FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA

# DPWS MIDDLEWARE TO SUPPORT AGENT-BASED MANUFACTURING CONTROL AND SIMULATION

POR

**RUI RODRIGUES MILAGAIA**

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para obtenção do grau de Mestre em Engenharia Electrotécnica e de Computadores

**ORIENTADOR:** PROF. JOSÉ ANTÓNIO BARATA OLIVEIRA

**LISBOA**

**2009**

To my father, mother and sister

# Acknowledgements

I would like to thank all the people that made this work possible. The invaluable coordination provided by Prof. José António Barata de Oliveira, Dr. Armando Walter Colombo, Dr. Ronald Shoop and Dr. Ralf Neubert. Eng. Axel Bepperling that led me closely for the first 6 months and let me follow some dreams while keeping my feet on the ground. Daniel Cachapa Vieira, old friend that helped me through a lot of difficult situations, personal and work wise. Prof. Martin Feike for his precious help with the 3D Simulation Software QUEST. All the Schneider Electric Seligenstadt HUB staff that were always available whenever help was needed. My friends and colleagues in Portugal with whom I shared countless ideas and worries, Pedro Miguel Faleiro and José Eduardo Cerqueira dos Santos. And my family and girlfriend that, although could not understand what I was talking about, always listened.

# Sumário

O grande desafio que se coloca hoje em dia aos sistemas de manufactura é o desenvolvimento de soluções altamente reconfiguráveis verdadeiramente distribuídas. A tendência actual é cada vez mais os sistemas de manufactura serem construídos através de componentes autónomos, inteligentes e distribuídos que irão suportar as funcionalidades de reconfigurabilidade e adaptabilidade. Os paradigmas que se apresentam como mais promissores para a implementação deste tipo de sistemas são os multiagentes e as arquitecturas de serviços (SOA – Service Oriented Architecture), nomeadamente através da implementação DPWS – Device Profile for Web Services, que se destina essencialmente a dispositivos.

Uma das limitações importantes nos sistemas de multi-agentes mais utilizados hoje em dia é o facto do sistema que gere os agentes não ser totalmente distribuído. Na verdade, uma falha no agente responsável pelo registo dos agentes coloca imediatamente em causa o funcionamento do sistema. O DPWS, por seu lado, não apresenta esta limitação, uma vez que o sistema de gestão dos serviços presentes é completamente distribuído. No entanto o DPWS não suporta tão eficientemente as noções de autonomia presentes nos agentes.

A possibilidade de tornar sistemas de multi-agentes verdadeiramente distribuídos juntando ambas as abordagens levou à elaboração desta tese. Foi desenvolvida uma camada middleware que permite aos agentes utilizarem as funcionalidades DPWS para atingiram o objectivo proposto. Esta camada middleware interliga agentes, bases de dados, hardware, simuladores, aplicações de manutenção, correcção de erros e gestão de produção, etc. Qualquer entidade que participe num sistema de produção pode ter uma interface DPWS. Para provar o conceito foi desenvolvido um modelo 3D de um sistema de manufactura com transportadores que é controlado por agentes que comunicam através de DPWS.

**Palavras-chave**: Arquitectura Orientada aos Serviços, Sistemas de Multi-Agentes, Device Profile for Web Services (DPWS), Sistemas de Manufactura Reconfiguráveis, Manufactura Ágil.

# Abstract

In present manufacturing systems, the current challenge is the development of highly reconfigurable, truly distributed solutions. The tendency is to build manufacturing systems with autonomous, intelligent and distributed components that will support reconfiguration and adaptability. The most promising paradigms for the implementation of such systems are multi-agents and service oriented architectures (SOA), mainly over the DPWS (Device Profile for Web Services) implementation which was aimed at devices.

An important limitation of most current multi-agent systems is that the management system is not totally distributed. Failure in the agent responsible for the registry can overthrow the entire system. DPWS does not have this limitation, since the management system is totally distributed. However, DPWS does not support agent autonomy notions as efficiently.

The possibility of creating a truly distributed multi-agent system by linking both approaches led to this thesis. A Middleware layer was developed that enables agents to benefit from DPWS functionalities in order to reach the proposed goal. This middleware layer joins agents, databases, hardware, simulators, human interface applications such as production system management, error correction and maintenance, etc. To prove this concept a 3D model of an agent controlled manufacturing system with transporters augmented with DPWS communication interfaces was developed.

**Key-Words**: Service Oriented Architecture (SOA), Multi Agent System (MAS), Device Profile for Web Services (DPWS), Reconfigurable Manufacturing System, Agile Manufacturing.

# Glossary of Abbreviations

| | |
|---|---|
| **ACL** | **A**gent **C**ommunication **L**anguage |
| **BDI** | **B**elief **D**esire **I**ntention |
| **BMS** | **B**ionic **M**anufacturing **S**ystems |
| **DCOM** | **D**istributed **C**omponent **O**bject **M**odel |
| **DPWS** | **D**evice **P**rofile for **W**eb **S**ervices |
| **EAS** | **E**volvable **A**ssembly **S**ystems |
| **EPS** | **E**volvable **P**roduction **S**ystems |
| **FIPA** | The **F**oundation for **I**ntelligent **P**hysical **A**gents |
| **HMS** | **H**olonic **M**anufacturing **S**ystem |
| **MAS** | **M**ulti **A**gent **S**ystem |
| **OWL** | **W**eb **O**ntology **L**anguage |
| **RMS** | **R**econfigurable **M**anufacturing **S**ystem |
| **SCL** | **S**imulation **C**ontrol **L**anguage |
| **SOA** | **S**ervice **O**riented **A**rchitecture |
| **UUID** | **U**niversally **U**nique **ID**entifier |
| **WSDL** | **W**eb **S**ervice **D**escription **L**anguage |
| **XML** | e**X**tensible **M**arkup **L**anguage |

# Table of Contents

# Table of Figures

# Table of Tables

# 1 Introduction

In present manufacturing systems, the current challenge is the development of highly reconfigurable, truly distributed solutions. The tendency is to build manufacturing systems with autonomous, intelligent and distributed components that will support reconfiguration and adaptability. The most promising paradigms for the implementation of such systems are MAS (Multi Agent Systems) and SOA (Service Oriented Architectures), mainly over the DPWS (Device Profile for Web Services) implementation which was aimed at devices. The DPWS Stack is a stack of protocols aimed at communication through a SOA for devices. It was created in the context of the European project SIRENA [1] led by Schneider Electric.

An important limitation of most current multi-agent systems is that the management system is not totally distributed. Failure in the agent responsible for the registry can overthrow the entire system. DPWS does not have this limitation, since the management system is totally distributed. However, DPWS does not support agent autonomy notions as efficiently.

By merging both the SOA paradigm with the MAS paradigm we achieve a truly distributed, autonomous and interoperable multi-agent system. The SOA paradigm provides autonomy, interoperability and a distributed environment. These characteristics are applied to devices with SOA frameworks, such as the DPWS protocol stack, exposing their services to agents, human interfaces or PLCs. Allied to a multi-agent framework, a SOA multi-agent system is possible. With such an alliance, the communication in the manufacturing system can be completely integrated from the device level to the business-level with agents in the loop. With such a framework it can be possible to find, communicating in the same manner, sensors, actuators, manufacture level agents, business level agents, simulators, databases, human interface applications amongst many others.

With the main features of SOA being the desired characteristics of MAS by definition, building a communication interface based on a SOA framework, such as DPWS, seems to be a promising step in the MAS paradigm. This thesis, developed in the context of the European project SOCRADES [2], proposes an architecture that enables the DPWS Stack to be effectively used by agents and all other entities that participate in the manufacturing system from the device level to the business-level.

To enable such features as the possibility of creating a truly distributed multi-agent system by linking both, SOA and MAS, the development of a Middleware layer between the

DPWS Stack and the agent is proposed. This layer would enable agents, and any other entity, to benefit from DPWS functionalities and finally be able to achieve total autonomy depicted in its definition.

The proposed middleware layer will enable seamless communication between agents, databases, hardware, simulators, human interface applications such as production system management, error correction and maintenance, etc.

To prove this concept a 3D model of an agent controlled manufacturing system with transporters, augmented with DPWS communication interfaces, will be developed.

## 1.1 THESIS OUTLINE

This thesis is organized in five chapters: Introduction, State of the art & Basic Concepts, DPWS Middleware Architecture, Case Study and Conclusion and Future Work.

The current **Chapter 1** introduces the problem and briefly describes the outline of this work.

**Chapter 2** on State of the Art & Basic Concepts introduces the basic concepts in this work and presents the current state of the art in manufacturing paradigms and in recent work related to the theme of this thesis.

**Chapter 3** on the DPWS Middleware Architecture presents the solution found to the problem and fully describes its architecture.

**Chapter 4** on the Case Study presents a system where the proposed approach was used and how it was implemented.

**Chapter 5** on Conclusions and Future Work discusses the main results of this work and suggests direction for future research and implementation on the problem.

# 2 State of the Art & Basic Concepts

In today's globalized market, modern enterprises have to adopt innovative business methodologies if they are to remain high end competitors. Higher flexibility and agility are required to achieve these goals. In the past few years developments and achievements in the Information Technology (IT) field have provided the manufacturing world with tools that make the implementation of old concepts possible. These tools are the grounds for new paradigm shifts in the industry such as Bionic Manufacturing Systems (BMS) [3], Holonic Manufacturing Systems (HMS) [4; 5; 6; 7], Reconfigurable Manufacturing Systems (RMS) [8; 9], Evolvable Assembly Systems (EAS) [10; 11; 12; 13; 14], and Evolvable Production Systems (EPS) [15; 16; 17].

One of the main obstacles to the realization of these new methods is usually the conservative stance of industry. Due to years of proven methods, industry is reluctant to embark with new paradigms. The argument "existing approaches are sufficient" given by conservative industrial sectors is quite common. Although that stance worked until now, current market conditions require a lot more dynamism from industry thus forcing it to change. In the next few years we will bear witness to radical changes in the manufacturing world to accommodate current market needs.

Another great obstacle to the adoption of new paradigms is integration with legacy systems. Existing technology has failed to provide a complete solution that can operate independently of platform and format. An intermediate step, in which legacy and new paradigm systems coexist, has to be made possible. To this purpose, new paradigms, such as the emergent Web Services paradigm and research in Service Oriented Architectures (SOA), have proven to be a strong possibility. These paradigms are supported by platform agnostic technologies, bringing the industry a step forward in shop floor, enterprise and business integration.

Recent paradigms such as Reconfigurable Manufacturing Systems (RMS) and Evolvable Production Systems (EPS) foresee usage of modular intelligent devices to enable quick reconfiguration, instead of reprogramming.

These new paradigms bring higher flexibility and agility to the shop floor but all comes at a cost. The sophisticated systems that can provide industry with the needed modernization are inherently complex and dynamic. Some of these systems, like agent based systems, have emergent behavior failing to provide total predictability over the production lifecycle. The unpredictability that is usually attributed to emergence scares industry and thus delays the application of such systems.

A significant number of research projects around these paradigms are currently active worldwide. Among academic proposals one may mention: ADACOR [18] – Holonic approach to manufacturing control, ABAS [19] – Agent based approach that models assembly operations, COBASA [20] – Multi-agent based approach for improved shop floor agility and re-configurability, iShopFloor [21] – Agent based approach for a plug and play operational environment over large networks, etc.

There are also a number of European projects which address the issue of reconfigurability, evolvability and agility: EUPASS [22] – development of evolvable micro-assembly systems, SODA [23] – creation of a service oriented ecosystem based on the DPWS framework developed under the SIRENA [1] project, SOCRADES [2] – development of a service oriented architecture (SOA) for automation systems.

## 2.1 AGENCY

The agent concept arose in Hewitt's Actor Model [24] and was widely researched in the 90s in the multi-agent system context. When research in AI changed its focus from goal seeking to rational behavior, from ideal to resource-bound reasoning, from capturing expertise in narrow domains to re-usable and sharable knowledge repositories, from the single to multiple cognitive entities acting in communities, agent technology made a breakthrough. Agent technology is looked upon as a strong possibility for future manufacturing system implementations.

In [25] Jennings and Wooldridge present the following definition for agency: *"An agent is considered a software entity situated in a production environment, with enough intelligence that is capable of autonomous control actions in this environment and of co-operation relationships by participating in associations agreements with other entities in order to meet its design objectives. An agent should be able to act without the direct intervention of humans or other agents, and should have control over its own actions and internal state"*. Most agent definitions relate to the author's background but a number of characteristics seem to be widely accepted [26]:

- Autonomy – an agent is autonomous when it is able to act alone without help from third parties (like other agents or humans).

- Sociability – an agent must be able to communicate with other agents or even other entities.
- Rationality – an agent can reason about the data it receives in order to find the best solution to achieve its goal.
- Reactivity – an agent can react upon changes in the environment, changing its behavior accordingly.
- Proactivity – a proactive agent has some control on its reactions basing them on its own agenda and objectives. A proactive agent might not react against changes in the environment should that reaction go against the achievement of its final goal.
- Adaptability – an agent is capable of learning and changes its behavior when a better solution is discovered adapting itself to changes in the environment.
- Mobility – an agent is capable of moving inside its network keeping its state of execution.

An agent is a computational system that operates in an environment that may include other agents. An agent observes its environment, has its own knowledge and beliefs about its environment, has preferences regarding the states of the environment and initiates and executes actions to change the environment [27].

An agent can be perceived as an elaborate model to enable planning of a complex system. To model its decision making process as well as its knowledge base regarding its environment the BDI-model (Belief, Desire and Intention) can be used. The BDI-model includes the agent knowledge about it environment (beliefs), preferred states to achieve in the long-term (desires) and planned decisions to be made to complete a plan (intentions) [28]. A BDI agent is continually updating its knowledge base (beliefs) on its environment and using it to reason about possible plans. It acts by realizing its intentions which are based on its beliefs and desires. This model gives the agent the means to generate actions based on goals that can be added or retracted to the agent.

A system that has multiple agents is called a Multi Agent System (MAS). When an agent's environment contains other agents a community emerges and communication and coordination protocols are needed. Agents need means to make decisions and execute plans even when information or physical resources aren't readily available to it. It must then gather information and establish contracts with other agents in order to reach its goals. A widely

used protocol for this goal is the Contract Net Protocol (CNP) [29]. The contract net protocol is modeled on the contracting mechanism used by businesses to govern the exchange of goods and services. Whenever an agent requires that a task is to be made it announces the need and requests for bids. Other agents that can complete the task make their bidding after which the agent chooses the most suitable agent to perform the task evaluating its bid. The chosen agent gets the contract.

Agent systems announced the need for communication between agents and the BDI model provided the theoretical basis for Agent Communication Languages (ACL). The first ACL was the Knowledge Query and Manipulation Language (KQML) that included means for agents to tell facts, ask queries, subscribe to services and search for other agents. Nowadays the most commonly used ACL is the Foundation for Intelligent Physical Agents (FIPA) standard [30].

Agents work as individual problem solvers that can sense and act upon their environment. They can use their knowledge of the environment to reason, learn and create plans. They have means to find other agents, communicate with them and establish contracts in order to fulfill their goals. An agent sophisticated decision model is prepared to deal with situations not predicted by the system programmer and add dynamism to its system.

Due to their autonomous nature, software agents present themselves as a potential solution to manufacturing systems. A number of frameworks were implemented to give voice to this need but, due to undefined standards, most are not interoperable. Interoperability [31] between different agent system platforms is thus a pursued goal.

Although there has not been a framework that completely complies with all of the agents definition characteristics, some implemented frameworks show interesting developments in this area, such as JADE [32] or ZEUS [33] amongst others. There has been an effort in interoperation between multiagent frameworks. A number of initiatives to create open standards arose so that different frameworks could interact with one another such as the FIPA foundation [30], a number of European projects such as SOCRADES [2] and others.

The communication technology used by each platform differs from one another mainly due to the implementation time. Update to modern, more suitable, communication technology presents concerns with backward compatibility. An open set of standards that

provide the freedom required by agent technology as well as grant them their definition needs is necessary for it to evolve to full feature interoperable platforms.

With several initiatives and large scale research, Agency is well underway in becoming a major player in the future of manufacturing world.

## 2.2 MULTI-AGENT SYSTEM (MAS)

Multi-Agent Systems are believed to be an important approach in automated manufacturing systems. In the 80s decade, distributed computing over LAN and expert system advances motivated the interest in distributed agents. This new paradigm began to be widely researched during the 90s decade due to the launch of the internet. That research was mostly focused around agent interaction [34]. The MAS paradigm has the potential to create a solution that has advantages, over current centralized, highly hierarchical systems, in feasibility, robustness, flexibility, reconfigurability and redeployability [35; 36].

A Multi-Agent Systems is formed by a group of agents interacting and communicating in the same network. Each agent has its knowledge base that includes all information gathered from its environment constituting its model of the environment. Each agent also has a limited set of means to gather that information and to pursue its goals. To reach them it may have to request services from other agents if his means do not suffice. The environment can have different models depending on the agent, for the same environment can be perceived in a different manner depending on the agent means. Whenever a new kind of interaction with the environment or an agent with different capabilities enters the network the MAS should be able to incorporate these new tools and evolve. MAS are built around interactivity between agents and that interaction enables agents to solve problems and accomplish goals that no individual agent could accomplish. That interaction may result in emergent behavior which is not planned when the agents are implemented.

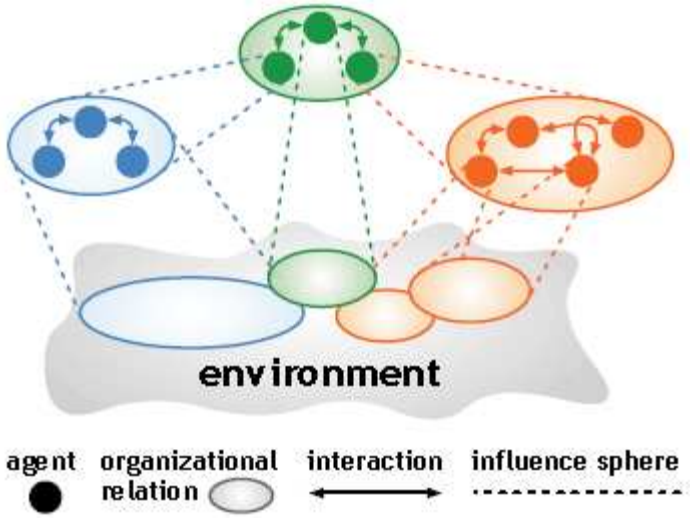


**Figure 2-1 - Generic scheme of a multi-agent system [37].**

In MAS, even though there may be no centralized data or control there is organization. That organization determines the agents' possible activities and interactions as shown in Figure 2-1.

Communication in a multiagent system can be made indirectly through the environment or by direct exchange of information. In the latter there is need of a language with syntax and semantics that both intervening agents share in order to have communication. To that effect an Ontology can be used. Ontology is an explicit-specification of the structure of concepts used by a given domain. All data is usually expressed in a logic-based language that distinguishes classes, instances, properties, relations and functions in a machine-processable way. The data can be checked for consistency and reasoning can be accomplished in order to infer data from the available information. Ontologies can be used as a ground for different agents to communicate effectively with each other and serve as a repository of content and knowledge.

Enterprise integration is another field where MAS could play an important role. Integration between business, engineering, operational and administrative functions is required in order to facilitate information exchange, decision making, coordination and collaboration in the enterprise. Due to globalization, the nature of enterprises tends to be distributed. This makes modeling, monitoring and control of processes critical. A system that improves collaboration in an enterprise would help develop better flexibility, reliability and performance. Using the MAS paradigm in enterprise integration would enable the continuity of operations even during temporary lapses in connectivity. For such an implementation some

issues such as scaling, temporal, representation of process models and constraints between business functions have to be addressed. [27]

With computer systems becoming ever more complex and due to the need of more sophisticated systems to control present manufacturing we need powerful abstractions and metaphors to design systems capable of such tasks. The agent and MAS concept helps us to structure our knowledge around self controlling components that have the autonomy and capability to communicate between them and solve these new problems.

The manufacturing world called for new, more robust, adaptable, fault-tolerant, decentralized and open organizational structures even before the agent and MAS paradigm arose from artificial intelligence and computer science [38; 39].

Several projects in this area are being conducted to create a multi-agent platform that implements all the required characteristics of a multi-agent system. To achieve that, some multi-agent system characteristics have to be taken into account, such as autonomy and a truly distributed environment, to name just a few.

Current multi-agent platforms such as JADE [32] or ZEUS [33] provide some interesting implementations but none present a truly distributed system. Most developed environment implementations of MAS are focused on agent models and communication. To achieve true autonomy between agents a truly distributed platform must be implemented.

Due to different implementations of multi-agent platforms another problem that arises is interoperability. Different protocols, message types and even technologies turn agents to stranded software entities that can only communicate with other common framework agents. The pursuit of standard communication protocols for agent communication began and led to the creation of the Foundation of Intelligent and Physical Agents (FIPA) [30]. FIPA is a foundation that promotes interoperability between different agent frameworks by establishing agent communication standards.

Nowadays the Multi-Agent paradigm is accepted as a promising approach that satisfies many requirements of present manufacturing. In [36] the authors show that at least 25% of manufacturing problems can be solved by using an agent based approach. There is, now, a clear understanding that the autonomy, embodiment, communication and cooperation provided by a society of agents can meet the requirements of modern manufacturing.

There are, however a number of disadvantages to a MAS approach, such as constraints related to available decision making time, properties of the physical equipment and to the limited number of acceptable manufacturing structures. There are a number of arguments for the slow adoption of MAS in the industry. In principle MAS do not reduce the complexity of problems. Interoperability between different agent platforms and between agent platforms and legacy systems is expensive and, in most cases, inexistent. The increased overhead in agent communication can degrade the performance in MAS. This problem especially affects rough-grained systems which in consequence cannot be scaled up [40; 41]. There still is not Industry level support to agent-oriented software engineering. The lack of methods for automatic wrapping of legacy systems is still an issue [42]. The emergency characteristic of MAS is a serious barrier to the adoption of this approach due to the inherent uncertainty. The industry wants total predictability and guarantees regarding reliability and operational performance. The adoption of MAS is risky and expensive but the benefits may overcome these difficulties.

