, C.M., ET AL., REFERENCE MODEL FOR SERVICE ORIENTED ARCHITECTURE 1.0, OASIS STANDARD, OCTOBER 2006; [HTTP://DOCS.OASIS-OPEN.ORG/SOA-RM/V1.0/SOA-RM.PDF](http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf)
- [36] "DEFINITION OF SOA", THE OPEN GROUP, [HTTP://WWW.OPENGROUP.ORG/SOA/SOA/DEF.HTM](http://www.opengroup.org/soa/soa/def.htm)
- [37] JAMMES, F.; SMIT, H.; , "SERVICE-ORIENTED PARADIGMS IN INDUSTRIAL AUTOMATION," INDUSTRIAL INFORMATICS, IEEE TRANSACTIONS ON , VOL.1, NO.1, PP. 62- 70, FEB. 2005
- [38] MAHNKE, W., LEITNER, S., AND DAMM, M., "OPC UNIFIED ARCHITECTURE". BERLIN, GERMANY: SPRINGER, 2009.
- [39] "OPC UNIFIED ARCHITECTURE SPECIFICATION PART 3: ADDRESS SPACE MODEL", OPC FOUNDATION, RELEASE 1.01, FEB 6, 2009.
- [40] MCGUINNESS, D.L., AND VAN HARMELEN, F., "OWL WEB ONTOLOGY LANGUAGE OVERVIEW", WORLD WIDE WEB CONSORTIUM (W3C) RECOMMENDATION, FEBRUARY 2004; [HTTP://WWW.W3.ORG/TR/OWL-FEATURES/](http://www.w3.org/TR/owl-features/)
- [41] SMITH, M.C., WELTY, C., AND MCGUINNESS, D.L., "OWL WEB ONTOLOGY LANGUAGE GUIDE", WORLD WIDE WEB CONSORTIUM (W3C) RECOMMENDATION, FEBRUARY 2004; [HTTP://WWW.W3.ORG/TR/OWL-GUIDE/](http://www.w3.org/TR/owl-guide/)
- [42] UNIFIED AUTOMATION, UAEXPERT OPC UA CLIENT APPLICATION, NOVEMBER 2010; [HTTP://WWW.UNIFIED-AUTOMATION.COM/UAEXPERT.HTM](http://www.unified-automation.com/uaexpert.htm)