## 2.3 SERVICE-ORIENTED ARCHITECTURE (SOA)

With current market demands overrunning the old manufacturing world legacy systems the need for reconfigurable, robust systems that can support the new world requirements is high. Over the last few years the Service-Oriented Architecture (SOA) paradigm has gained much attention from the information technology scientific community. With the advent of new technologies such as web services, the SOA paradigm is seen as a promising approach to create platform agnostic interoperable systems.

The World Wide Web Consortium (W3C) defines Web Services as [43]: "*a software system design to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP* [44] *messages typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards*".

The emergence of Web Services was accompanied by the availability of low cost and high performance embedded devices as well as the consolidation of internet related

technologies. Some of the most commonly used Web Services technologies giving it its platform agnostic status are:

- Extensible Markup Language (XML) [45] – platform neutral data model;
- Simple Object Access Protocol (SOAP) [44] – data communication and encapsulation protocol;
- Universal Description, Discovery and Integration (UDDI) – xml-based service advertisement and discovery mechanism.

Web Services also have downfalls that mirror themselves in Web Service based SOA implementations. Web Services do not properly support transactions and frequently exhibit code explosion when there are interactions between many heterogeneous services. In most current implementations, when adding a new service with an unknown description to an already complex system, every service that interacts with it has to be reprogrammed. [46]

In [31] the authors provide a definition that, although incomplete, includes two very important keywords: *"A service-oriented architecture (SOA) is a set of architectural tenets for building autonomous yet interoperable systems"*. Although incomplete this definition includes two of the most characteristics of the SOA paradigm. The main characteristics present in most SOA definitions are: [47]

- Autonomy: all services are independent and structurally decoupled.
- Interoperability: achieved by the specifying the hosted services and interaction patterns.
- Platform Independence: services described, ideally, using text-based formats such as XML [45], WSDL [48], ebXML, etc. These representations are platform, architecture, programming language and technology independent. These languages can easily be decoded by any system.
- Encapsulation: services expose functionalities through clean interfaces that hide the complexity therein.
- Availability/Discovery: services can be published and made available for public use.

The emergence of the SOA paradigm has helped the research and development of modern distributed control paradigms and architectures. Most research on this area is directed at e-business and inter-enterprise interactions. The success achieved by SOA in these areas

motivated the development of light weight web service stacks, such as DPWS, to enable integration from the shop floor to the business level. For this achievement, the next generation of automation systems must rely on automated, decoupled, intelligent, low cost computing devices. The SOA paradigm enables the development of systems with: encapsulated self-contained functionalities, independent development of each service, environment independency, loosely coupled, distributed complexity, etc. Combined with MAS the SOA paradigm seems well underway to becoming one of the next generation automation standards.

A number of projects have been developed and are being developed around the SOA paradigm:

- the *ITEA-SIRENA* [1] project, led by Schneider-Electric, developed a protocol Stack of a web-service based SOA description. The result of this project was the Device Profile for Web Services (DPWS) [49] protocol stack aimed at the device level.
- the SODA [23] project, following the design effort of the award winning SIRENA, was launched with the objective to create a DPWS framework. The SODA project has developed two compliant implementations of the DPWS Stack in C and JAVA. With any of these protocol stacks it is possible to have truly distributed devices exposing their services through DPWS web services. This project is ongoing and aims to create a service oriented ecosystem based on the Device Profile for Web Services (DPWS).
- the SOCRADES [2] project aims to develop a DPWS-based design, execution and management platform for automation systems making the most of the SOA paradigm at the device and application level.
- the InLife [50] project aims to include a test case that explores service oriented DPWS-based diagnosis on distributed manufacturing systems [51]. It explores SOA and web services for diagnosis and to optimize device's life cycle management.

These initiatives demonstrate the effort in the development of ubiquitous service oriented technology and the wish to achieve embedded web services in small devices. These can be applied in numerous areas such as automations, automotive and home electronics, telecommunication systems, etc. In [49] it is presented the possibility, with some research, to

create dedicated hardware that will be able to process messages faster than 1 millisecond making these systems faster to respond than present PLCs.

SOA is being widely researched in both European projects and at academic level. Amongst many one might mention:

- in [52] the authors present an approach to integrate real and simulated production automation devices using DPWS. The Simulation of manufacturing systems prior and during deployment is a desired functionality in the next generation of automation systems,
- in [53] the author presents a diagnostic infrastructure to be added to the production line in order to avoid and predict system down-time. Self-healing is a characteristic of the presented approach.
- In [54] the authors present a generic interface for DPWS-based devices. This interface sustains system reconfiguration rather than reprogramming. The authors state that this approach scales reasonably and reduces the memory footprint.

Within SOA enabling technologies, Web Services is the most researched and explored as it seems a promising ubiquitous computing technology. Consequently, it is often chosen as the "vehicle" for implementing SOA platforms.

Implementation related work highlights SOA's downside. In very complex and heterogeneous environments where changes happen dynamically the complexity of the application's code explodes and reprogramming is often needed. [54]

By uniting SOA with the Multi-Agent System paradigm the desired truly distributed multi-agent platform can be achieved. Also, with the SOA paradigm the pursued integration of the device level with enterprise level can be achievable [55] [49].

## 2.4 MANUFACTURING PARADIGMS

Recently, manufacturing paradigms such as Bionic Manufacturing Systems (BMS), Holonic Manufacturing Systems (HMS), Reconfigurable Manufacturing Systems (RMS), Evolvable Assembly Systems (EAS), and Evolvable Production Systems (EPS) are gaining

focus due to market demand. Old methods are proving unable to handle the rapid change of the current market.

The word Holon was presented by Koestler [56]. The word is a combination of the Greek word *"holos"* (the whole) with the *"on"* suffix suggesting a part. So a *"Holon"* is a whole and part at the same time. A Holon is defined by the Holonic Manufacturing Systems consortium as *"an autonomous and cooperative building block of a manufacturing system for transforming, transporting, storing and/or validating information and physical objects"* [57]. A Holon can be a combination of other Holons and be itself a part of another Holon. Holons are assumed to be autonomous and cooperative.

Another term regarding Holons is the Holarchy which is defined as *"a system of holons which can cooperate to achieve a goal or objective"* [57].

A Holonic Manufacturing System is *"a Holarchy which integrates the entire range of manufacturing activities from order booking through design, production and marketing to realize the agile manufacturing enterprise"* [57].

In Holonic Manufacturing systems the control is distributed by all participating Holons. Each Holon decides which actions to take based upon their knowledge. Cooperation is a trait of a HMS. Each Holon may cooperate with other holons in order to reach his goals and to satisfy system-wide constraints. In order to fulfill system goals holons may cooperate and form temporary coalitions. Holonic Manufacturing Systems tries to combine the responsiveness and robustness of decentralized, network-like organizations, and the stability and efficiency of hierarchical control architectures. [27]

A Multi-Agent System (MAS) is a particularity of this architecture type [58]. Agents in a Multi-Agent System, acting as Holons upon a manufacturing system and forming Holarchies to achieve their common goals can be regarded as a Holonic Manufacturing System.

## 2.5 SOA IN MAS

The use of SOA in MAS provides a new modeling metaphor for complex systems. It enables the development of distributed agents, encapsulated by a service interface, to interact

amongst each other in pursuit of their goals. The SOA architecture empowers agents to act in an environment as they are designed to, by definition. This turns the system into a collection of platform agnostic, self contained, loosely-coupled agents. As a multi-agent system the agents can group themselves and work together in order to provide functionality that no agent could provide alone. This reconfiguration capability is fundamental in modern production systems. [54]

The shift to this paradigm merge is being accompanied by increased offer in tiny embedded devices that have the processing power to support the development of intelligent automation environments. [47]

The MAS role has been mostly related with "what lies behind the interface". Most MAS platforms can only fully function in a LAN and their compliance is restricted to well defined but less interoperable standards. In contrast, SOA regards mainly the interface of a device or system and assures interoperability with a wide range of systems. Being typically supported by widely used web technologies it can easily spawn over the internet. SOA in MAS completes the circle and provides agents with the tools to expose their functionalities. By converging MAS and SOA in a unified framework we can attain unprecedented support to a wide range of complex networked systems.

**Table 2-1 - Comparative Analysis between SOA and MAS [47]**

| **Characteristics** | **SOA** | **MAS** |
|---|---|---|
| **Basic Unit** | Service | Agent |
| **Autonomy** | Both Entities denote autonomy as the functionality provided is self-contained | |
| **Behavior description** | In SOA the focus is on detailing the public interface rather than describing execution details | There are well established methods to describe the behavior of an agent |
| **Social ability** | Social ability is not defined for SOA nevertheless the use of a service implies the acceptance of the rules defined in the interface description | The agents denote social ability regulated by internal or environmental rules |
| **Complexity encapsulation** | Again, the self-contained nature of the functionalities provided allows hiding the details. In SOA this encapsulation is explicit | |

| | | |
|---|---|---|
| **Communication infrastructure** | SOA are supported by Web related technologies and can seamlessly run on the internet | Most implementations are optimized for LAN use |
| **Support for dynamically reconfigurable run-time architectures** | Reconfiguration often requires reprogramming | The adaptable nature of agents makes them reactive to changes in the environment |
| **Interoperability** | Assured by the use of general purpose web technologies | Heavily dependent on compliance with FIPA-like standards |
| **Computational requirements** | Lightweight implementations like the DPWS guarantee high performance without interoperability constraints | Most implementations have heavy computational requirements |

In [59] the authors present an Agent-Based Service Oriented Integration Framework where web services were used as a backbone of some of the agents. This integration was aimed at business transactions and e-Business therefore not targeting low cost computationally limited devices such as the ones aimed at the manufacture industry.

In [60] a reconfigurable HMS controlled by agents is presented. These agents communicate via DCOM. The achievements presented in [60] demonstrate the effectiveness of agent controlled manufacturing systems.

In [47] the authors present an attempt to merge MAS and SOA to create a lightweight environment for embedded devices. The DPWS framework was used providing web service interfaces to an execution model based on the agent concept. The case study demonstrates nearly forty independent entities spread across several machines in NOVAFLEX's cell interacting to complete assembly tasks. Through a web service interface each entity provides:

- Data Encapsulation
- Communication Support
- Complexity encapsulation
- Service Publishing and Discovery.

The agent inspired execution model behind the web services supports:

- Structured communication with the adequate semantics

- State control and interactions' monitoring
- Generic execution of process plans.

With the main features of SOA being the desired characteristics of MAS by definition, building a communication interface based on a SOA framework, such as DPWS, seems to be a promising step in the MAS paradigm.

# 3 DPWS Middleware Architecture

## 3.1 INTRODUCTION

With the main features of SOA being the desired characteristics, still not completely achieved, of MAS by definition, building a communication interface based on a SOA framework, such as DPWS, seems to be the next logical step in MAS.

By merging both the SOA paradigm with the MAS paradigm we achieve a truly distributed, autonomous and interoperable multi-agent system. The SOA paradigm provides autonomy, interoperability and a distributed environment. These characteristics are applied to devices with the DPWS protocol stack exposing their services to agents, human interfaces or PLCs. Allied to the multi-agent framework, a SOA multi-agent system is possible. With such an alliance the communication in a manufacturing system would be completely integrated from the device level to the business-level with agents in the loop. With such a framework it would be possible to find, communicating in the same manner, sensor, actuators, manufacture level agents, business level agents, simulators, databases, human interface applications amongst many others.

In [60] a reconfigurable HMS controlled by agents is presented. These agents communicate via DCOM. The achievements presented in [60] demonstrate the effectiveness of agent controlled manufacturing systems.

This thesis, developed in the SOCRADES [2] context, proposes an architecture that enables the DPWS Stack to be effectively used by agents and all other entities that participate in the manufacturing system from the device level to the business-level.

## 3.2 WHY A DPWS SYSTEM

Most multi agent systems on the market rely on a centralized unit, such as the Directory Facilitator (DF) in JADE [32]. This unit always has to be running and in case of a crash the system cannot run and has to be restarted. A truly distributed agent system must not rely on a centralized unit. Each agent must be truly autonomous.

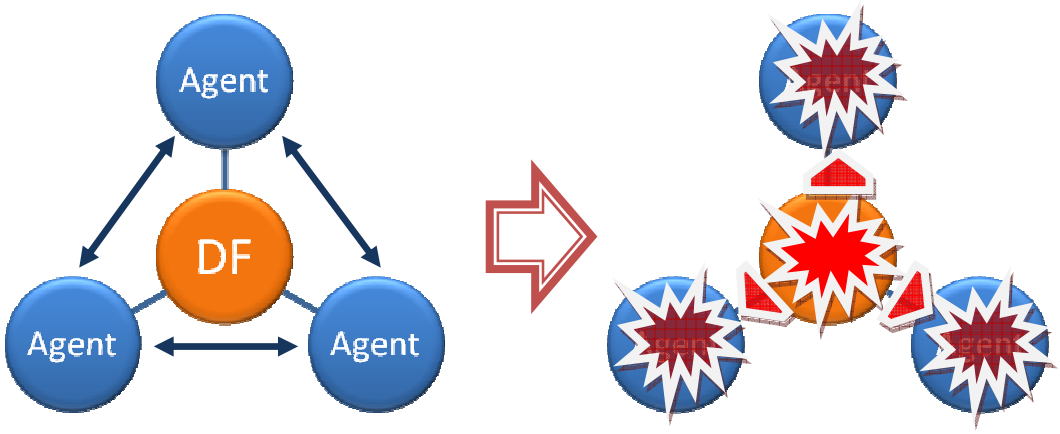


**Figure 3-1 – JADE Directory Facilitator Crash**

The search for a truly decentralized agent platform led to the Device Profile for Web Services (DPWS) stack of protocols. This stack, DPWS, provides, amongst other features, a web service interface and discovery capabilities to devices, thus disabling the need for a Directory Facilitator. With a DPWS interface agents would be able to create a truly decentralized, distributed system.

A truly distributed platform on a redundancy rich system, with no indispensable entities, will enable a nonstop production. During production there will always be crashes and errors. A certain amount of redundancy and the inexistency of unique, irreplaceable entities form a nonstop system. The DPWS protocol stack, by enabling truly autonomous agents in a truly distributed system, provides these possibilities.

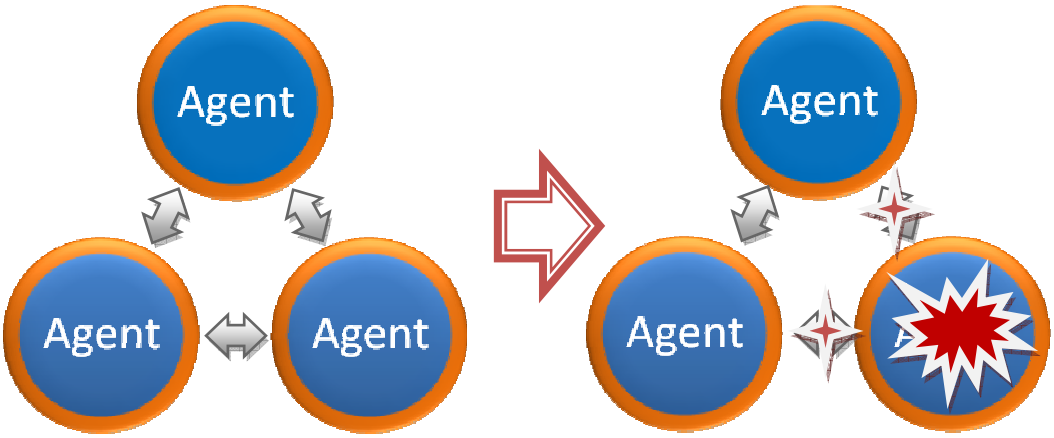


**Figure 3-2 - DPWS System Entity Crash**

Current DPWS implementations are only available for devices. The interface exposes the device services to the system and has direct control over the device when a service is requested.

In an agent system, when an agent service is requested the agent must first make the decision to execute what was requested or not and have the means to warn the requestor of its decision. A middleware that serves as a bridge between the agent and its communication interface that makes all communication transparent is needed.

## 3.3 PROPOSED ARCHITECTURE

The current development of the Schneider Electric DPWS Stack includes a number of protocols. The Stack currently has WS-Discovery, WS-Eventing, WS-Addressing, WS-Policy, WS-Security and WS-MetadataExchange:

- WS-Discovery provides a mechanism through which devices can find one another. Devices advertise their presence when they join and leave the network. Every device can, at any time, look for devices running on the network.
- WS-Eventing provides a mechanism for devices to announce changes through asynchronous messages. Devices subscribe to events published by other devices to be notified when the given event takes place.
- WS-Addressing provides a unique identifier to every DPWS entity. All addressing information is integrated in SOAP message headers allowing for the message content to be carried over any transport protocol (HTTP, SMTP, TCP, UDP …).
- WS-Policy is used to express policies associated to a Web Service in the form of "policy assertions", complementing the WSDL description of the service.
- WS-Security is an optional set of mechanisms for ensuring end-to-end message integrity, confidentiality and authentication. The DPWS protocol stack integrates all the above core standards. With DPWS, all messaging is based on the use of SOAP and WS-Addressing.

- WS-MetadataExchange allows for dynamical retrieval of metadata associated to a Web Service (description, schema and policy), thus providing a Web Service introspection mechanism.

In order to provide a transparent DPWS interface suitable for agent communication an extra layer between the agent and stack was implemented, the DPWS Middleware.

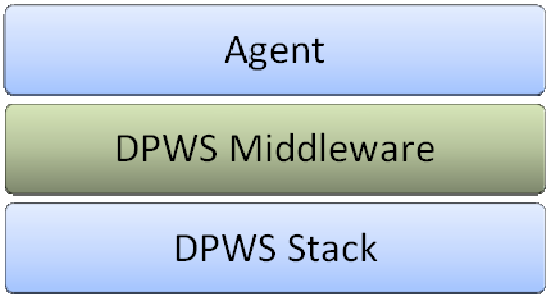


**Figure 3-3 - DPWS Middleware architecture location**

The DPWS Middleware should manage all communication leading to and from the agent. Discovery and metadata handling should be made at the middleware level, providing only relevant information to the agent. Other entity crashes should also be handled by the middleware whenever possible and communicated to the agent.

Event Subscription should also be managed by the middleware, it should maintain subscriptions requested by the agent. Subscriptions are made for a short period of time after which they expire. The middleware should maintain the subscriptions required by the agent and warn it in case they cannot be renewed.

Every service requested to the agent will be filtered of DPWS related data and delivered to the agent with only relevant information. Whenever the agent wants to request a service of a system entity it should only provide the entity name and the requesting service related information. The Middleware should then get the entity address, create the message and deliver it to the DPWS interface provided by the stack that will deliver it to the recipient.

### 3.3.1 OVERVIEW

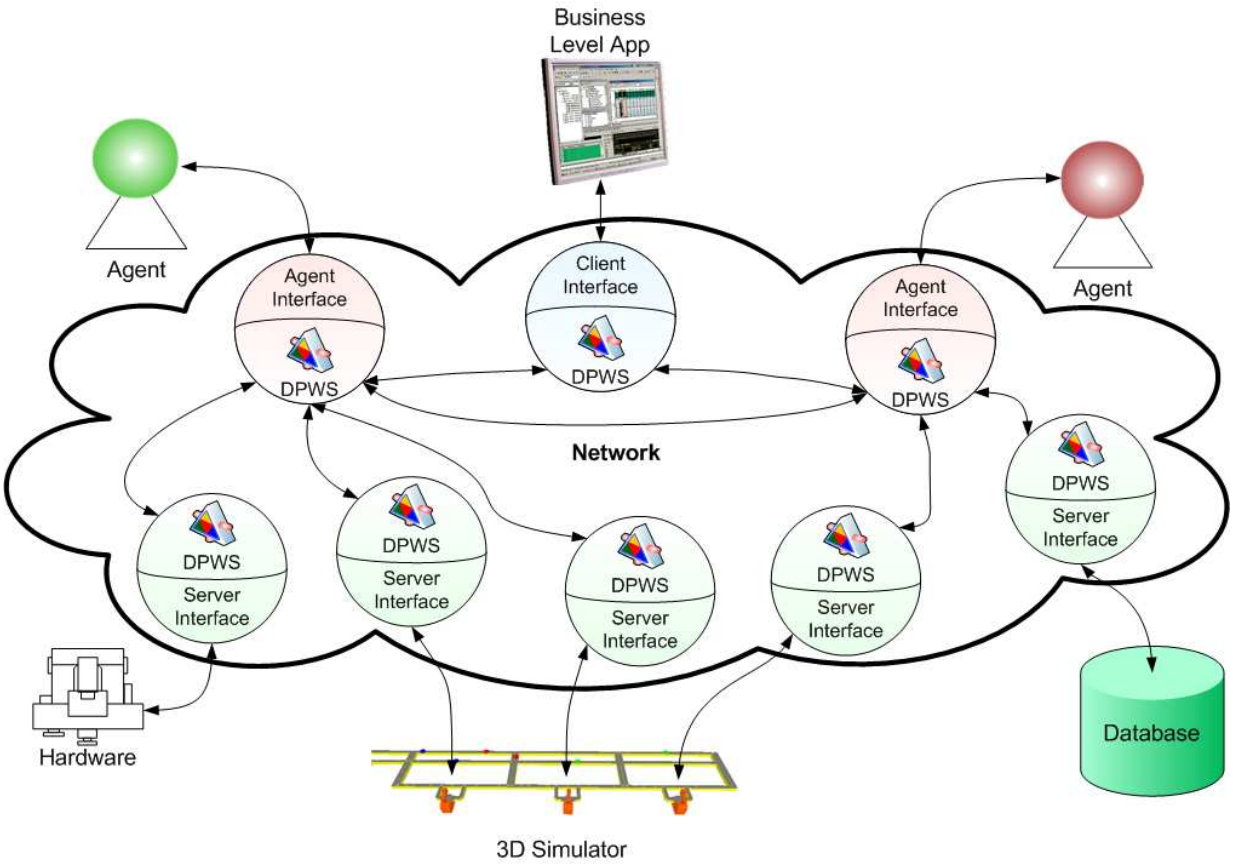


**Figure 3-4 - General Architecture**

The proposed architecture links every entity in a system through the DPWS interface. Every entity includes the DPWS middleware layer.

Each device will provide services that will be available for every other entity. With a wide range of services the system can multiply its functionality possibilities by adding various types of entities. With DPWS advantages the system can run, in a decentralized approach, avoiding problems that one without this architecture would have, such as entity arrival and departure warnings. The network transparency given by the DPWS Middleware layer provides an easy, low maintenance and simple configuration of the system.

Through this DPWS Middleware interface agents will be able to acknowledge joining entities as well as subscribe to events these devices may send and call services provided by them. Joining agents will look up for already running entities and send a Hello message to announce its arrival.

Redundancy is a strong characteristic in this kind of systems that enforces robustness by always providing an alternative to every entity in case of failure. This redundancy optimizes the production output of the Shopfloor.

### 3.3.2 System Entities

The DPWS Middleware is composed of a client and a server. Depending of the entity the middleware may need only one of these parts.

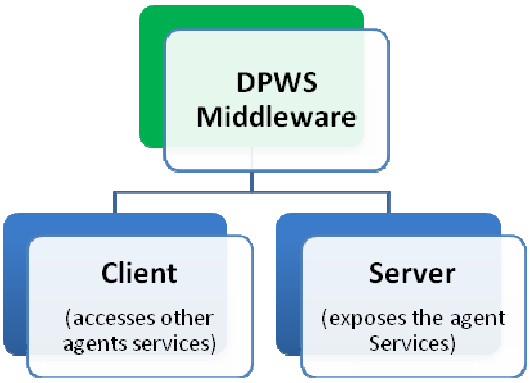


**Figure 3-5 – DPWS Middleware Top Layer**

A number of features need both the client and the server to be enabled but in some entities these may not be necessary.

There are three entity types that can be built with different implementations of the DPWS Middleware.

The more complex entity types, like agents and human interface applications, need both the client and the server since they will use other entities services and expose some of their own.

The simplest entities in a system, interface-wise, machines, sensors and actuators provide a number of services through a server but need not a client since they do not request services from others. The current DPWS stack is already sufficient for these entities, only in case these want to implement more complex services the need for the Middleware may arise.

Another type of entity is possible, a monitorization or error correcting entity that only implements the client part of the Middleware. This type of entity enters the system without announcement and acts upon it. It can use other entities services to correct the system or monitor it and expose no services.

#### 3.3.2.1 AGENT MIDDLEWARE

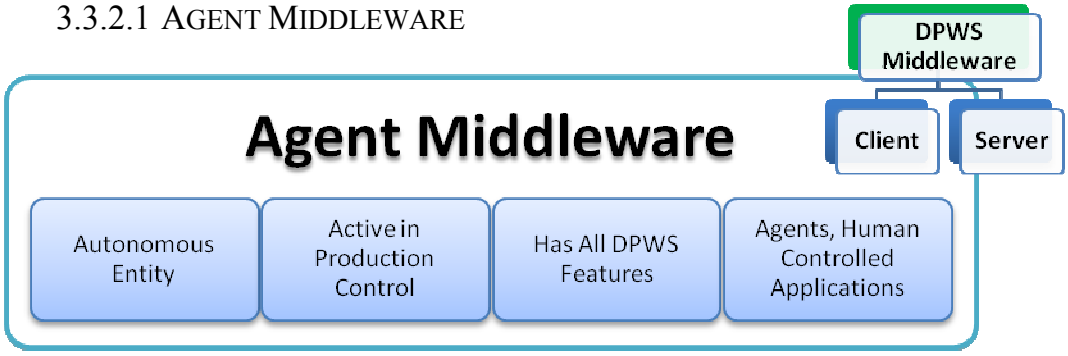


**Figure 3-6 – The Agent Entity**

The Agent DPWS Middleware is to be used by active parts on the system. Primarily designed for Agents it can also be used with human interface applications as Configuration Tools and Production Managers. This Entity Type has all DPWS features.

This Interface is the most versatile of all three as it is designed to act as the communication interface of autonomous entities. Every Entity that has part in the Production control should have an Agent Interface.

This interface enables the exposure of services provided by the entity and the use of other entities services of any type. Subscription management, automated discovery and message coding and decoding are all available through this entity type Middleware interface.

#### 3.3.2.2 SERVER MIDDLEWARE


**Figure 3-7 – The Server Entity**

The Server DPWS Middleware is to be used by entities like Machines, Sensors, Actuators, Databases and any other entity that will have a passive contribution to the system.

It will expose its services and announce itself to the system during joining, leaving and look up requesting by other entities.

Its services will be called by other system entities, like agents, that will be unknown for the Entity since it does not have Discovery capabilities.

In this entity the DPWS Middleware may not be necessary due to its simplicity. It may be used just for simplicity in implementation as well as for service integration. An entity with various services that want to interact with each other will need the Middleware to bridge them.

#### 3.3.2.3 CLIENT MIDDLEWARE

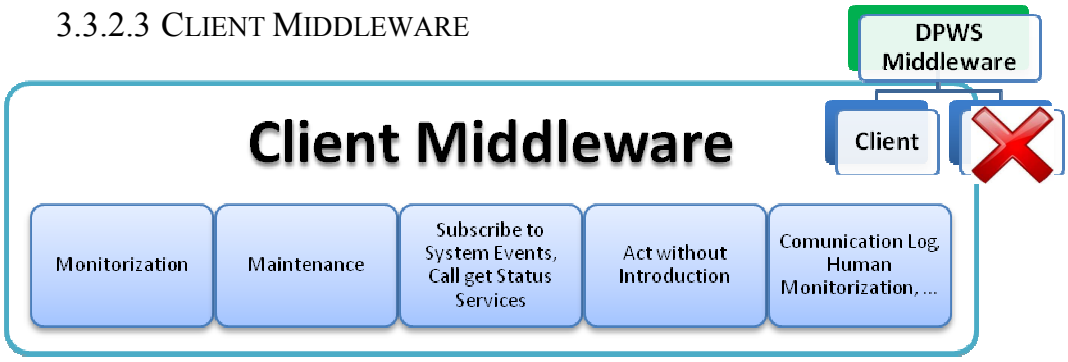


**Figure 3-8 – The Client Entity**

The Client DPWS Middleware should only be used in system monitoring entities as well as error correction entities. It is designed to simply subscribe to system events, call getStatus Services and use specific error correction services.

All action that this entity has upon the system will be invisible as it does not announce when it joins and leaves. An application with this kind of interface is able to fix system errors without introducing itself to the system.

It is directed at System Communication Loggers, Machine Monitorization and Maintenance.

### 3.3.3 HOW IT WORKS

This section will contain a description of how the DPWS Middleware works as well as the lower DPWS stack layer. Every type of message exchanged between any system entities will be covered as well as more complex entity interactions.

#### 3.3.3.1 DPWS STACK LAYER

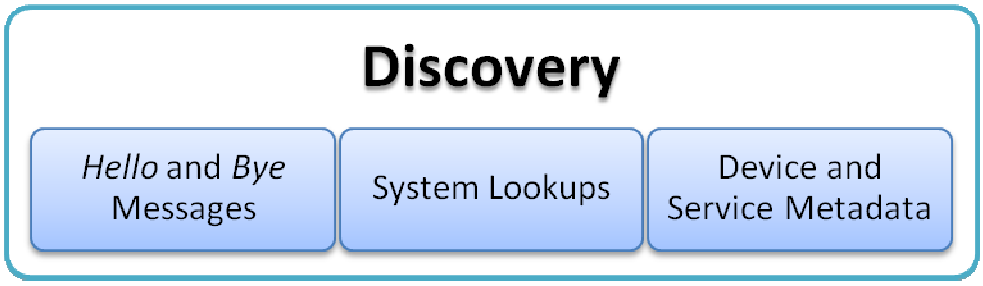


**Figure 3-9 - DPWS Stack Discovery Functionalities**

Discovery protocols included in the DPWS Stack have any entity joining the system sending a Hello message and, if leaving correctly, sending a Bye Message. Also in Discovery, an agent can look up for other, already running, entities by sending a probe and waiting for matched responses. This mechanism is part of the DPWS Stack. With Metadata Exchange protocols an entity can, at any time, request other known entity metadata as well as the exposed services metadata.


**Figure 3-10 - Services Types**

When the system is running most exchanged messages are of Request, Request Response and Event type. When an agent requests a service from an entity that does not require an answer a Request message is sent. When an answer is required a Request Response communication takes place. When an entity wants to announce that a certain event has occurred, an Event message is sent to all subscribers. The type of these messages is established at development time when designing the web service interface. Each service operation has a pre established communication type.


**Figure 3-11 - Subscription Operations**

For an entity to announce an event to other entities these have to, previously, subscribe to that event. This is made through a Subscribe request sent by the entity. The Subscription expires, after some time, before which the entity must renew the subscription with a Renew Subscription Message. At any time the entity can get the status of a subscription or cancel it with the Get Subscription Status message and Cancel Subscription message, respectively.

These are the basic functionalities provided by the DPWS Stack. With the proposed architecture, by adding the DPWS Middleware layer, all these functionalities will become available in a much simpler form. New functionalities suitable for agents and other entities are also implemented in the Middleware.

#### 3.3.3.2 DPWS MIDDLEWARE LAYER


**Figure 3-12 - Middleware discovery features**

Discovery has been simplified and filtered to present only relevant information for the agent. Only after all the metadata structure has been acquired from new entities does the Middleware present this new entity to the agent by giving it the entity name and type along with some possibly relevant information such as topology data.


**Figure 3-13 - Middleware communication features**

Agents only work with entities names and types, all information related to communication and message coding and decoding is filtered by the Middleware. Message building, with correct address and data types, as well as decoding received messages, to objects the agent can work with, is made by the Middleware. It keeps tables of entities and their services addresses in order to simplify the agent communication.


**Figure 3-14 - Middleware Subscription management features**

Subscriptions are kept in the Middleware and maintained. Subscriptions should always be renewed until the agent requests their cancelation. If a subscription cannot be renewed, then the service is no longer available and the entity can be considered out of order. In this case the Middleware warns the agent of the abnormal departure of the entity from the system.

#### 3.3.3.3 AGENT LAYER

The agent receives entity names and types it can contact. It can request services of other entities by giving their name to the Middleware. The agent can request a subscription for a service and it will be maintained. With the DPWS middleware the agent can request a service as if it was a local method call, enabling an easy distributed decision making.

#### 3.3.3.4 MESSAGE TYPES

In this section the all of the basic DPWS Stack messages and message types will be described.

##### *3.3.3.4.1 DISCOVERY MESSAGES*

These message types are implemented by the Discovery protocols of the DPWS Stack. These messages are sent to announce the joining and leaving of an entity into and out of the system. A look up method is also available to search for other entities in the system.

To get more information about any entity and its services the Discovery protocols included metadata requests.

- ***HELLO***

Whenever a new entity joins the system it sends a Hello message to all other entities already running through multicasting. This message contains the unique identifier UUID of the new entity as well as scopes, types and a list of hosted services.

- ***BYE***

When an entity leaves the system properly it will send a Bye message to all other entities in the system through multicasting. This message also contains the unique identifier UUID of the leaving entity.

- ***LOOK UP***

The look up mechanism provided by the Discovery protocols in the DPWS stack has a number of messages involved.

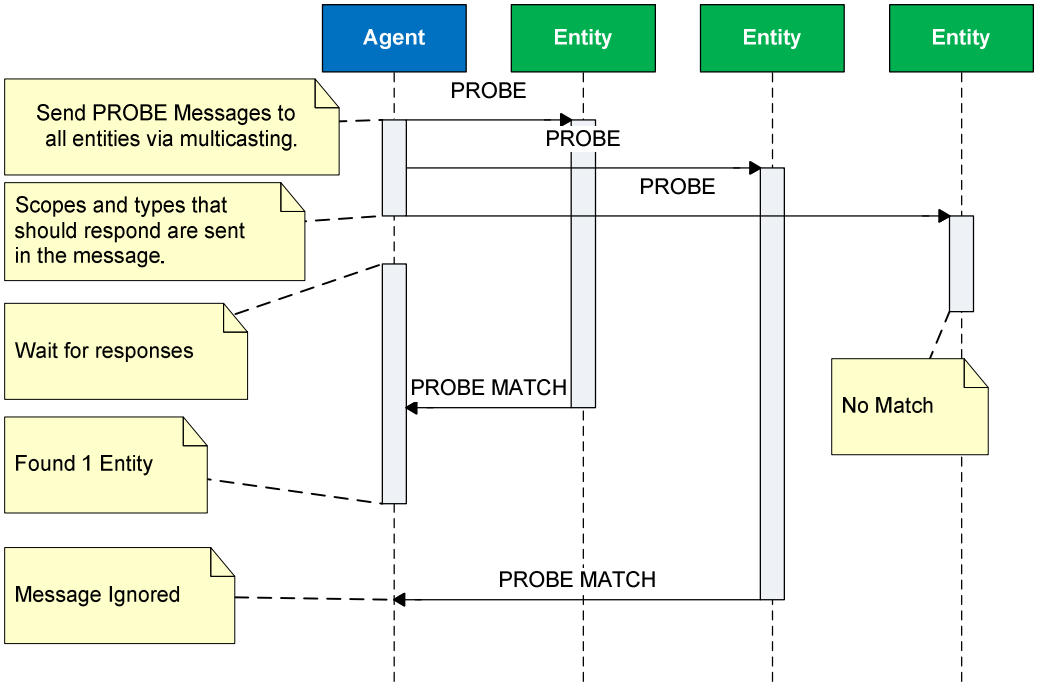


**Figure 3-15 - Look up communication diagram**

The entity that does the look up sends a Probe message to the system entities through multicasting with the scopes and types of the entities that it wants an answer from. It then waits for some time (default wait time is 5 seconds) to receive Probe Match messages from every entity that matches the given scopes and types. After the waiting time the Stack gives the Middleware the number and address of every matched entity that responded. Messages arriving after the waiting time are ignored. The Probe Match messages contain the same information of the Hello message.

- ***GET DEVICE METADATA***

The Get Device Metadata message is usually sent after receiving a Hello message or after a look up in order to get more information on the newfound entity. The entity receives the message and responds with the requested information. Two groups of information can be extracted from the response to this message: device information and model information. Since DPWS was mainly developed for devices hence the given names. Device information includes the device Friendly name, serial number and firmware version. Model information contains manufacturer, manufacturer URL, model name, model number, model URL and presentation URL.

- ***GET SERVICE METADATA***

The Get Service Metadata message is sent to retrieve more information about a certain service. The response to this message contains the service ID, type and endpoint reference. With this information the requesting entity will have all the necessary data to request services from the responding one.

#### *3.3.3.4.2 SERVICE MESSAGES*

The message types described below are the messages exchanged by entities when services are requested. When an entity requests a service from another entity it will send a Request message for that service.

A Request Response service incorporates a response message from the requested entity to the requesting one. Usually these types of services are used to get information.

A third service type, Event, is sent when an announcement is made to all entities that previously requested it by subscribing.

These three operation types are built at design time in the service descriptor (WSDL). They must be carefully chosen to expose correctly the services provided by the entity.

- ***REQUEST***

This operation type is used often used to start an action. The requesting entity makes a service Request to other entity as described in that entity service descriptor.

A message with the required parameters has to be build and sent. The message does not require parameters if none are necessary.

- ***REQUEST RESPONSE***

This operation type is often used to request information. It is similar to the Request operation but requires a response message in return. Both messages have their previously designed parameters.

The Request message can have no parameters but the Response message is obligated, to have at least one, by the stack.

- ***EVENT***

The Event operation is usually used as an announcement service that an entity may implement. This operation type has a mechanism of Subscriptions build to make it work. This mechanism is described below.

When an entity wants to announce an event it checks the list of subscribers and sends the Event message to all of them.

The Event message may or may not have parameters. That is an option made at design time.

#### *3.3.3.4.3 SUBSCRIPTION RELATED MESSAGES*

In this section the messages regarding Subscriptions are explained. When an entity needs to receive events from another entity it subscribes to those events. Every subscription has expiration and has to be renewed before it expires. While the subscription is valid the entity will receive events from the entity it subscribed to.

At any time the subscribed entity can cancel the subscription, renew it or get the subscription status from the entity it subscribed to.

Subscriptions are managed internally in the DPWS Stack on the server side, the entity that exposes an Event service.

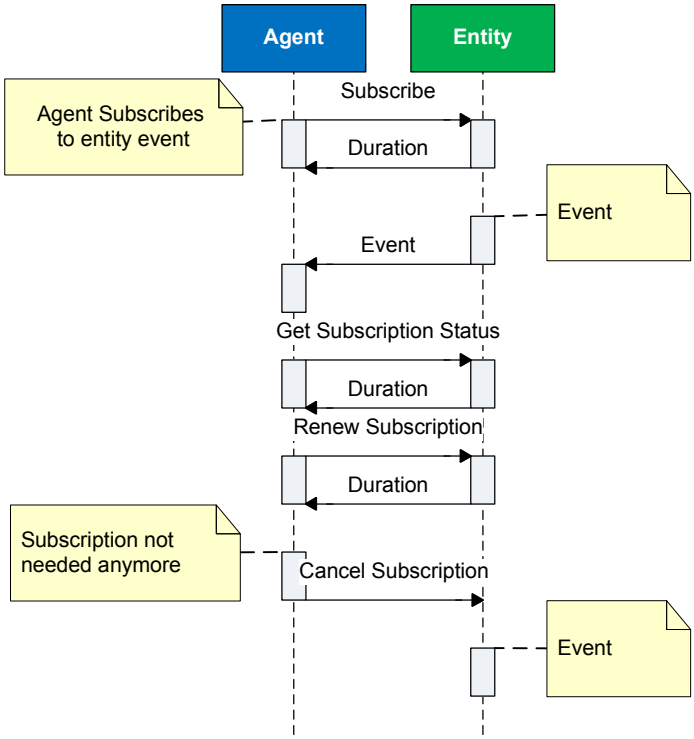


**Figure 3-16 - Subscription Communication**

A subscriber can subscribe to a single event or all of the events of a port type with a single subscription by adding filters to it. When a subscription is made the service returns the duration of the subscription to the subscriber. With that information the subscriber can renew the subscription before it ends if necessary.

- ***SUBSCRIBE***

The Subscribe message is sent by an agent when it wants to be warned when a certain event takes place. The agent then sends the Subscribe message to the selected entity, with a list of services it wants to subscribe to (entire port types or single events), and the desired duration of the subscription. The recipient then responds with the actual duration of the subscription. If the requested duration is more than what is defined as the maximum duration, at the recipient, it will send the defined duration, if less then it is accepted.

- ***RENEW SUBSCRIPTION***

Every subscription has a duration after which it will expire. If an agent wants to keep receiving events regarding that subscription it must keep renewing it before it expires. This is

made with a Renew Subscription message. This message contains the subscription ID and the desired subscription duration. The recipient responds to this message with the actual duration of the subscription. If the subscription expires this message is ignored.

- ***GET SUBSCRIPTION STATUS***

Whenever an agent needs to know what is the status of a subscription it sends a Get Subscription Status message to the recipient. This message contains the subscription ID and the response contains the remaining duration of the subscription. If the subscription expires this message is ignored.

- ***CANCEL SUBSCRIPTION***

If a subscription is no longer needed by an agent the agent can cancel it by sending a Cancel Subscription message. This message contains the subscription ID. This request is only valid if a subscription was made and in case it expires the message is ignored.

#### 3.3.3.5 JOINING

Joining the system is made differently depending on entity type. When an Agent Entity joins a system it announces itself and its services and looks for other entities. The Agent Entity subscribes to other entities services and takes subscriptions for its services.

When a Server Entity joins a system it announces itself to the system and waits for requests. It has no further action. It accepts Subscriptions and service requests.

When a Client Entity joins a system it looks for other entities but never announces itself. It can then use the other entities services but remain invisible to every entity. It exposes no services.

When an entity is acknowledged by an agent, be it through Hello message or Lookup, the agent requests its device Metadata as well as Service Metadata to know more about the device and how to call and subscribe to each service. This procedure takes place before subscriptions can be made, after the agent receives a Hello message and right after a look up. Whenever a previously unknown entity is discovered the agent requests for all metadata information. This procedure is a standard procedure in Agent and Client Entities.

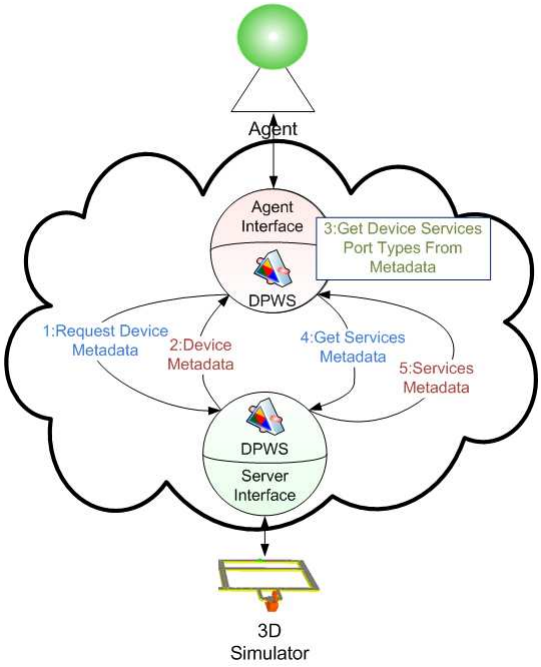


**Figure 3-17 - Request Metadata**

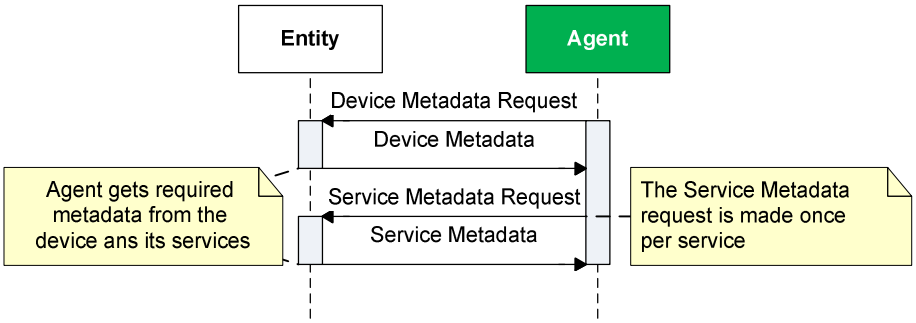


**Figure 3-18 - Metadata Request message sequence**

#### *3.3.3.5.1 AGENT ENTITY*

When the joining entity is an agent it sends a Hello message, to all entities already running, signaling its arrival and the availability of its services. It then performs a lookup to know what other entities can be found running. After finding them (receiving lookup acknowledgement messages) it can subscribe to their services and start pursuing its goals.