- [43] INDUCTIVE AUTOMATION, IGNITION OPC-UA SERVER;  
[HTTP://WWW.INDUCTIVEAUTOMATION.COM/SCADA-SOFTWARE/OPC-UA-SERVER](http://www.inductiveautomation.com/scada-software/opc-ua-server)
- [44] "OPC UNIFIED ARCHITECTURE SPECIFICATION PART 4: SERVICES", OPC FOUNDATION STANDARD, FEBRUARY 2009.
- [45] INDUCTIVE AUTOMATION, IGNITION SERVER;  
[HTTP://WWW.INDUCTIVEAUTOMATION.COM/SCADA-SOFTWARE](http://www.inductiveautomation.com/scada-software)
- [46] CANDIDO, G.; JAMMES, F.; DE OLIVEIRA, J.B.; COLOMBO, A.W.; , "SOA AT DEVICE LEVEL IN THE INDUSTRIAL DOMAIN: ASSESSMENT OF OPC UA AND DPWS SPECIFICATIONS," INDUSTRIAL INFORMATICS (INDIN), 2010 8TH IEEE INTERNATIONAL CONFERENCE ON , VOL., NO., PP.598-603, 13-16 JULY 2010
- [47] BONY, B.; HARNISCHFEGER M.; JAMMES, F.; , "CONVERGENCE OF OPC UA AND DPWS WITH A CROSS-DOMAIN DATA MODEL," IN INDIN'11 – ANNUAL CONFERENCE OF THE IEEE INDUSTRIAL ELECTRONICS SOCIETY, 2011.
- [48] PLCOPEN OPC UA INFORMATION MODEL 1.00 SPECIFICATION
- [49] UA PART DI: OPC UA COMPANION SPECIFICATION FOR DEVICES, VERSION 1.00
- [50] IZAGUIRRE MONTEMAYOR, J.A.G.; LOBOV, A; LASTRA, J.L.M.; , "OPC-UA AND DPWS INTEROPERABILITY FOR FACTORY FLOOR MONITORING USING COMPLEX EVENT PROCESSING," INDUSTRIAL INFORMATICS (INDIN), 2011 9TH IEEE INTERNATIONAL CONFERENCE ON , VOL., NO., PP.205-211, 26-29 JULY 2011
- [51] INVOKING MfSvcUtil FROM THE COMMAND LINE. MSDN LIBRARY ARTICLE.  
[HTTP://MSDN.MICROSOFT.COM/EN-US/LIBRARY/EE435401.ASPX](http://msdn.microsoft.com/en-us/library/ee435401.aspx)
- [52] LUO, M. ET AL. EVENT-DRIVEN SERVICE ORIENTED FRAMEWORK FOR INTEGRATIVE SERVICEABILITY MANAGEMENT OF NETWORKED MANUFACTURING SYSTEMS. IEEE/ASME INTERNATIONAL CONFERENCE ON ADVANCED INTELLIGENT MECHATRONICS. SINGAPORE, JULY 2009.
- [53] MICHELSON, B. "EVENT-DRIVEN ARCHITECTURE OVERVIEW", PATRICIA SEYBOLD GROUP, BOSTON, FEBRUARY 2006.
- [54] LUCKHAM, D.C., "WHAT'S THE DIFFERENCE BETWEEN ESP AND CEP?", AVAILABLE ONLINE, 2006.
- [55] "OPC UNIFIED ARCHITECTURE PART 6: MAPPINGS", OPC FOUNDATION STANDARD, FEBRUARY 2009.
- [56] "OPC UNIFIED ARCHITECTURE PART 2: SECURITY MODEL", OPC FOUNDATION STANDARD, FEBRUARY 2009.
- [57] "OPC UNIFIED ARCHITECTURE PART 2: SECURITY MODEL", OPC FOUNDATION STANDARD, FEBRUARY 2009.
- [58] WS4D.ORG, JAVA MULTI-EDITION DPWS STACK:, [HTTP://WS4D.E-TECHNIK.UNI-ROSTOCK.DE/JMEDS/](http://ws4d.e-technik.uni-rostock.de/jmeds/)
- [59] INICOTECH: S1000 SMART REMOTE TERMINAL UNIT,  
[HTTP://WWW.INICOTECH.COM/S1000\\_OVERVIEW.HTML](http://www.inicotech.com/s1000_overview.html)

- [60] INDUCTIVE AUTOMATION, IGNITION PROGRAMMER'S GUIDE v7.2.8 (AVAILABLE WITH SDK DOWNLOAD), [HTTP://FILES.INDUCTIVEAUTOMATION.COM/RELEASE/BUILD7.2.8/08-04-2011\\_16\\_13/IGNITION-MODULESDK-7.2.8.ZIP](http://files.inductiveautomation.com/release/build7.2.8/08-04-2011_16_13/IGNITION-MODULESDK-7.2.8.ZIP)
- [61] ARCHITECTURE FOR SERVICE-ORIENTED PROCESS MONITORING AND CONTROL (IMC-AESOP). [HTTP://WWW.IMC-AESSOP.EU](http://www.imc-aessop.eu)
- [62] KING, P. "SCADA SYSTEMS: LOOKING AHEAD", CONTROL MICROSYSTEMS WHITEPAPER, AUGUST 2005.

## Appendix A: OPC UA Eventing Mechanism

OPC is a specification that allows interoperability between applications and field devices via COM/DCOM. Following the current trend toward interoperability using standard network protocols, the OPC Foundation has adopted a Web Service communication model for its latest specification. OPC servers, commonly known as "classic OPC servers", can be wrapped and unified with OPC-UA. Such wrapping provides a set of services and operations which, in contrast to DPWS, are pre-defined in the specification.