Metadata exchange takes place after discovery but before subscription.

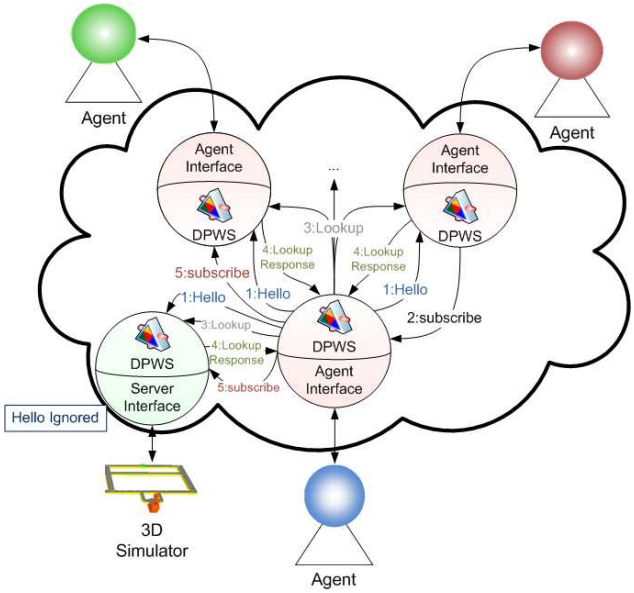


**Figure 3-19 - Agent joining the system**

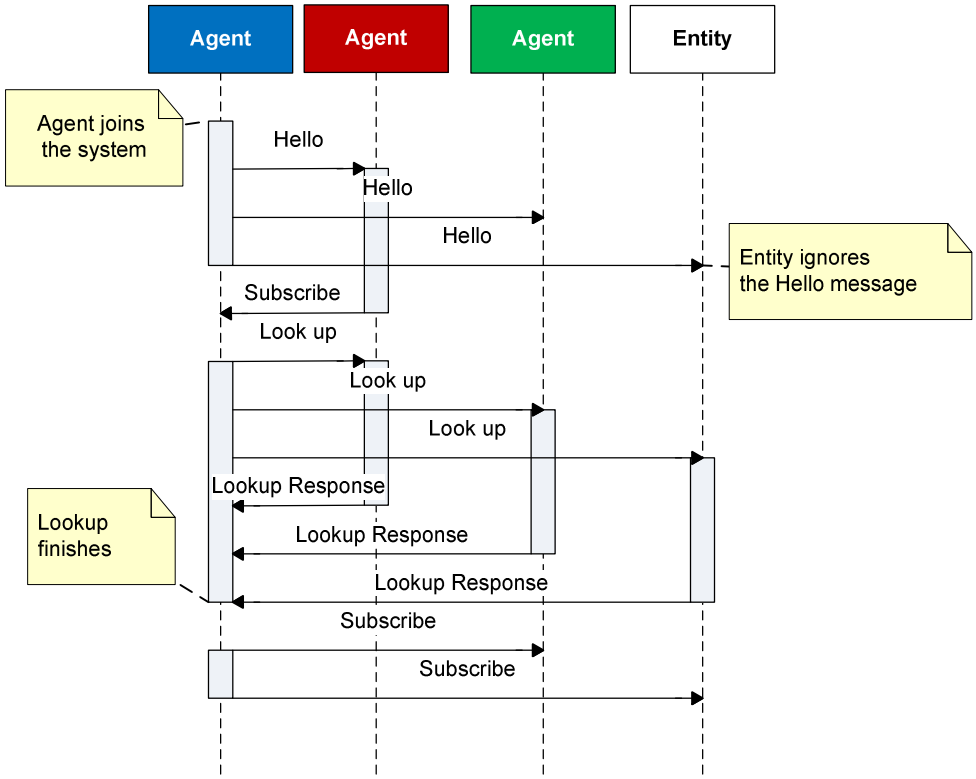


**Figure 3-20 - Agent Join message sequence**

#### 3.3.3.5.2 SERVER ENTITY

When a Server Entity joins the system it sends a Hello message, to all entities already running, signaling its arrival and the availability of its services. Agents can now subscribe for its services if they are relevant for their goals. Subscriptions are made for a certain duration before which they must be renewed or simply ignored.

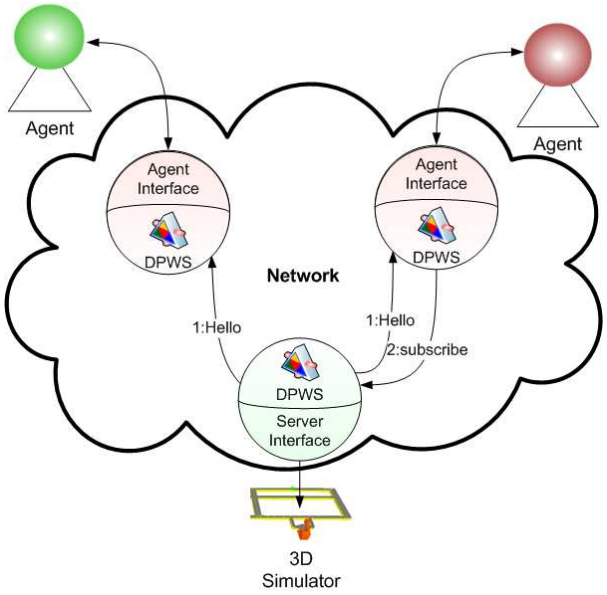


**Figure 3-21 – Server Entity Joining the system**

Before Subscribe takes place there is always metadata exchange between the agent and the entity. This exchange is described in the beginning of this section.

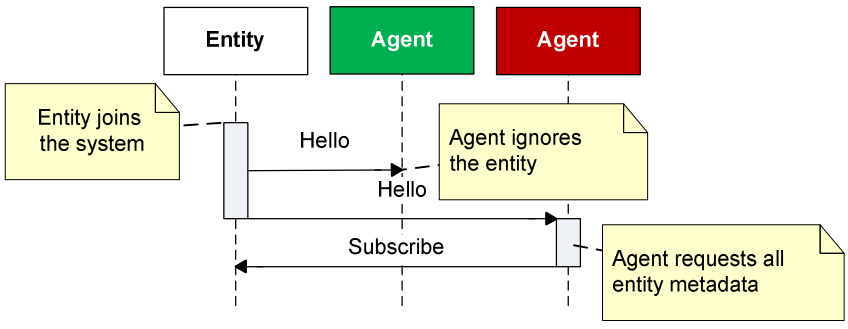


**Figure 3-22 – Server Entity Join message sequence**

#### 3.3.3.5.3 CLIENT ENTITY

When a Client Entity joins the system it does not sends a Hello message like Agent and Server Entities. The first system related action it does is a look up for running entities. After the look up it gets the metadata of those entities. It can now subscribe for and request their services. This entity never announces itself to the system remaining invisible to every other entity.

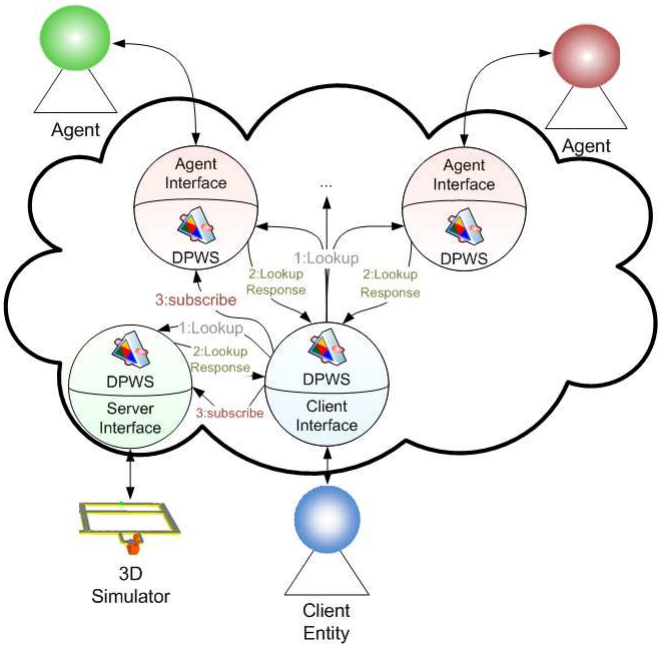


**Figure 3-23 – Client Entity Joining the system**

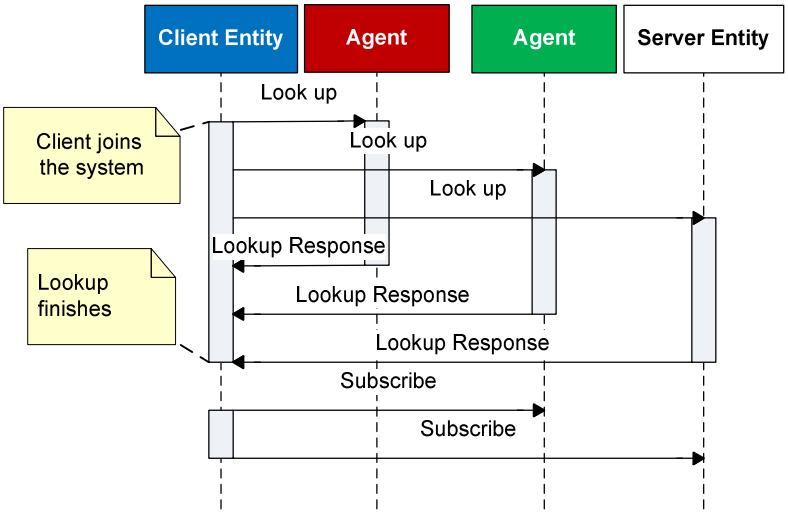


**Figure 3-24 – Server Entity Join message sequence**

#### 3.3.3.6 LEAVING

There are two possibilities when a Server or Agent entity leaves the system. The normal possibility is when the entity is shutdown and sends a Bye message to all entities in the system. The other possibility happens when the entity crashes.

When a Client Entity leaves the system it simply shuts down. Since it did not announce itself when joining, it does not have to announce its leaving.

When a Server or Agent Entity leaves the system it sends a Bye message to announce its departure so that agents know that that entity can no longer be contacted.

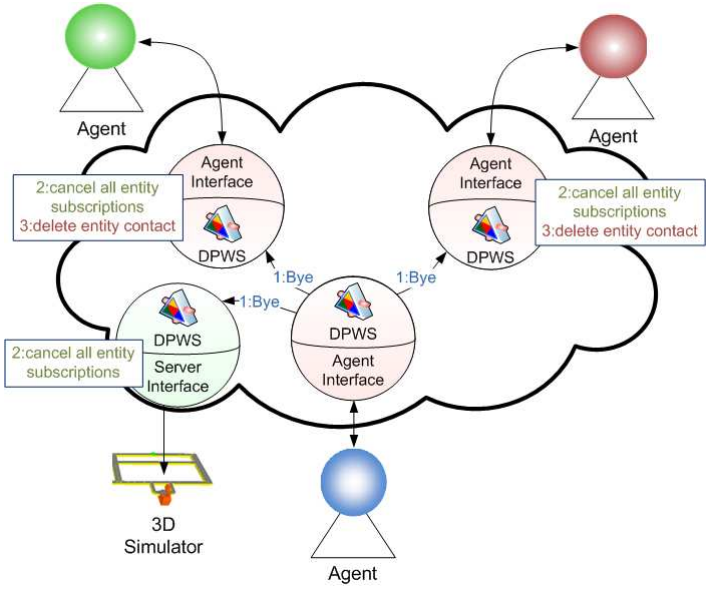


**Figure 3-25 - Entity leaving**

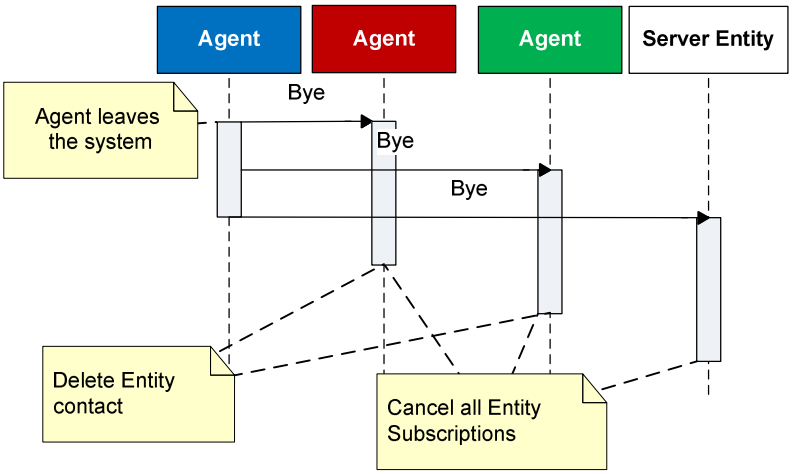


**Figure 3-26 - Entity Leaving sequence**

#### *3.3.3.6.1 Entity Crash*

In case the entity crashes it does not send the Bye message. It will not reply to Lookups made by arriving agents so no new subscriptions will be made. Since the Bye message is not sent the system agents do not know the entity is not available anymore.

When an agent tries to contact the crashed agent it will know that the entity is no longer available. Agents that were subscribed will know that the entity is no longer available when they try to renew their subscriptions.

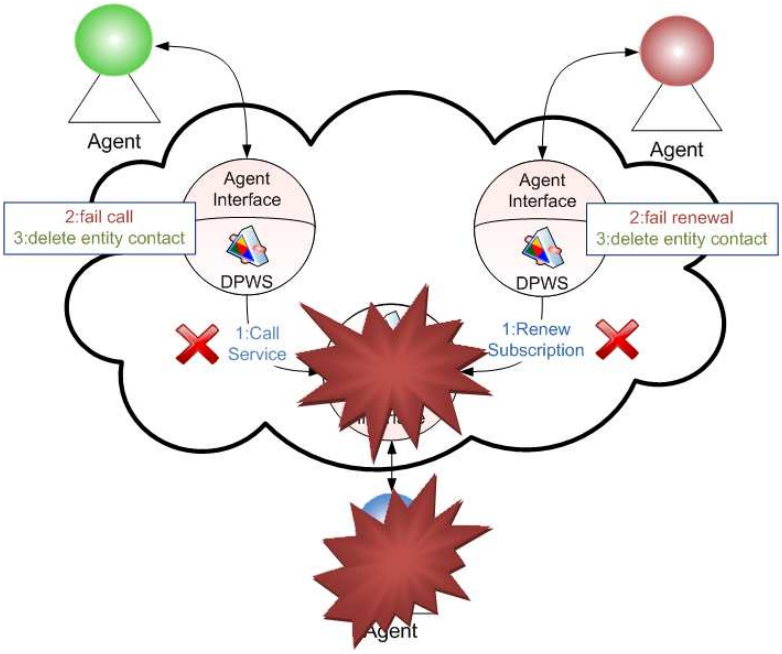


**Figure 3-27 - Entity crash**

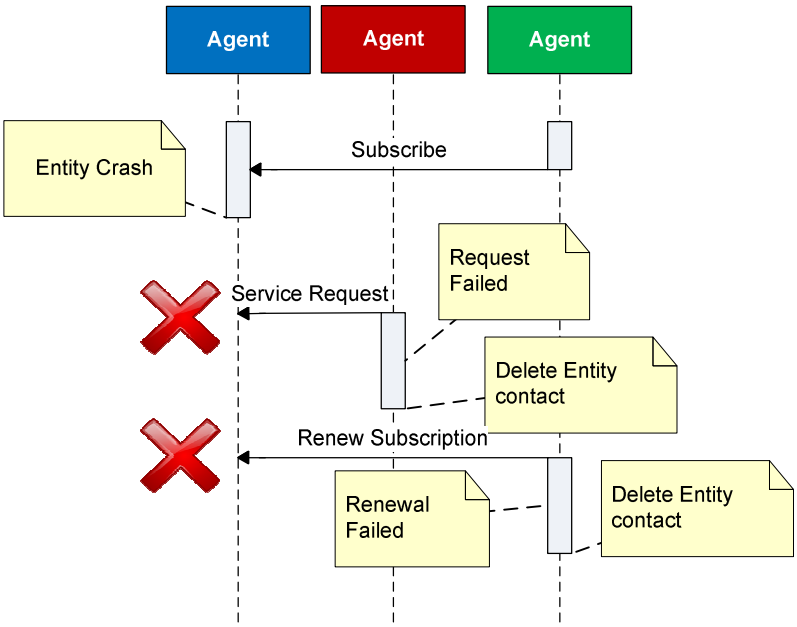


**Figure 3-28 - Entity Crash Sequence**

### 3.3.3.7 SERVICE USE

Most DPWS entities expose services to the network. These services can be of three types: request, request/response and event.

- The request type is called by an agent and executed by the service with no response. This service type is usually used to request an action.
- The request/response is called by an agent, executed by the service and a response is returned to the agent. This service is usually used to request information such as the status of a machine or process.
- The event type is initialized by the service and every previously subscribed agent will receive the message. The service is provided to subscribers. If an agent wants announcements from an entity it subscribes for that entity events. When the event occurs the agent will receive a message.

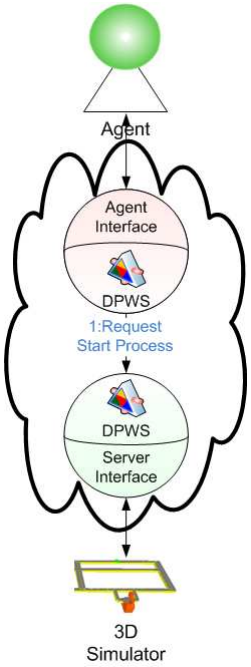


**Figure 3-29 - Service Request**

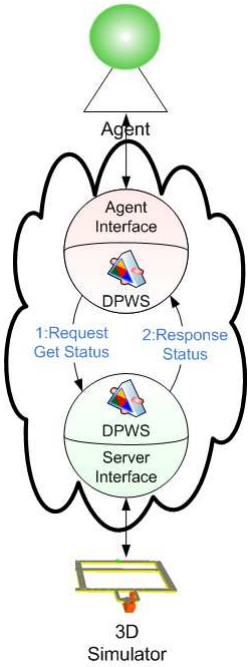


**Figure 3-30 - Service Request Response**

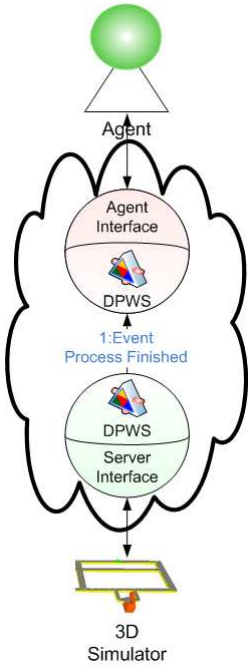


**Figure 3-31 - Service Event**

## 3.4 DPWS MIDDLEWARE ARCHITECTURE

The DPWS Middleware is layer between the Entity related code and the DPWS Communication interface (Figure 3-3).

This extra layer works as a wrapper around the DPWS Stack that filters it and bridges all communication to the Entity.

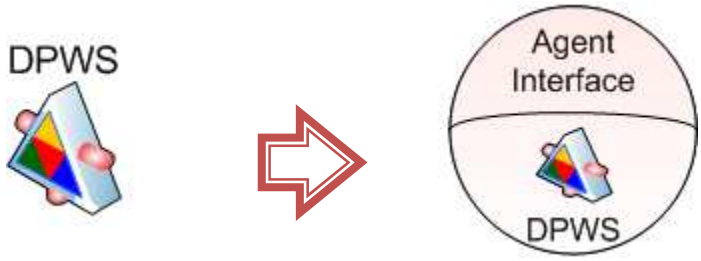


**Figure 3-32 - DPWS Wrapper**

The wrapper forwards all service requests and events to the entity after decoding them and building the agent friendly object. It manages known entities and services addresses to easy communication from the agent. Subscriptions are managed and maintained in the middleware. Discovery services provided by the DPWS Stack are combined in the middleware so that the agent only receives relevant information when the discovery is complete. Only when all metadata is retrieved does the agent receive the new entity name and type. The middleware architecture is presented below.

### 3.4.1 OVERVIEW

After the code generation by the DPWS Toolkit a simplified DPWS Middleware can be implemented for the entity. This interface must be able to translate the names of other entities to their addresses as well as translate the messages received and given to agent understandable information. It must warn the agent whenever a new entity arrives or leaves the system. Subscription management is also a task the Interface must support.

The necessary files to implement the Middleware with the following architecture could be made by a generator given some configuration.

The following figure illustrates the proposed DPWS Middleware approach.

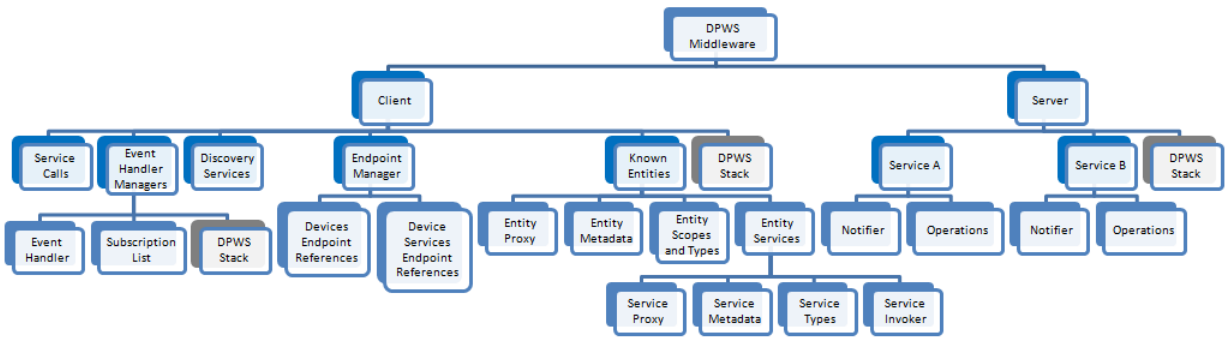


**Figure 3-33 - DPWS Middleware Architecture**

### 3.4.2 AGENT

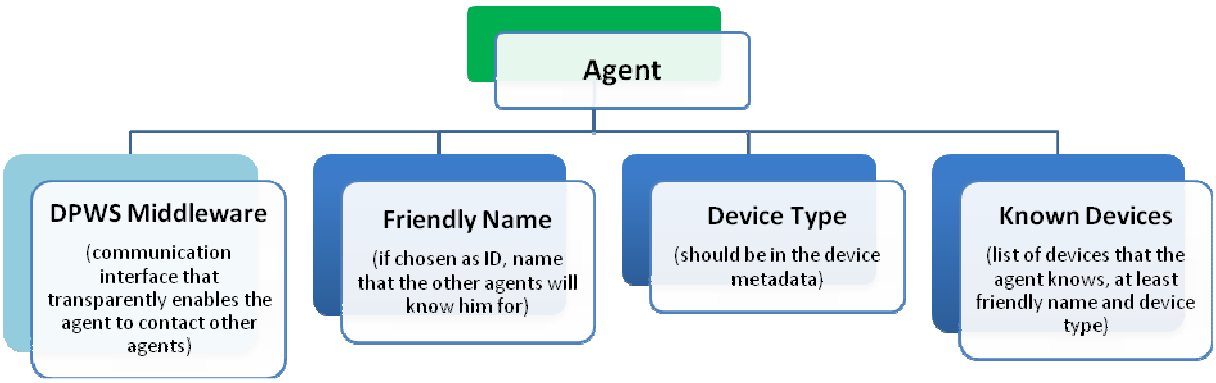


**Figure 3-34 - Base Agent Architecture**

Every Agent in the system will have an identification ID (a friendly name) that will be his identification to other agents, and an Entity Type so that they will know what kind of entity they are contacting. The agent will also have a known entities list that he will use when he wants to contact another entity in the system. Each entity is a combination of, at least, the entity ID (its friendly name) and its type. Other information that can be significant to the system, like topology, may be added.

Topology and other relevant data should be included in the device metadata but, currently, there is not a possibility to add custom metadata to the Stack.

The agent will known which type of services a certain type can provide and search the list for entities with the required type. With the friendly name of the entity to contact, the agent can now use its services through the DPWS Middleware. Giving the Entity name and the action to call to the DPWS Middleware is enough to make the contact.

### 3.4.3 DPWS MIDDLEWARE

The DPWS Middleware is composed of a Client and Server (Figure 3-5). The Client and Server parts of the Middleware are a higher level of the client and server made available in the DPWS Stack.

The Middleware Client has all necessary information to contact all known entities and keeps subscriptions. All Discovery services are provided by the Client. Events previously subscribed to are sent to the Client that forwards them to the agent.

The Middleware Server exposes the agent services to the network. The higher layer of the Server simply eases the configuration process and bridges the exposed services requests to the agent.

Both, the Middleware Client and Server, provide an easier configuration of the DPWS Stack communication interface.

The Middleware has mechanisms to make one use the other, like having the client warn the server that an entity has left the system without warning when it can't renew a subscription. The Server does not have the system's entities information and so it is never capable of telling the agent who requested the service by the name. The DPWS Middleware higher level is the intermediary between the Client and the Server. It will receive the Request from the Server, with the requestor address, and ask the Client the name and type of the requesting entity. The Request is then passed to the Agent with the request data as well as the information of the requesting entity.

In the following diagrams actions done by the DPWS stack out of the box, without knowledge of the controlling entity are shown in grey boxes whereas actions that make use of the middleware and reach the entity are shown in blue.

### 3.4.4 SERVER

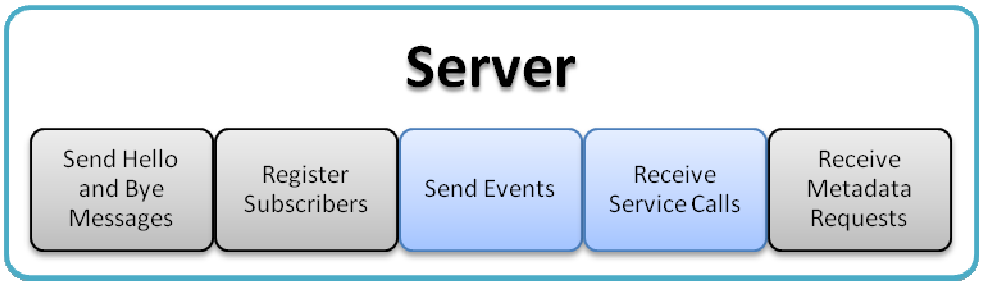


**Figure 3-35 - Server Services**

The Middleware Server is responsible for, sending Hello and Bye messages to the system, registering Subscribers for the agent services, sending Events and warn the entity that its services have been requested. All Services provided by the agent, as well as agent metadata are available to the system through the Server.

The DPWS Stack has automated the sending of Hello and Bye messages as well as subscriber registration. The device and services metadata are also configured at development time and the requests and responses regarding metadata are handled by the DPWS Stack.

The Middleware server simplifies implementation by providing an easier configuration, translates requests made to the agent to agent processable objects and sends events by agent request.

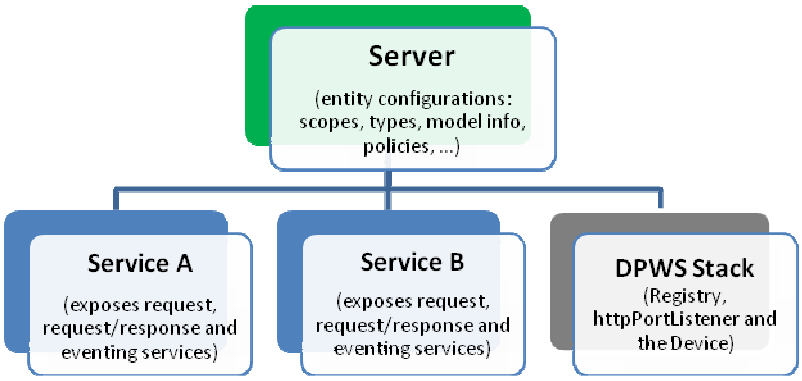


**Figure 3-36 – Middleware Server Architecture**

The server exposes to the agents in the system the entities services and information. When an entity enters the system the entity server announces its arrival by sending a Hello message to all agents already running. By receiving this message every agent will now be able to request its metadata and have access to this entity services and information. When it leaves the system it will send a Bye message signaling that its services will no longer be available.

The services exposed by the Server should be divided in service types, a service per entity type that will use it. Each service will have its own requests, request responses and events for de designed type.

#### 3.4.4.1 SERVICES

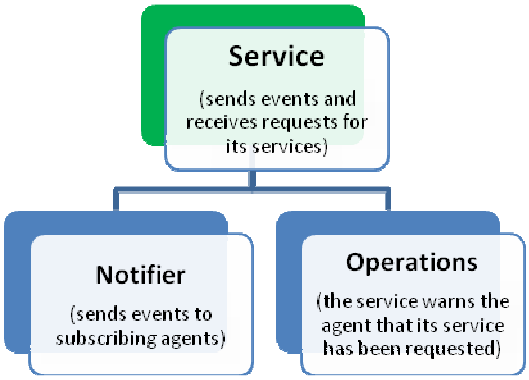


**Figure 3-37 - Services Architecture**

The Service is responsible for sending events through the Notifier by agent order, and announcing the service call to the agent. If a request response operation type was called it waits for the agent answer.

Services in the DPWS Stack have to be implemented by the developer on the generated class. Middleware Services can be generated as they translate the request to an object that is sent to the agent. With the request information the agent can then provide the service.

### 3.4.5 CLIENT

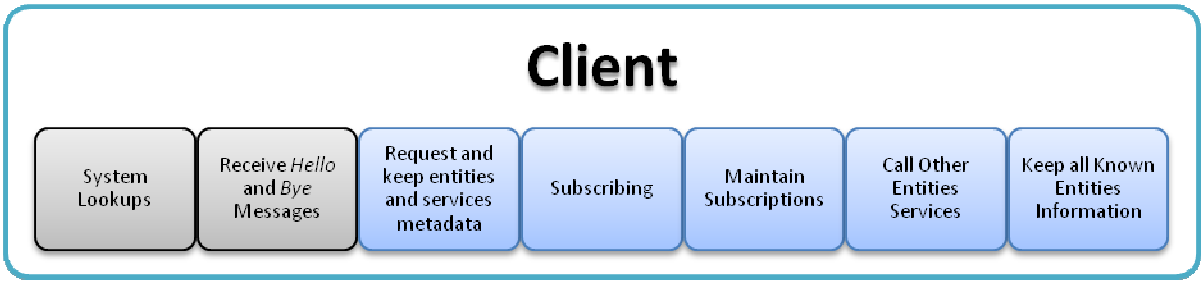


**Figure 3-38 - Client Services**

The Client is responsible for system Lookups, subscribe to other entities services, maintain subscriptions, calling other agent services, receiving Hello and Bye Messages, requesting metadata from other entities and keeping a list of known devices with their information sorted by their friendly name. The Information kept by the Client on other entities consists of entity address, services exposed by that entity, their addresses and entity and services metadata.

The Client is the interface of the system for the agent. It warns the agent about arrivals and departures on the system as well as crashes when it tries to renew a subscription with an entity that is no longer available. Whenever an entity leaves the system the Client must cancel all subscriptions to that entity, and when the leaving entity is an agent it must warn the server to cancel subscriptions in case it did not unsubscribe first.

The Middleware client offers a number of services to the agent that the DPWS Stack client does not. Besides easing configuration and implementation the Middleware client filters DPWS related information so that the agent only receives relevant data. The Middleware Client has a higher level in every part of communication as it processes all information that comes from and to the DPWS Stack client.

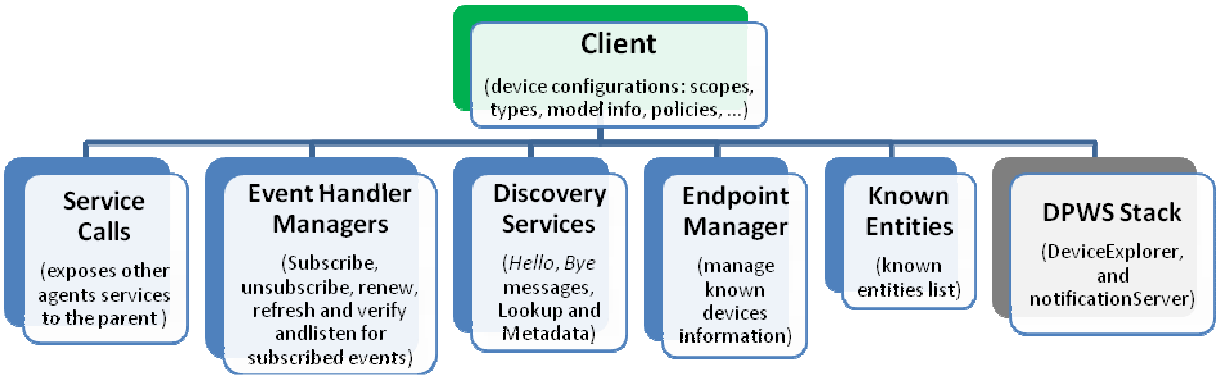


**Figure 3-39 – Middleware Client Architecture**

The Client is configured with the scopes and types that it should listen and look up for. It has one Event Handler Manager, per service type that it can call, to manage Subscriptions. Subscription management includes subscribing, unsubscribing, renewing, getting the subscription status, verify if subscribed and maintain (auto-renew) subscriptions. Subscriptions can be made to an entire service (port type), or to a group of available events that the port type implements. Also, per service type, it has an Invoker with which it will call the system entities request and request response services.

#### 3.4.5.1 EVENT HANDLER MANAGER

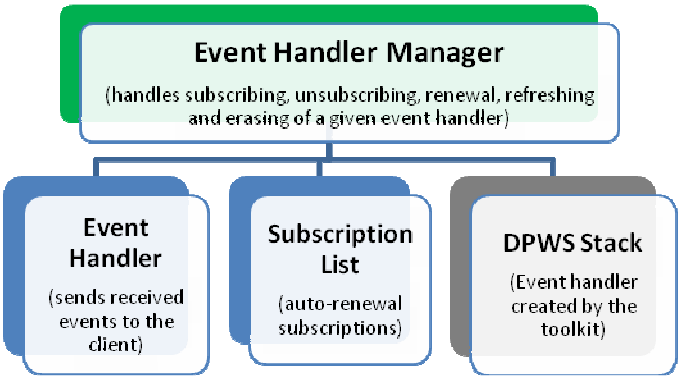


**Figure 3-40 - Event Handler Manager Architecture**

The Middleware Client has an Event Handler Manager for each Service type it can receive events from. It includes an Event Handler that receives the events the agents previously subscribed to and translates them to the agent. It keeps a list of made subscriptions with related information to be able to check if still subscribed and operate on them with available DPWS Stack operations (renew, cancel, get status).

The subscription list has Complex Subscriptions that automatically renew themselves when they are about to expire. In case they fail to renew (the Entity that provides the service might have left or crashed) they will notify the client, which will notify the agent that will decide what to do with that information.

The DPWS Stack generates an interface to implement the event handler. The implementation and extra features belong to the Middleware.

#### 3.4.5.2 ENDPOINT MANAGER

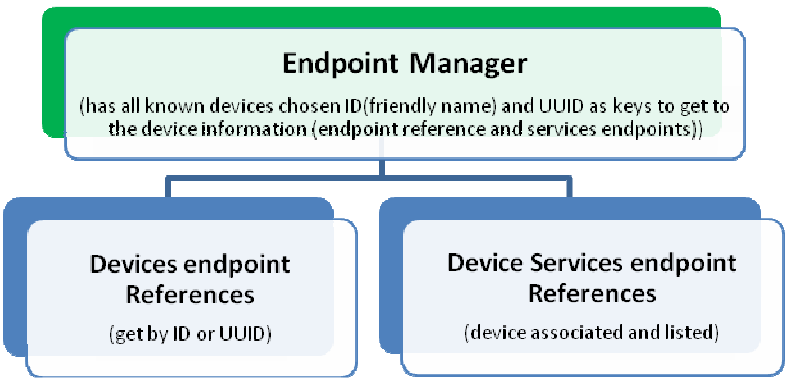


**Figure 3-41 - Endpoint Manager Architecture**

The Endpoint manager has all known Entities entries with every Entity endpoint and their services endpoints. Through this manager the client will be able to search for devices and their services when given a call is requested by the agent with the device name it wants to send to.

The Endpoint manager is also used by the Middleware server to translate the UUID of requesting agents into friendly names that the agent knows.

This is a complete new service the Middleware provides to the agent.

#### 3.4.5.3 KNOWN ENTITIES

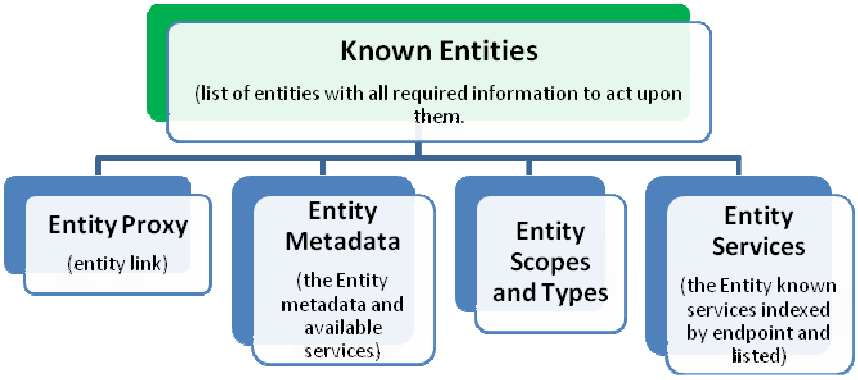


**Figure 3-42 - Known Entities Architecture**

The Entities list has a list of Entities along with information on how to contact them. The structure includes the entity proxy, its metadata, scopes and types and a list of services it provides along with their address and metadata.

This structure belongs to the Middleware Client and has all necessary information to contact any known entity. It is a completely new structure added do the Middleware as it was indispensable for the proposed architecture.

##### *3.4.5.3.1 ENTITY SERVICES*

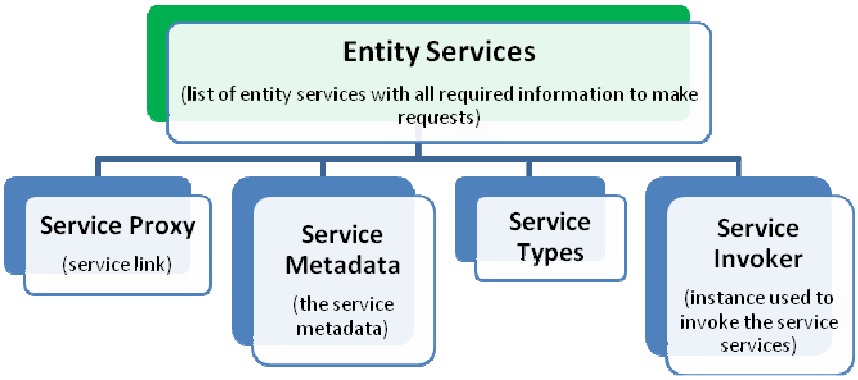


**Figure 3-43 - Entity Services Architecture**

The Entity Services list has a collection of services of a given Entity along with its metadata, types, proxy and the means to invoke its services (the invoker).

This structure is part of the structure presented above and has the same use. It is part of the structure which gives the Middleware the means to contact other known entities. As the above, this structure was deemed necessary for the Middleware.

## 3.5 IMPLEMENTATION

In this section the implementation of an agent control system with DPWS communication will be covered, focusing on the implementation of the DPWS Middleware.

### 3.5.1 HOW TO MAKE A DPWS SYSTEM

To implement an agent control system communicating via DPWS, the following steps must be completed:

- Design the services descriptors, the WSDL files;
- Generate the DPWS Stack related code with the given generator;
- Implement the DPWS Middleware Layer following the above architecture;
- Integrate the implemented communication layer with the system entities.

The Middleware Layer implementation step can be avoided by creating another generator. Given some configuration it is possible to create a generator to create all

Middleware necessary files. With the generator it would be possible to design the services descriptor and generate all the communication layer necessary code.

Due to the discovery capabilities of the DPWS protocol stack, the order in which these run is not important. After these steps the system will be running. Every agent will know who is on the system and ready to use their services in order to reach their goals.

#### 3.5.1.1 THE WSDL SERVICE DESCRIPTOR

To make the DPWS Middleware, a WSDL file must be written so that a DPWS Toolkit can generate the code needed for the interface. For this matter a WSDL generator Tool was implemented and is explained in chapter 4.10.

The WSDL has a structure defining the Service that the device will implement. The Service is the main node in the structure. Everything that a device exposes is through this Service.

The structure presented here is the W3C WSDL standard [61] with some minor changes made by the Stack developers.

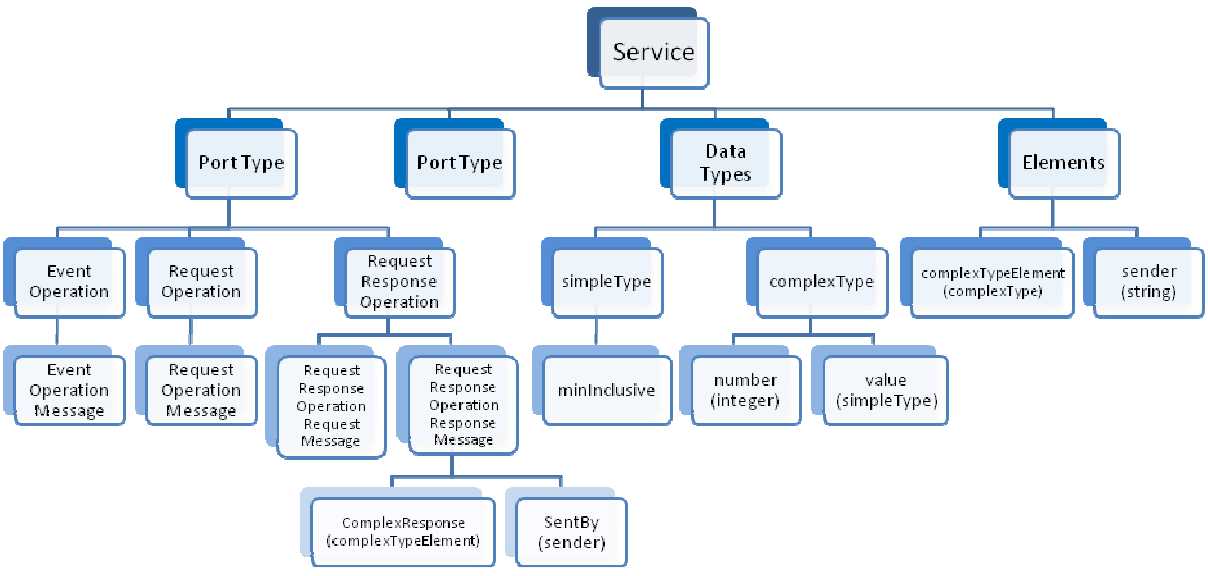


**Figure 3-44 - Service Description Tree**

The Service has one or more Port Types. It is through these Port Types that services can be made for a specific agent type (ex: a port type for machine agents, a port type for a network debugger, etc). Some services should only be available for a certain type of agent

given that other types do not need or should not use them. This simplifies, for example, subscriptions, as agents subscribe only for events that they can use. Implementation is also simplified as each agent implements only the services that they will use.

Each Port Type can have various operations that can be called (request and request response operations) or subscribed to (event operations).

Operations, as explained in 3.3.3.7, can be of type Request, Request Response or Event. The Request Operation receives a message that can be empty, the Request Response receives a message that can also be empty but responds with a non empty message and the Event Operation sends a message that can also be empty. These Messages are what will be exchanged between entities.

Each message description can have Elements. These Elements also have to be declared in the WSDL file. Elements have a name and a type. The type of the Element can be a normal type like integer, string, token, short and other XML Schema types, or a declared Data Type.

Data Types can be Simple or Complex. Simple Data Types have a name, a XML Schema type and may have restrictions such as minimum value, maximum value, possible values and many more. Complex Data Types have a name and a list of elements each with its own type, be it a XML Schema type of a Simple Data Type [62].

The Service should be as complete as possible so that each entity can provide a number or functionalities to the system. Events should be widely used so that the system can easily be monitored and diagnosed when something unexpected happens. A complete set of services enables the system to accept new entity types adding new functionality without reprogramming every other entity.

#### 3.5.1.2 THE GENERATED CODE

Each one of the available toolkits create by Schneider Electric (the C and Java Stacks) has a code generator for the needed code to start implementing the DPWS interface.