OPC UA's event model defines a general purpose eventing system. Events represent transient occurrences, such as configuration changes, value changes, and errors. Event Notifications report occurrences of events. Events themselves are not directly visible in the AddressSpace, but Objects and views can be used to subscribe to Events. Clients subscribe to Nodes and receive Notifications of Event Occurrences using the MonitoredItem and Subscription Service sets.

An OPC UA server that supports eventing exposes one or more EventNotifier Nodes. The Server Object, defined in the OPC UA Part 5: Information Model, is an example of an EventNotifier. The Events generated by the server are available via the Server Object. Events can also be exposed through any node in the AddressSpace identified by the EventNotifier attribute, which indicates if the Node can be used to subscribe to Events or read / write historic Events. The server determines which events are provided by which node.

Each event is of a specific EventType. The OPC UA Specification defines a BaseEventType, and many others that derive from this type, such as SystemEventType, AuditCreateSessionEventType, AuditUpdateEventType, DeviceFailureEventType, and ModelChangeEvents. EventTypes do not have a special NodeClass, but are instead represented as ObjectTypes in the AddressSpace. EventTypes can be either abstract or not. Abstract event types are never instantiated in the AddressSpace, and their occurrence is only exposed through Subscriptions. Non-abstract event types can be visible in the AddressSpace, and are also accessible through Event Notification mechanisms.

Events are categorized by subtyping existing EventTypes without extending them by defining additional properties or changing the inherited semantics. For example, DeviceFailureEventType could be subtyped into TransmitterFailureEventType and ComputerFailureEventType. The following reference types are used for organizing Events and Event Sources in the OPC UA Address Space [39]:

Table 13: Event Source Reference Types

Reference Type	Semantics
GeneratesEvent	Indicates EventTypes that ObjectTypes and VariableTypes generate, or Methods may generate on Method calls.
AlwaysGeneratesEvent	Indicates the EventTypes that Methods must generate on each Method call.
HasEventSource	Used for categorization and organization of Event sources. Any Object that is the source of Event Subscriptions can reference as an Event Source any Node of any NodeClass that can generate event notifications via a subscription.
HasNotifier	Used for hierarchical organization of Event Notifiers. Objects or Views that are a source of event subscriptions can specify any other objects or views that are a source of event subscriptions as notifiers. If the target node generates an event, the event is also provided in the source node.

Categorization of Event Sources using HasEventSource and HasNotifier ReferenceTypes is shown in Figure 31.

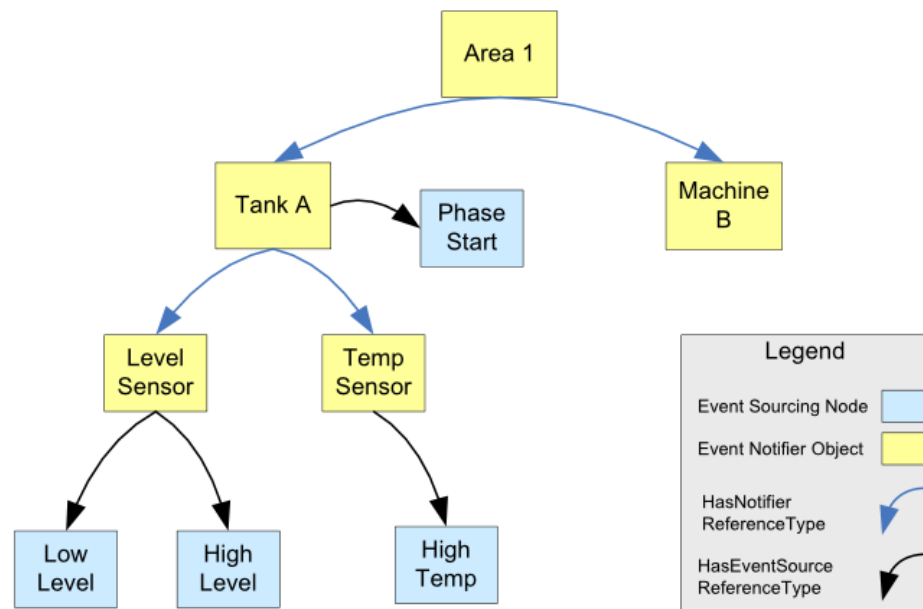


Figure 31: Event Reference Example [39]

### MonitoredItem

A Subscription is an endpoint in the Server that publishes notifications to Clients. Clients control the publishing rate by sending Publish Requests.