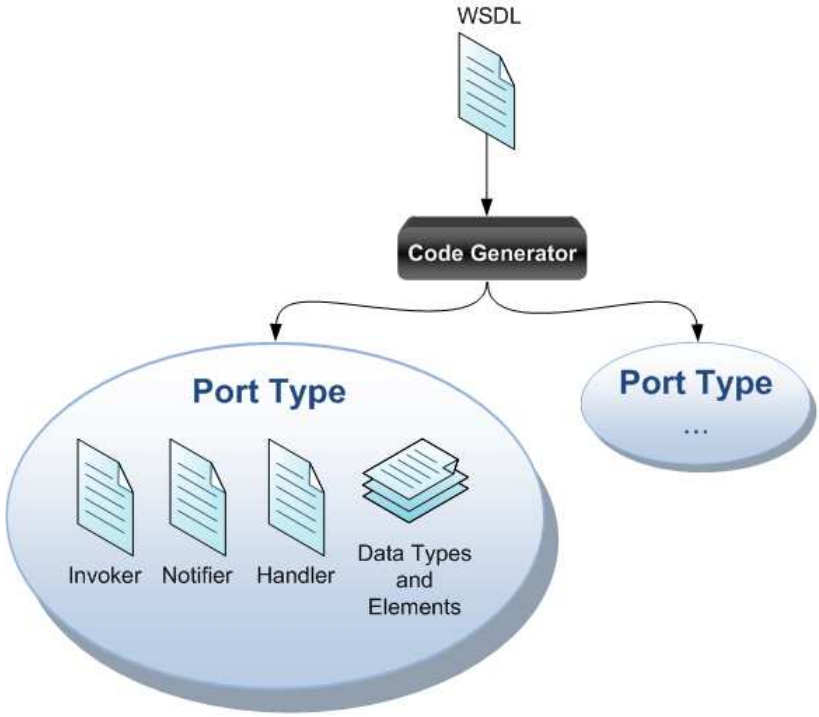


**Figure 3-45 - Generated Code**

The Java DPWS generator reads the WSDL file and generates the classes and interfaces that will be used to implement the service.

It generates:

- the classes of each Data Type and Element declared in the WSDL file;
- one Invoker per Port Type that enables clients to invoke this Port Type services;
- one Notifier per Port Type that has at least one Event Operation that enables the device to send events to subscribed agents;
- a Handler Interface, per Port Type with Events, that enables the agent to handle received events. This interface is to be implemented by every entity that can subscribe to these service events.

With the generated files the Middleware can be developed or generated.

#### 3.5.1.3 PROGRAM THE DPWS MIDDLEWARE

After the necessary generated code, provided by the DPWS Stack generator, the Middleware is ready to be implemented.

Following the Java implementation that was made, the Middleware has two development phases. The first phase includes the necessary Middleware files that only have to be implemented once, the Middleware base structure.

This structure includes:

- Middleware Client;
- Middleware Server;
- Known Entities information list;
- Endpoint Manager;
- Subscription Manager.

The second phase is the entity specific implementation of the Middleware that needs the generated code from the DPWS Stack generator.

This includes:

- DPWS Middleware (top structure to be used by the entity)
- Entity specific Client
- Entity specific Server
- Services implementations
- Event Handlers implementations

The first implementation phase only happens once since it is generic code. The second implementation phase is quite mechanized, with only a few decision needed. This phase could be skipped with the implementation of a configurable generator so that the complete DPWS Communication interface of any entity depended only on designing the Service descriptor, the WSDL file.

#### 3.5.1.4 INTEGRATING EVERY SYSTEM ENTITY

After the DPWS Interfaces have been implemented they are ready to be integrated with the agents. The agent needs a friendly name for which it will be known throughout the system, an entity type so that other entities know what services it exposes and a list of known entities with their name, type and other relevant data. With this information the agent is now ready to use services provided by other entities to fulfill its goals. This method is fully detailed in section 3.4.

Every service method will be called like a local method by the agent. The whole system will be contactable transparently through the complete DPWS Interface as if all requests were made locally.

### 3.5.2 RUNNING THE SYSTEM

Starting a DPWS system is simple as there is no order to launch the entities. The Discovery protocols guarantee that every entity will find every other entity regardless of running order. The complete DPWS Interface will start with the agent and the discovery protocols will find the other entities in the system through Look ups, Hello and Bye messages.

When the minimum necessary entities, to run the production system, have been started, the system can operate.

During system run, at whatever time any entity can start and, in case it is an agent, use the systems services. The running agents, if previously prepared, can also use the new entity services.

# 4 Case Study

## 4.1 OVERVIEW

To demonstrate the proposed architecture with DPWS communication a Demonstrator was build. The Demonstrator includes a 3D model of a simplified plant, a DPWS Interface with Server Middleware per 3D entity, Agents that control the system, a Configuration Tool, a Production Manager Tool, a Communication Log and a Database. Each of the Entities in the Demonstrator communicates with the others via a DPWS Interface that implements the proposed concept.

The Demonstrator was entirely built to demonstrate the Middleware operating in all presented entity types. The only blocks that weren't implemented were the DPWS Web services that were generated by the DPWS Stack given the, Demonstrator specific designed, service descriptors, the WSDL.

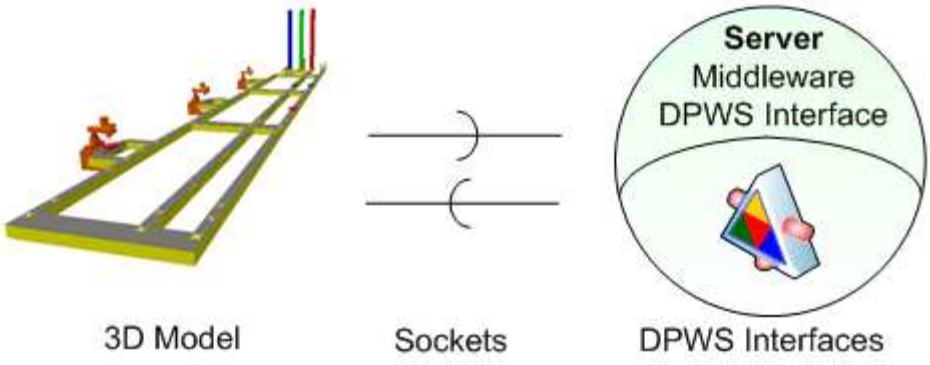


**Figure 4-1 - 3D model Communication Model**

The 3D model was designed in DELMIA Quest to mimic a system which description was provided by Schneider Electric. The 3D model has 3 workpiece types to produce with different colors. After the 3D model was designed the basic actions had to be programmed into the model using the available language SCL (Simulation Control Language). The 3D model has a total of 103 decision points that had to be programmed individually. It includes 9 entities each with its own SCL initialization code. Each of the implemented entities communicates with their DPWS interface via sockets implemented in C. Each has a client and a server that send and receive messages to and from the server and client implemented in the entities DPWS Interfaces.

The rest of the system was all implemented in Java using the Schneider Electric DPWS Java Stack.

The DPWS Interfaces for the 3D model were implemented following the presented Server Middleware concept to represent the virtual world in the system. These interfaces could be representing a real machine instead of a virtual one without changing the system.

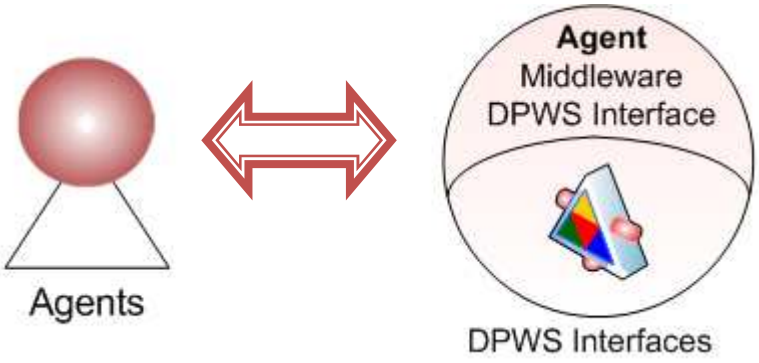


**Figure 4-2 - Agent Communication Model**

With the 3D model complete the agents had a virtual world to command. All agents were implemented with the Agent Middleware concept. Communication between the agent and the Middleware is local since the Middleware is an object.

To fully demonstrate the Middleware some other entities were developed: the Configuration Tool, the Production Manager, the Database and the Communication Log.

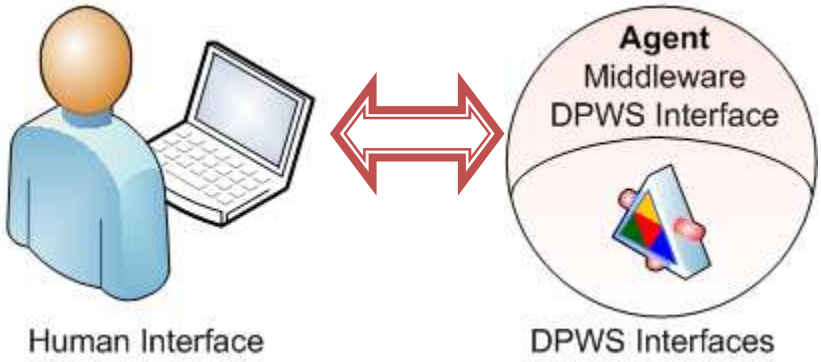


**Figure 4-3 - Configuration Tool and Production Manager Communication Model**

The Configuration Tool and the Production Manager are Human Interface applications with the Agent Middleware. The Configuration Tool enables the configuration of all agents and the simulator. The Production Manager enables the design of production plans by workpiece type and sends production orders into the system.

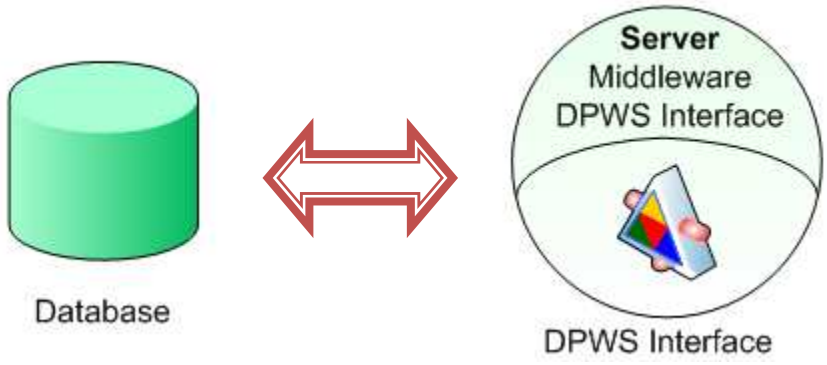


**Figure 4-4 - Database Communication Model**

The Database is represented in the system by a Server Middleware. All system production data has its persistency in the database. Whenever an entity crashes the production can continue because all necessary data is secure.

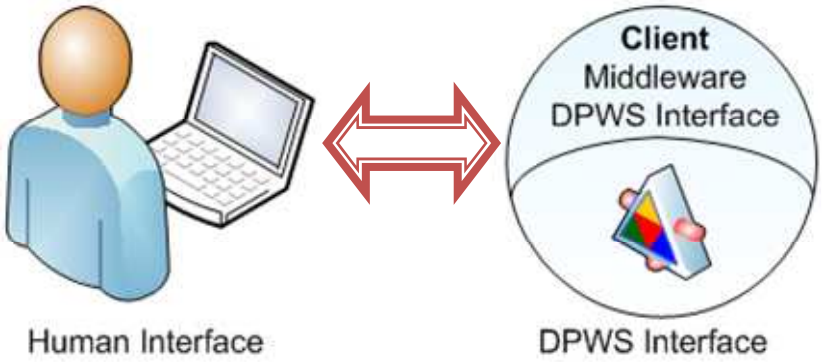


**Figure 4-5 - Communication Log Communication Model**

The Communication Log is a Monitor entity with the Client Middleware. Every entity in the system exposes a Standard Log Event Service to which the Communication Log can subscribe. This entity can monitor all communication made in the system by subscribing to this service.

All Entities are necessary to be running for the system to start but the Communication Log, which can be run at any time or not run at all.

## 4.2 3D MODEL

The 3D Model of the Demonstrator is a simplified model of a motor production plant designed by **DaimlerChrysler**. It was built in DELMIA QUEST software and programmed using the available language SCL (Simulation Control Language).

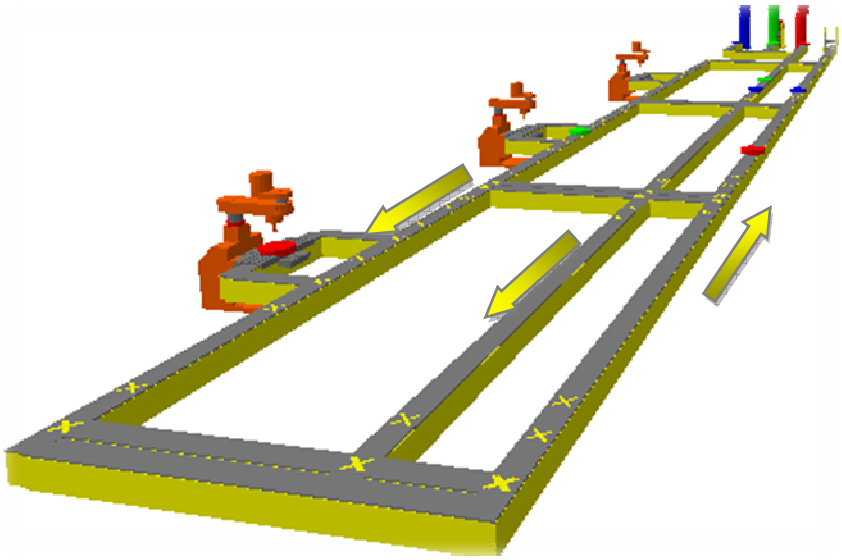

**Figure 4-6 - Demonstrator 3D Model**

The model includes a warehouse, a loader, 4 shift tables, 3 machines and an unloader. Since the model warehouse has only three workpiece types and very basic functionality it was logically grouped with the loader.

Each model part has a series of decision points where, when a workpiece arrives, the associated code is executed.

Each model part communicates with the outside DPWS interface, through sockets. One server socket and one client socket connection per part. After every one of the 9 socket servers and clients are connected the 3D model is ready to start operating and accepting orders.

### 4.2.1 WORKPIECES

Workpieces are the moving parts of the 3D model. There three types of workpieces, each represented by a short colored cylinder. There is a red, blue and green workpiece types.

Each workpiece has an ID associated. This ID is set when the workpiece is initialized at the loader. The ID is given by the Loader Agent. This ID can be read at any decision point in the system. It is then sent to the agents, when necessary, to keep track of the workpieces location.

### 4.2.2 DECISION POINTS

Decision points are part of the 3D model. Each decision point has SCL code associated that runs when a workpiece reaches it. Some decision points read the workpiece ID when it is upon them and send it alongside with a message to the agent. Other decision points simply route the workpiece to another conveyor by following a previously made decision. An example of this is the shift table that sends a message to the shift table agent when a workpiece arrives. It then receives the shifting order and shifts the workpiece from an entrance to an exit without exchanging more messages with the agent even though there are several routing decisions in the shift. This procedure makes the workpiece pass through various decision points that act upon the workpiece by reading previously set local variables.

There are 9 decision points in the system that have the initialization code necessary to make the socket connection with the DPWS interfaces. All of this decision points have a socket client and server to communicate through which strings are exchanged.

### 4.2.3 LOADER MODEL

The loader model receives workpieces from the warehouse and releases them into the system. It is composed by a single decision point that accepts the release order from the agent. Any entity that enters the system has to go through this entity. Workpieces are initialized (they are assigned an ID) in the loader decision point.

The warehouse model has 3 workpiece types. Each of these types can be called into the system. Each type is identified by their color: red, blue or green. This model is logically grouped with the loader therefore the workpieces are called into the system by a loader agent.

The loader and warehouse models are both controlled by a loader agent.

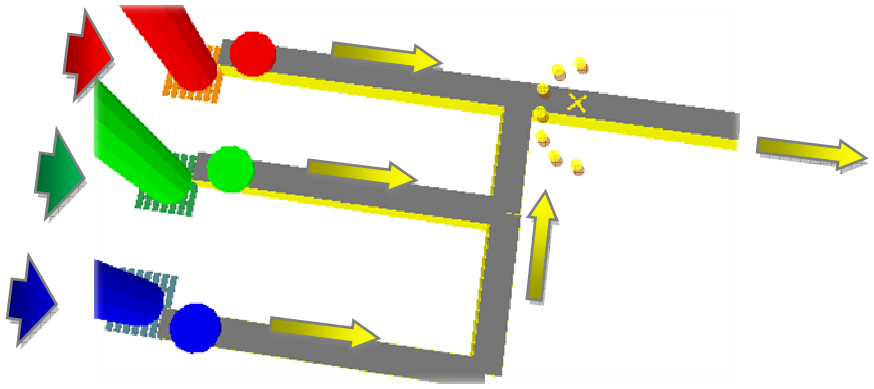

**Figure 4-7 - Loader and Warehouse 3D Model**

The warehouse and loader models accepts commands to manage the entry of workpieces into the system by type.

The loader model is represented in the agent system via its DPWS Interface. The DPWS Interface exposes the loader and warehouse functionalities to the Loader Agent that controls them. With 1 loader and warehouse models in the demonstrator the system needs 1 loader agent.

### 4.2.4 SHIFTTABLE MODEL

There are 4 shift table models in the Demonstrator. The shift table models are the means of transportation of each workpiece to any part of the system. Their functionality is to shift workpieces from each of the system main 3 conveyors to another.

The shift table model has a number of entries and exits. When a workpiece arrives at an entrance this is announced. When a decision has been made the workpiece enters the shift table, goes through a number of decision points until it has reached an exit. Only one workpiece can enter the shift table at any moment.

Depending on the shift table model it can have up to 21 decision points.

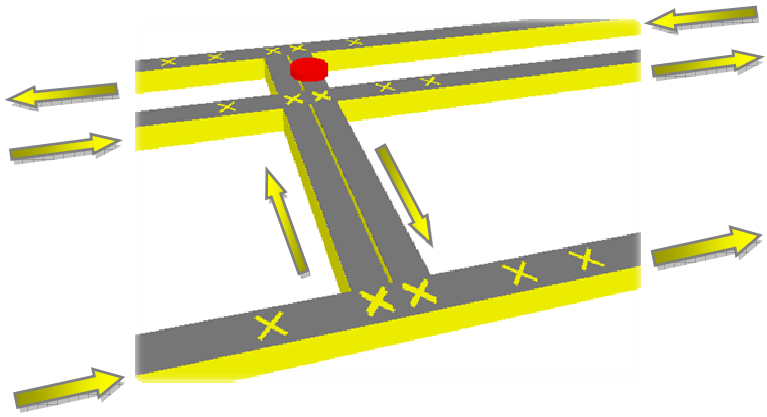

**Figure 4-8 - ShiftTable 3D Model**

The shift table model is composed of two conveyors perpendicular to the three main conveyors of the system. These two conveyors can shift workpieces up and down between the main ones.

The shift tables in the system separate the other entities topologically. Between every two shift tables there is a machine in the lowest conveyor. Connected to the first shift table are the loader and unloader models.

Workpieces travel through the system in the middle and top conveyors to reach their contracted entity. The lower conveyor is only visited by workpieces that are going to be processed by the adjacent machine.

Every shift table model accepts commands to shift workpieces from one conveyor to another and announce relevant events.

The shift table model is represented in the agent system via its DPWS Interface. The DPWS Interface exposes the shift table model to the ShiftTable Agent that controls it. With 4 shift table models in the demonstrator the system needs 4 shift table agents.

### 4.2.5 MACHINE MODEL

There are 3 machine models in the Demonstrator. The machine models simulate the processing of workpieces in the system. Their functionality is to process workpieces and to manage the conveyor system associated with them, so that unwanted workpieces may pass and wanted processed.

Each machine in the system has 8 decision points

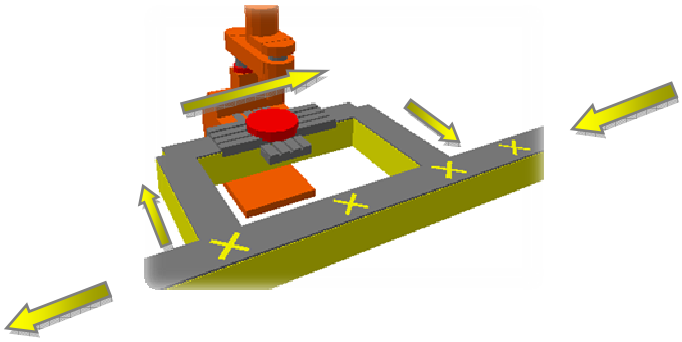

**Figure 4-9 - Machine 3D Model**

The machine model is composed of a small conveyor system that allows it to convey workpieces to the machine and out, or just let them pass, as needed. The model has a machine and once a workpiece reaches the processing position it will stay there for some time until the processing has been finished.

The machine can have a maximum of four workpieces in its system: one at the exit, one processing, one waiting to be processed and one waiting to enter the machine conveyor system. All these can be managed by the machine model by receiving decision commands from the Machine Agent controlling it.

Every machine model accepts commands to manage the flow of workpieces in its conveyor system and announces relevant events to enable that management.

The machine model is represented in the agent system via its DPWS Interface. The DPWS Interface exposes the machine model to the Machine Agent that controls it. With 3 machine models in the demonstrator the system needs 3 machine agents.

### 4.2.6 UNLOADER MODEL

The unloader model receives workpieces from the system and sends them away. Its function is to expose or archive of completely processed workpieces.

**Figure 4-10 - Unloader 3D Model**

It is composed by a single decision point that accepts the release order. It is located at the end of one of the main conveyors. At the end of the unloader there is a sink where workpieces disappear.

The unloader model is represented in the agent system via its DPWS Interface. The DPWS Interface exposes the unloader model to the Unloader Agent that controls it. With 1 unloader model in the demonstrator the system needs 1 unloader agent.

### 4.2.7 MODELS DPWS INTERFACE

Every one of the 9 models in the demonstrator has its own representation in the system. There are 9 Server Entity DPWS Interfaces to represent the 3D model. Each one of these entities translates the string sent by and to the models into DPWS services so that agents can find and control the model. Events sent by the model are translated into DPWS messages and commands sent to the model are translated into strings that are sent through the designated socket.

## 4.3 AGENTS

The Agents in the demonstrator control the production system. In order to make the best decision they communicate between themselves to get the needed information.

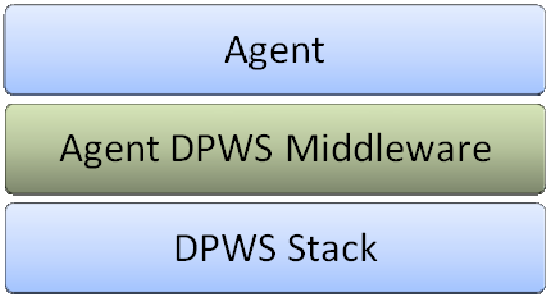


**Figure 4-11 – Agent Architecture**

Every agent in the system has the agent Middleware which is the agent interface to the rest of the system.

The demonstrator has 5 agent types: loader agents, shift table agents, machine agents, unloader agents and workpiece agents. Each agent type is responsible for something in the 3D environment.

Some decisions that an agent has to make need more information than the one it has. Therefore the agent must gather the required information from the other entities in the system. There are two forms of information gathering.

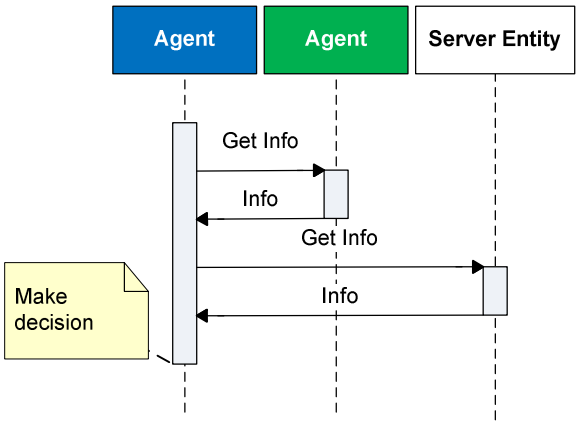


**Figure 4-12 - Decision Making Direct Information Gathering**

The simplest one is having the agent directly request the needed information from another entity by calling a service. The entity then responds with the requested information. This method requires that the entity provides that service.

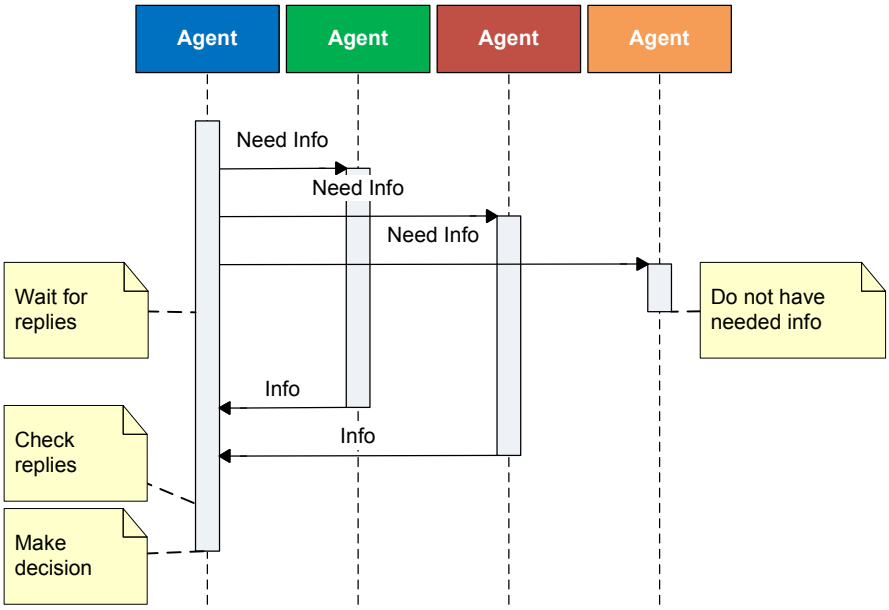


**Figure 4-13 - Decision Making Indirect Information Gathering**

The other method has the agent advertising its need for information and waiting for responses from other entities that may have it. After a pre defined time the agent will check the received information and make its decision.

Every agent in the system has a service to enable a communication log. When a Communication Log entity joins the system it can choose which entities to include in the log and subscribe to them.

### 4.3.1 LOADER AGENT

The Loader Agent is responsible for the demonstrator warehouse and loader. Its task is to bring in the unprocessed workpieces from the warehouse to the system and introducing them to their responsible workpiece agents. The workpiece to workpiece agent assignment is made through advertisement. The loader agent advertises that it has a workpiece that needs representation and waits for workpiece agent responses. It then checks them and chooses the most suitable to represent the workpiece.

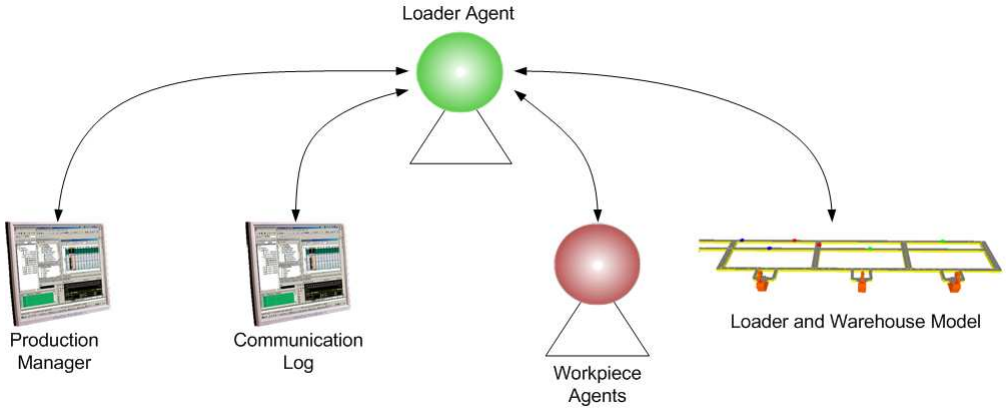


**Figure 4-14 - Loader Agent Interactions**

The Loader Agent accepts production orders from the Production Manager, calls workpieces from the warehouse, initializes and advertises them so that a workpiece agent will agree to take responsibility for that workpiece after which it releases it into the system.

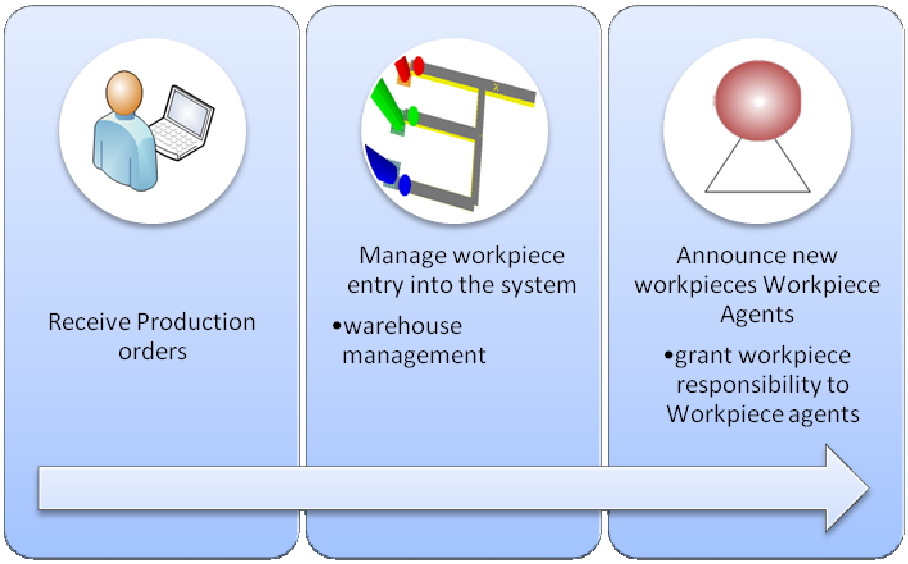


**Figure 4-15 - Loader Agent Tasks**

The warehouse in this demonstrator has three types of workpieces and the orders, made by the production manager, can be of any of these types. The order also includes the number of workpieces to produce and their base priority. In case there are many orders the Loader Agent must first dispatch the higher priority ones.

### 4.3.2 UNLOADER AGENT

The Unloader Agent is responsible for the unloader model. Its task is to unload workpieces from the system. Every workpiece that enters the system must leave through the unloader.

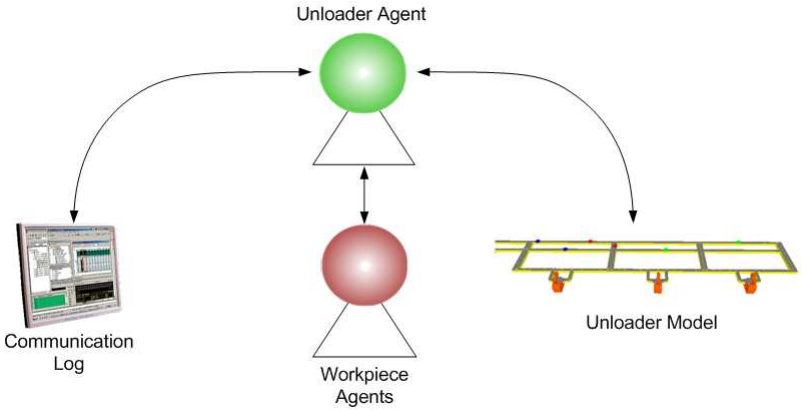


**Figure 4-16 - Unloader Agent Interactions**

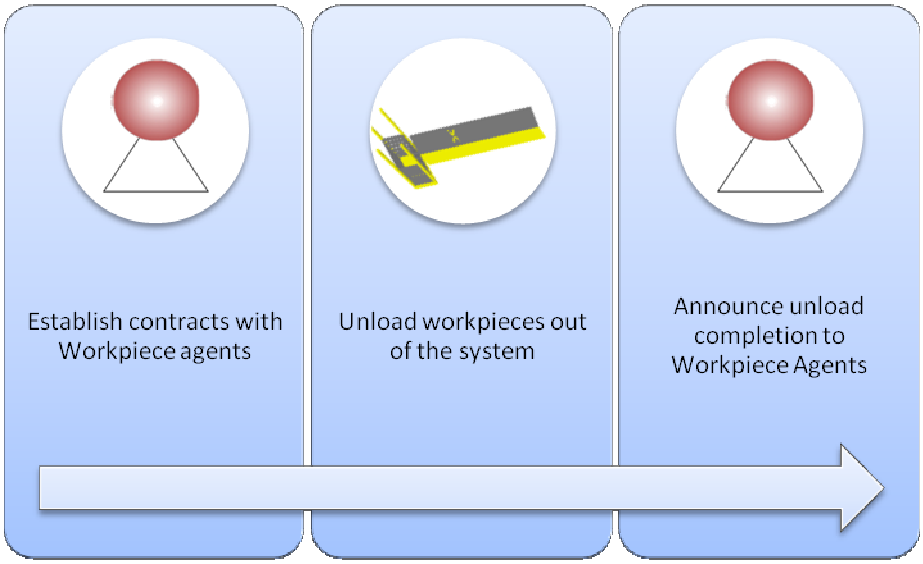


**Figure 4-17 - Unloader Agent Tasks**

It receives workpiece advertisement from the workpiece agent when they have been completely processed and after a contract is established the workpieces are transported to it. After the workpiece arrives it then warns the workpiece agent that finishes the contract it has with the workpiece, and the workpiece is unloaded from the system finishing its production cycle.

### 4.3.3 SHIFTTABLE AGENT

The ShiftTable Agents are responsible for shift table models. Their task is to route workpieces inside the system. They receive transportation requests and try to execute them when the workpieces in question arrive.

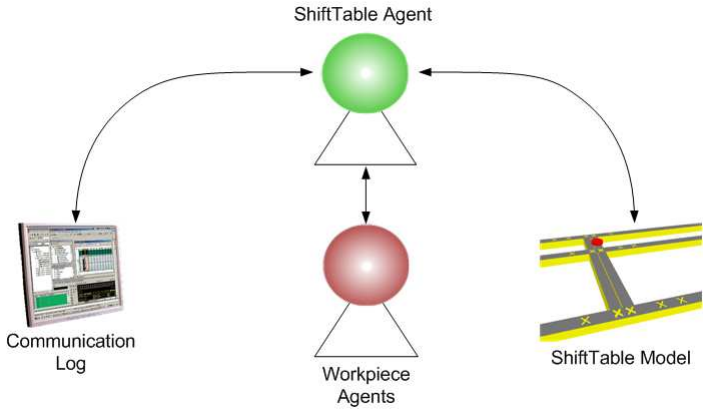


**Figure 4-18 - ShiftTable Agent Interactions**

The ShiftTable is composed by entrance and exit conveyors and a shifter that shifts workpieces from an entrance conveyor to an exit one. A workpiece can arrive in a conveyor and be shifted to another conveyor on another level. The shifter can shift workpieces up and down the three main conveyors of the system.

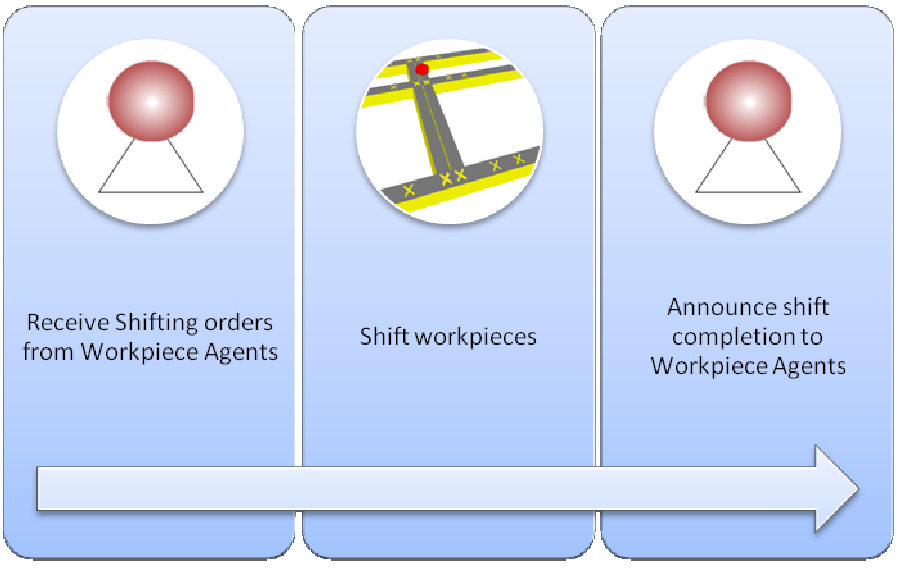


**Figure 4-19 – Shift Table Agent Tasks**

Whenever a workpiece arrives on an entrance, the shift table will warn the ShiftTable Agent and it will verify if there is a request for that workpiece. It will then try to execute it considering the present status of the shift table. After shifting, the shift table agent will signal

the responsible Workpiece agent that the workpiece has been shifted. The shift table also warns the agent every time an occupied exit becomes free.

### 4.3.4 MACHINE AGENT

The Machine Agent is responsible for a machine model. Its task is to process system workpieces according to their process plan.

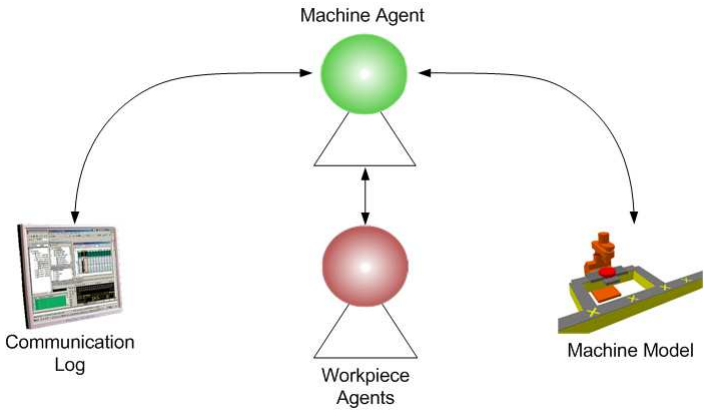


**Figure 4-20 - Machine Agent Interactions**

The workpiece is advertised by the workpiece agent and the machine agent responds. After the workpiece agent chooses the processing machine it will try to add the workpiece to the machine processing list. If acknowledged the workpiece is then transported to the machine.

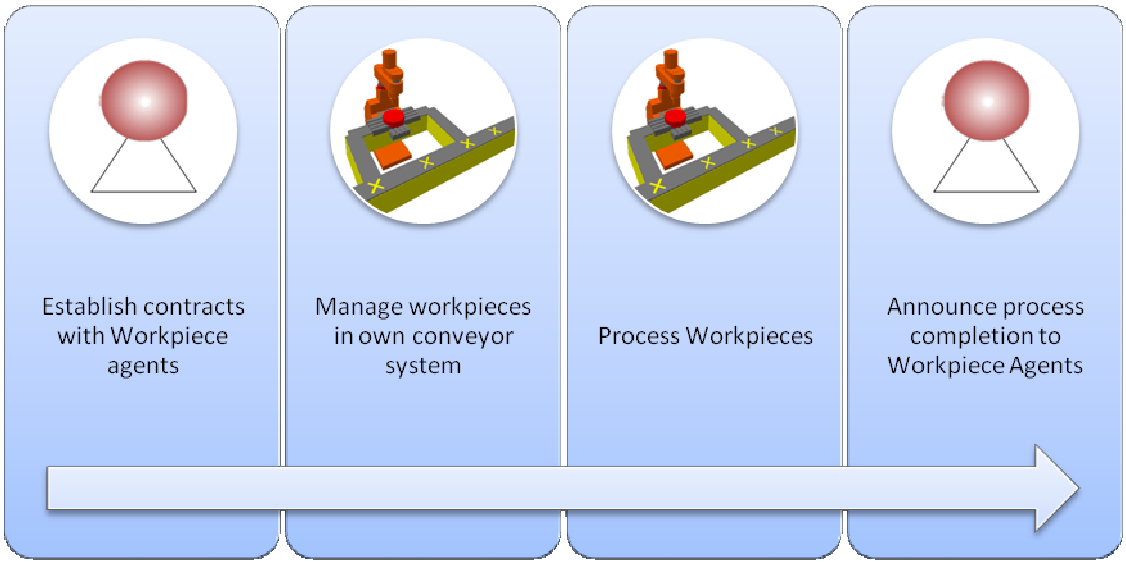


**Figure 4-21 - Machine Agent Tasks**

When it arrives, the machine will guide it to the processing position and process the workpiece as requested by the Workpiece Agent.

After the process is finished, or canceled, the machine agent will announce it to the responsible workpiece agent that will plan the next action. The workpiece is then released again into the system.

### 4.3.5 WORKPIECE AGENT

The Workpiece Agents are the center piece of the system. They have no hardware to control. They are responsible for the planning of production, transportation, loading, unloading, workpiece record creation and management of all workpieces in the system. The Workpiece Agent is the entity with more interactions on the system as it interacts with almost every other entity.

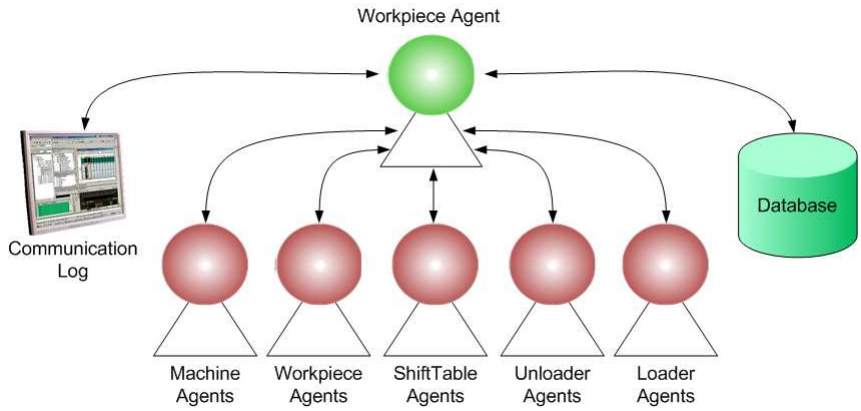


**Figure 4-22 - Workpiece Agent Interactions**

The first task of the workpiece Agent is to accept responsibility for the workpiece when it is entering the system. It then creates the workpiece record on the database and has the responsibility of keeping it updated throughout the process.

After the workpiece has been introduced into the system the workpiece agent must advertise it to machine agents in order to follow the workpiece process plan downloaded from the database. The agent will receive advertise responses, choose the best one and establish a contract with the chosen machine.

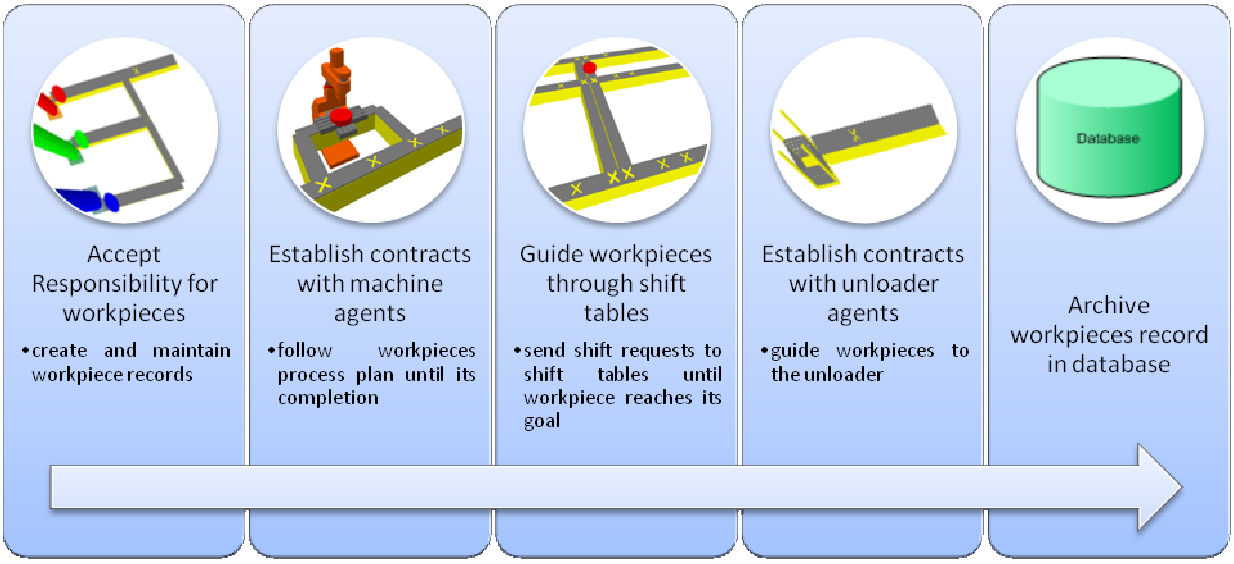


**Figure 4-23 - Workpiece Agent Tasks**

After the contract has been established it must then guide the workpiece through the shift tables until it reaches the machine. In this guidance the workpiece agent must announce the workpiece arrival, step by step, and request each shift necessary until the workpiece arrives at the machine. The machine processes the workpiece and announces it to the workpiece agent which updates the workpiece process status in the database.