MonitoredItems are created on the Server by the Client using the MonitoredItem Service Set. A MonitoredItem monitors Variables, Attributes, and Event Notifiers, and generate a Notification when they detect a data or status change, or an event/alarm occurrence. The Notification is then transferred to the Client by a Subscription. This model is shown in Figure 32:

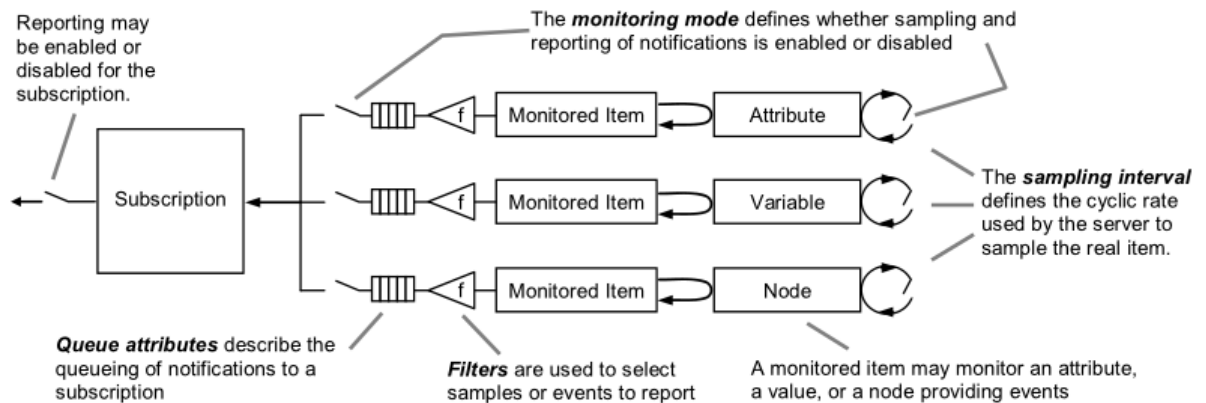


Figure 32: Monitored Item Model [44]

Each MonitoredItem identifies the item to monitor, and the Subscription to use to periodically publish Notifications. The MonitoredItem also specifies the rate at which the item is to be sampled, and the filter criteria for generating Notifications in the case of Variables and EventNotifiers. Filter Criteria for Attributes are indicated by their Attribute Definitions.

The sample rate for the Monitored Item may be faster than the publishing rate of the Subscription. For this reason, the Monitored Item can be used to Queue all Notifications, or just the latest Notification for transfer by the Subscription. MonitoredItem services also define a monitoring mode, configured to disable sampling and reporting, enable sampling only, or enable both sampling and reporting. Each sample is evaluated to determine if a Notification should be generated. If yes, the notification is queued. If reporting is enabled, the queue is made available to the Subscription for Transfer.

### Subscriptions

The Subscription service set is used to create and maintain subscriptions. Subscriptions periodically publish NotificationMessages for the set of MonitoredItems assigned to them. The NotificationMessage contains a common header, followed by a series of Notifications. The format is specific to the node type being monitored.

NotificationMessages are sent to the Client in response to Publish requests. Publish requests are queued in the Session as they are received, and one is dequeued and processed by a Subscription related to the Session each publishing cycle if there are Notifications to report. If there are Notifications available, but no Publish requests, the

Server enters a wait state and sends a NotificationMessage as soon as the Publish request is received. NotificationMessages are uniquely identified by sequence numbers.

Subscriptions have a keep-alive counter, which tracks how many publishing cycles have passed without having a Notification to report. If the keep-alive counter reaches some configured maximum amount specified when the Subscription was created, a Publish request is dequeued, and a keep-alive message informs the Client that the Subscription is still active. A NotificationMessage contains one or more Notifications. Subscriptions have a configured lifetime, which clients periodically renew. If unrenewed, the subscription is closed, and all MonitoredItems assigned to the Subscription are deleted.



## Appendix B: Instructions for Ignition Modules

### INSTALLING AND SETTING UP IGNITION AND THE DPWS DRIVER

- 0) Set up network interface (ip address 192.168.2.XXX, Subnet Mask 255.255.0.0)
- 1) Install Ignition v. 7.2.8, from the archived downloads section of  
<http://www.inductiveautomation.com/downloads/ignition/archive>
- 2) Register for a free developer license:  
<https://www.inductiveautomation.com/developers/register>
- 3) Go to Ignition Gateway configuration page in browser (admin/password)  
<http://localhost:8088/main/web/config/>
- 4) Receive CD Key in email
- 5) Put the Gateway in Developer Mode  
 In sidebar:  
 System > Licensing > "Purchase or activate this Ignition Gateway"  
 Continue on to activation, then enter the CD Key
- 6) Install the DPWS Module in the Gateway configuration page:  
 Configuration > Modules > Install or Upgrade a Module  
 Choose DpwsDriver-module-unsigned.modl
- 7) Add The OPC UA Device in the Gateway Configuration  
 OPC-UA > Devices > Add a Device  
 Choose DPWS>DPWS Driver  
 Click Next  
 Enter a name, such as "DPWSDevice"  
 Choose the Network Interface, such as "eth3"
- 8) Launch Designer, create new project  
 In Toolbar:  
 Put Designer in full Read/Write gateway communication mode  
 DPWS Scan (radar icon)  
 Delete discovered devices (red X)

**NOTE:** On some Laptop computers, if you unplug the network cable, the network adapter is powered down, and errors may occur in the module. It is best to plug an ethernet cable into the port, before starting the Ignition Gateway Service.

## BUILDING MODULES

These modules were written for Ignition Version 7.2.8. They have not been tested with the latest version, and will likely not work, due to changes in the SDK in 7.3. To Download Ignition 7.2.8, go to [www.inductiveautomation.com](http://www.inductiveautomation.com), and visit the Archived Downloads section.

- Install and launch eclipse.
- Choose the folder IgnitionModuleSDK-7.2.8 as the workspace.
- Right-click in the Package Explorer (left column) and choose "Import."
- Choose General > Existing Projects into Workspace, click Next
- Import all the projects from "aesop\_projects."

To compile a module, right click the appropriate build script (build-dpwsdriver.xml or build-cepoutputinterface.xml) in the "Build" project, and select Run As > Ant Build.

## DPWS DRIVER PROJECT OVERVIEW

The DPWS Driver module consists of four projects

- **DPWSDriver\_Client** - A Sample custom Visual component
- **DPWSDriver\_Common** - The RPC Interface description
- **DPWSDriver\_Designer** - A tab for the custom component, and toolbar buttons for DPWS Scan and Erasing discovered devices
- **DPWSDriver\_Gateway** - The back end, including Tag Provider, OPC UA Node Map Driver, DPWS Client, and configuration information.

This driver module discovers DPWS Devices, maps the events and operations into the Ignition OPC UA Server's address space, and exposes each of the nodes as SQLTags.

## CEP OUTPUT ADAPTER SINK PROJECT OVERVIEW

The CEP Output Interface Module consists of just one project: **CEPOutputInterface**. This module exposes a Web Service in Ignition for the StreamInsight Output Adapter to invoke when events are generated as a result of a CEP Query.

When this project is re-built, and the module is re-installed, the Ignition Gateway must be restarted from Windows Control Panel > Administrative Tools > Services (run as administrator)

When adding the device in the Gateway Configuration browser interface, you must specify a service name, and a port number. Changing either of these requires that you restart the Ignition Server. The defaults are "OutputAdapterSink" and 8099.

## BUILDING THE STREAMINSIGHT PROJECT FOR VISUAL STUDIO (C#)

Setup:

- Install Microsoft Visual C# 2010 Express
- Install StreamInsight (See Note below)
- Run Microsoft Visual C# 2010 Express as administrator.
- Open StreamInsightCEP\AesopCepBeta\AesopCepBeta.sln
- Open Program.cs in the project AesopCepBeta, and change the input adapter URL to match your local IP address, and the output adapter URL to match the address of the CEPOutputAdapterInterface Ignition Module. Using the default settings in the ignition module, this should be "http://192.168.X.XXX:8099/OutputAdapterSink".
- Run the Program, post SOAP messages to the input URL, and watch for Output events in Ignition.
- 

**NOTE:** Installing StreamInsight creates a windows user group called "StreamInsightUsers." WCF needs the appropriate URL permissions to create the Web Service endpoints, so if you run the application as administrator, you will have to add the administrator to the StreamInsightUsers group in "User Account > Advanced" settings.

Alternatively, run the following commands in an admin shell:

```
netsh http add urlacl
url=http://192.168.2.123:8000/CepInputAdapter/ChangeReporting
user=domain\username
```

You will give your StreamInsight server a name when you install. I used "CEPServer." Whichever name you give needs to appear at Line 34 of Program.cs

```
using(Server server = Server.create("CEPServer");
```

The input event format, output event format, and LINQ query will have to be changed for demonstrating some real application. Once you're familiar with LINQ queries, and as long as you make sure that the input interface descriptions are consistent with the external message source or recipient, this should be pretty straightforward.

The `IPointInputAdapter` interface and `WcfPointInputAdapter` implementation correspond to the `LevelChange` and `SwitchChange` input and output events in the S1000 project `cepInputAdapter_test.xml`

The `IOutputAdapterSink` interface and `CepPointOutputAdapter` implementation correspond to the service and action defined in `org.fast.cep.CEPHostedService` in the `CEPOutputInterface` Ignition Module. The interface description is equivalent to the `CepOutput Output Message` defined in `cepInputAdapter_test.xml`.

When running `cepInputAdapter_test.xml`, two global variables define the input adapter, and Ignition CEP Sink service addresses:

```
service_address
"http://192.168.2.123:8000/CepInputAdapter/ChangeReporting"
cep_sink_address http://192.168.2.123:8099/OutputAdapterSink
```

These should be changed to reflect the address of the computer that the services are running on.

The way the test program is written, the first four digital inputs send messages to the input adapter when the state changes, and the last four digital inputs send messages to the Ignition CEP Sink Service. The `StreamInsight` query sends a message with `ID=12` and `State=true` when `Switch 1` and `Switch 2` both change from `false` to `true` within a one second window. A more sophisticated query can be defined after reading up a bit on `LINQ`.

The Visual Studio Solution contains two projects:

**AesopCepBeta** - main project where the query is defined, and the input and output adapters are configured.

**WsIOAdapters** - The Web Service Input and output adapters. The following files are of interest:

- `CepOutputAdapterFactory.cs`
  - Creates an output adapter for a specific event type. Right now, only point events are supported.
- `CepPointOutputAdapter.cs`
  - Creates the web service client, and handles receiving output from the queries, and sending output through the Web Service Client
- `ClientAdapter.cs`
  - A facade over a WCF exposed client adapter proxy. Handles retry logic.
- `ClientOutputAdapterSink.cs`
  - Facade over the output adapter proxy.
  - Handles preparing the parameters for the output message.
- `ClientPointInputAdapter.cs`

- In Program.cs, there is a method called "ProduceEvents." This can be used to simulate input events. This is the input adapter for sending the messages to create simulated events.
- IOOutputAdapterSink.cs
  - This is where the output service and message format are defined.
- IPointInputAdapter.cs
  - This is where the input service interface is defined.
- WcfInputAdapterFactory.cs
  - Same as CepOutputAdapterFactory, but for the input adapter
- WcfPointEvent.cs
  - An event definition, used only for the ClientPointInputAdapter for simulating events.
- WcfPointInputAdapter.cs
  - The implementation of the IPointInputAdapter service interface.