This process is repeated until the workpiece is fully processed after which the workpiece agent will advertise the workpiece to available unloader agents. It then chooses the unloader agent, establish a contract with it and guide, once again, the workpiece to it. After the workpiece arrives at the unloader agent it will announce it to the workpiece agent which performs the last update of that workpiece and archives its process in the database. The workpiece is then released by the unloader and the process deleted from the workpiece agent.

All workpiece agents in the system must share the processing workpieces responsibility so that the control load is well distributed. When an unknown workpiece arrives in any place it will have to be represented. In case it is not, one of the workpiece agents must accept responsibility over it. This will happen especially if one of the workpiece agents crashes. The workpieces will have no representation and other workpiece agent must continue their, the lost workpieces, process. The, always updated, record in the database has this goal.

## 4.4 HUMAN INTERFACE

The Human Interface entity category includes all entities that have a User interface and are controlled by human interaction. These entities have control over activities such as system configuration, higher level commands, maintenance and error correction.

This category includes two subcategories. In these we have the applications that the system requires to operate such as the Configuration Tool and Production Manager, and the applications that are optional at run time. These optional are maintenance and error correcting entities such as the implemented Communication Log. These entities are needed to keep the system running on the long run but not needed every time.

- Required entities

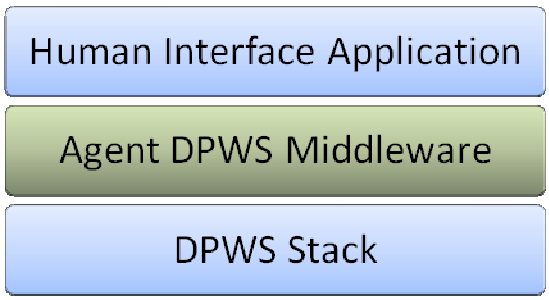


**Figure 4-24 – Required Human Interface Entity Architecture**

Since these applications need all the communication features of agents, they too have the Agent DPWS Middleware. The difference being that the control is made by humans instead of software agents.

In the implemented demonstrator there are two such entities: the Configuration Tool and the Production Manager.

- Maintenance and Error Correction entities

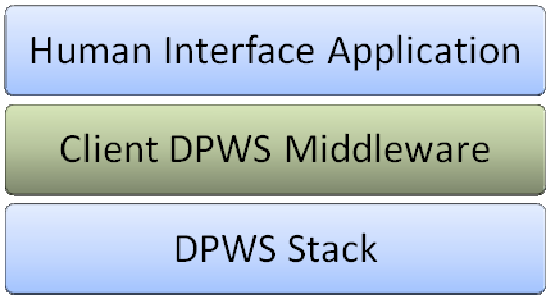


**Figure 4-25 – Optional Human Interface Entity Architecture**

Since these applications just need to use system services and not to provide services of their own they just require the Client DPWS Middleware. With this interface they will join the system without introduction, look for other entities and act upon them. That possibility enables these types of entities to join, fix and leave the system without acknowledgement. These entities can join the system at any time and in large numbers when needed, and leave without notification.

In the implemented demonstrator there is one such entity: the Communication Log.

Every application in the system has a service to enable the Communication Log functionality. This service is a standard service in this demonstrator. To have more maintenance and error correcting tools in the system, it would have to implement standard service interfaces to enable their action.

### 4.4.1 CONFIGURATION TOOL

The Configuration Tool is responsible for the dynamic configuration of all configurable entities in the system. In the implemented demonstrator these include all agents. The Configuration tool is the repository of all configuration of the system.

Before any agent can act upon the system, it must first get its configuration on the configuration tool. Afterwards it can finally start acting upon the system.

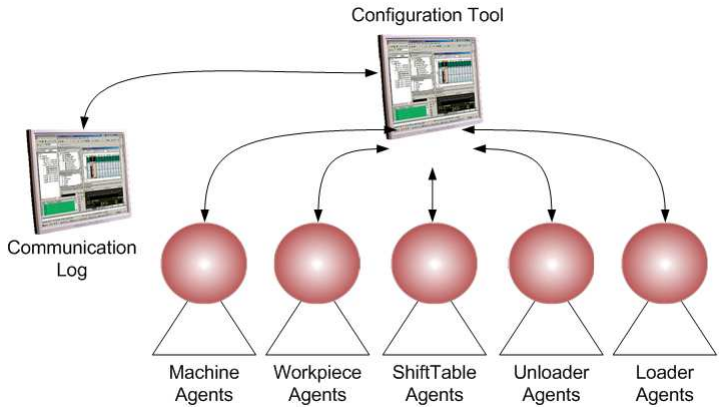


**Figure 4-26 - Configuration Tool Interactions**

Configurations can be changed at system run time and announced to the respective agents. When a new configuration is available for an agent the configuration tool announces it

so that the agent can update itself. Every configurable entity in the system is always subscribed to this entity to receive these announcements.

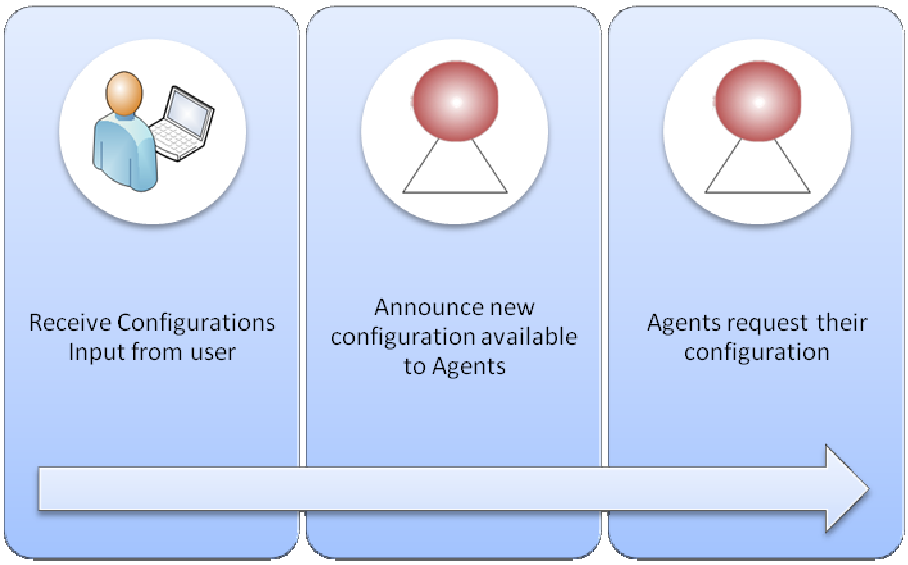


**Figure 4-27 – Configuration Tool Tasks**

When an agent enters the system and finds the Configuration Tool it checks if it has a valid configuration for it. If so, it requests it. The new Configuration Announcement sent by the Configuration Tool includes the new configuration.

Server entities cannot request configurations since they have not got a client interface. In this particular case the Configuration Tool has to detect the new Server Entity and provide its configuration through a service. The Server entities have to implement this standard service.

### 4.4.2 PRODUCTION MANAGER

The production manager is the highest level entity in the system. All workpiece process graphs are made through this application and stored in the database. Production orders are created at this entity and sent to the Loader Agent that will introduce the requested workpieces to the system.

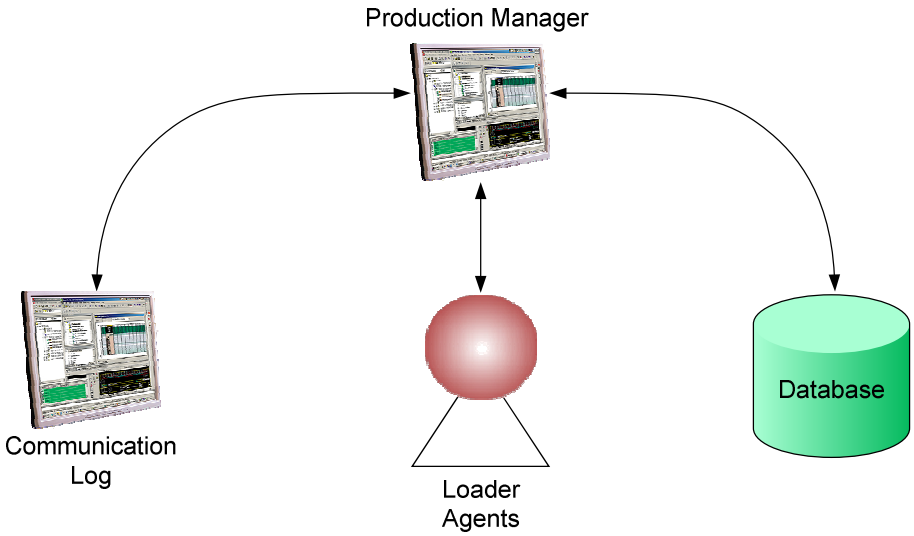


**Figure 4-28 - Production Manager Interactions**

The workpiece plan is designed for each workpiece type and recorded in the database where all workpiece agents will be able to request it.

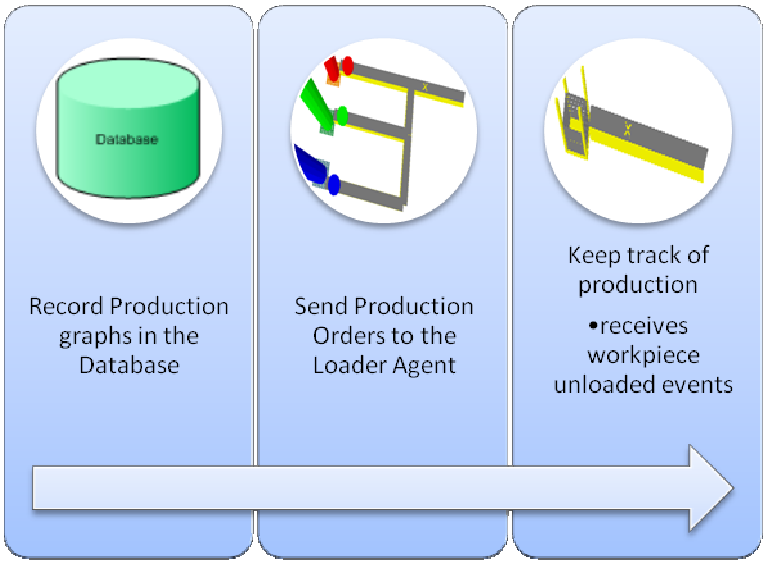


**Figure 4-29 – Production Manager Tasks**

The workpiece production orders consist of the workpiece type to produce, the number of units to produce and the base priority of every workpiece in the order. This order is then sent to the Loader Agent which controls the entry of workpieces into the system.

The Production Manager can also keep track of production by receiving events when each workpiece has been produced, sent by the unloader agent.

### 4.4.3 COMMUNICATION LOG

The Communication Log was designed to debug the system. Every multi agent system needs debugging tools and its distributed nature makes this task very difficult. The communication log enables the user to choose which entities to listen for and subscribe to log messages.

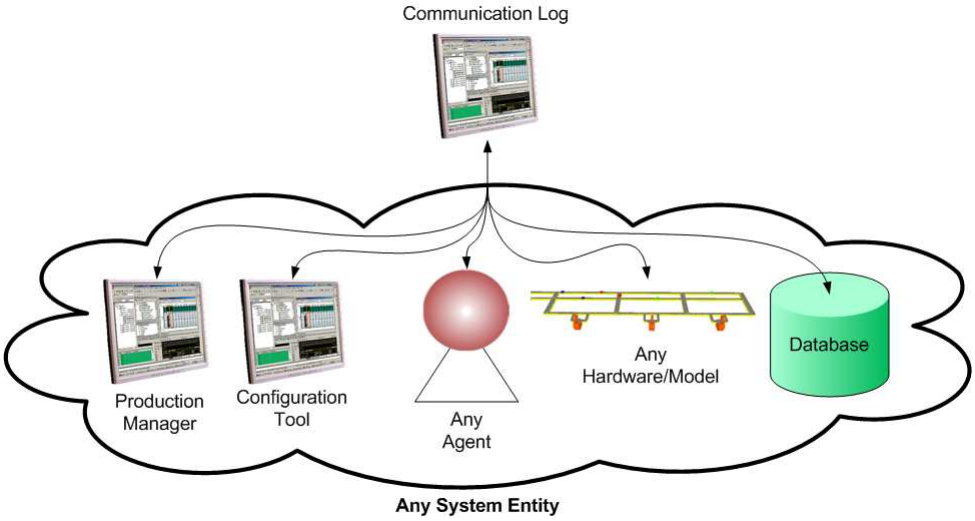


**Figure 4-30 - Communication Log Interactions**

To have this kind of functionality in the system all entities in it have to expose the same service that sends a log message for every other message it sends to any other entity.

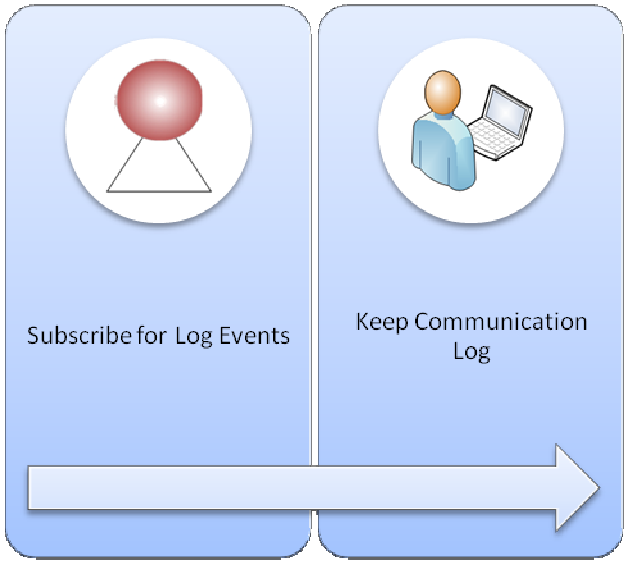


**Figure 4-31 – Communication Log**

The Communication Log is an example of a Monitorization Tool. These types of entities listen to the system and sometimes act upon it but never present themselves to its entities. All Monitorization Tools are invisible to the system.

## 4.5 DATABASE

The Database plays a passive part in the system as it does not start communication with any other entity. It announces its arrival and its services to every entity but does not have discovery capabilities.

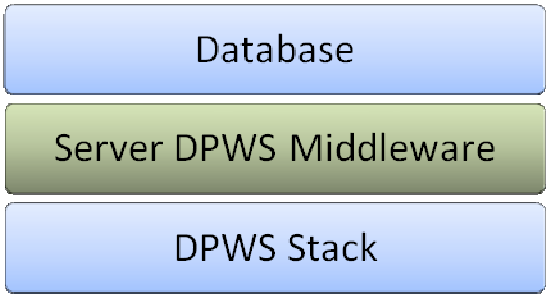


**Figure 4-32 – Database Architecture**

The database has the same DPWS interface as a 3D model because of its passive role in the system.

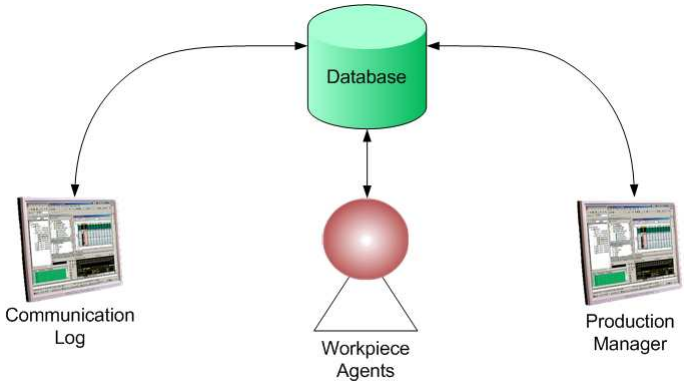


**Figure 4-33 - Database Interactions**

The database DPWS interface was made so that every communication link in the system would be DPWS based.

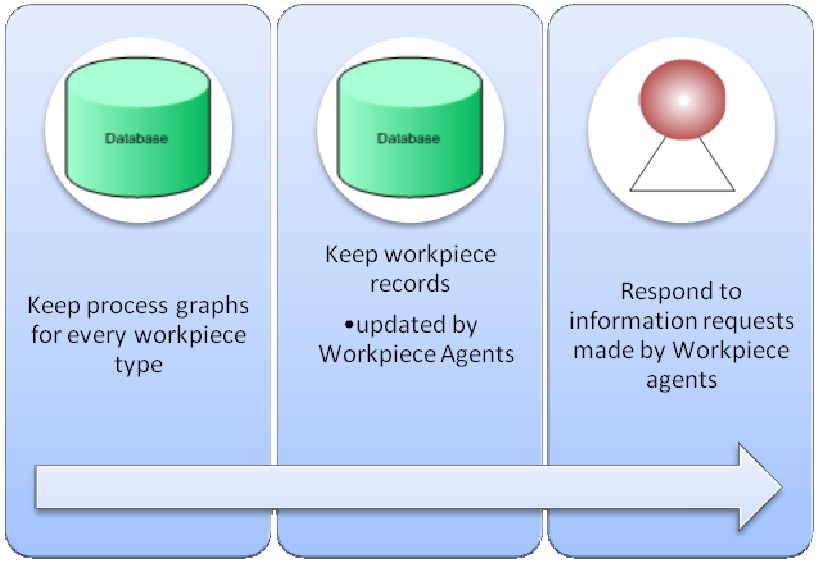


**Figure 4-34 – Database Tasks**

It is responsible for every workpiece record in the system, produced and still producing. It accepts workpiece record updates from Workpiece Agents. It has an important role in system recovery by having updates workpiece records. Business level information, such as process graphs for joining workpieces, is stored in the database.

## 4.6 TOPOLOGY

In the implemented demonstrator, the routing process was made step by step. The workpiece agent checks the best way to proceed after each interaction. After the workpiece has been shifted the Workpiece Agent verifies which is the best action to take next.

To make this decision the Workpiece Agent needs topological information about the system. In the demonstrator no entity knows the entire topology of the system, but through the names of each entity it is possible to infer its relative position.

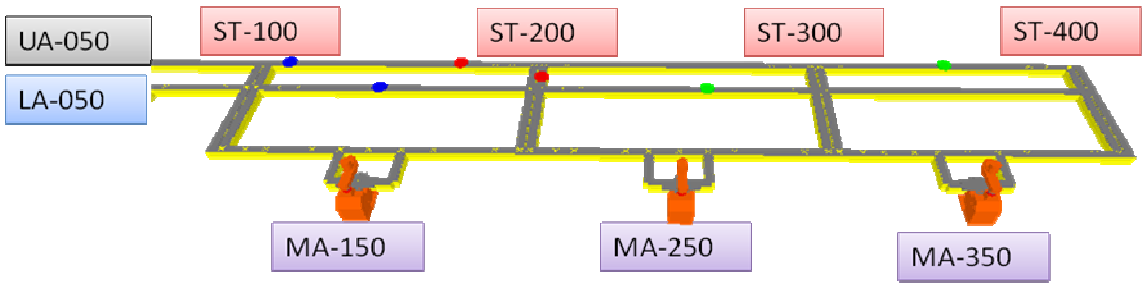


**Figure 4-35 - Demonstrator Entities Friendly Names**

The topological information, as well as type, of each entity should be in the same entity metadata, but since the current DPWS Stack does not have that possibility an alternative solution was made. With an entity friendly name it is possible to extract its type and relative position.

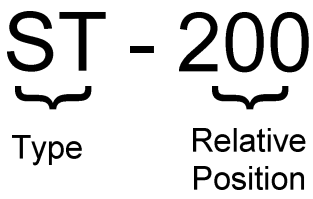


**Figure 4-36 - Friendly Name Decomposition**

From the first part of the name the type can be extracted. The second part of the name is a number that indicates the relative position of the entity. From the number we can infer that lower numbers are before this entity and higher are after, considering that the upper conveyor goes to the lower numbers. So, if the goal is MA-150 and the workpiece is in ST-200 it has to go back (go to the upper conveyor).

The shift tables divide the system topology by the 100 power. Every entity between the shift table 100 and shift table 200 has to have a number between these two.

Whenever a workpiece leaves an entity this entity sends the next entity name to the responsible Workpiece Agent. With this information the Workpiece Agent can route the workpiece toward its goal.

## 4.7 DEMONSTRATOR COMMUNICATION

All communication, between entities, in the Demonstrator is made via DPWS. Each entity in the system has its DPWS Interface through which it receives and sends requests to other entities.

The only exception is the communication between the 3D model and the DPWS Interface that exposes it, which is done by sockets.

Following the 3.3.2 entity classification we can divide the Demonstrator entities in:

- Agent Entities:

  - Loader Agents
  - Machine Agents
  - ShiftTable Agents
  - Workpiece Agents
  - Unloader Agents
  - Configuration Tools
  - Production Managers
- Server Entities:
  - Loader and Warehouse Model
  - ShiftTable Model
  - Machine Model
  - Unloader Model
  - Database
- Client Entities:
  - Communication Log

Agent Entities announce their arrival and departure, expose their services to the system, search for other entities, request and subscribe to other entities services. These entities have an Agent DPWS Middleware.

Server Entities announce when they join and leave, expose their services to the system and accept requests and subscriptions. These entities have a Server DPWS Middleware.

Client Entities look for other entities, request and subscribe to them. These entities have a Client DPWS Middleware.

In this demonstrator every entity has Middleware but an entity that just exposed its services through DPWS without the Middleware could enter the system as well.

## 4.8 HOW IT WORKS

In the following graph it is shown the overview of the processing of one workpiece.

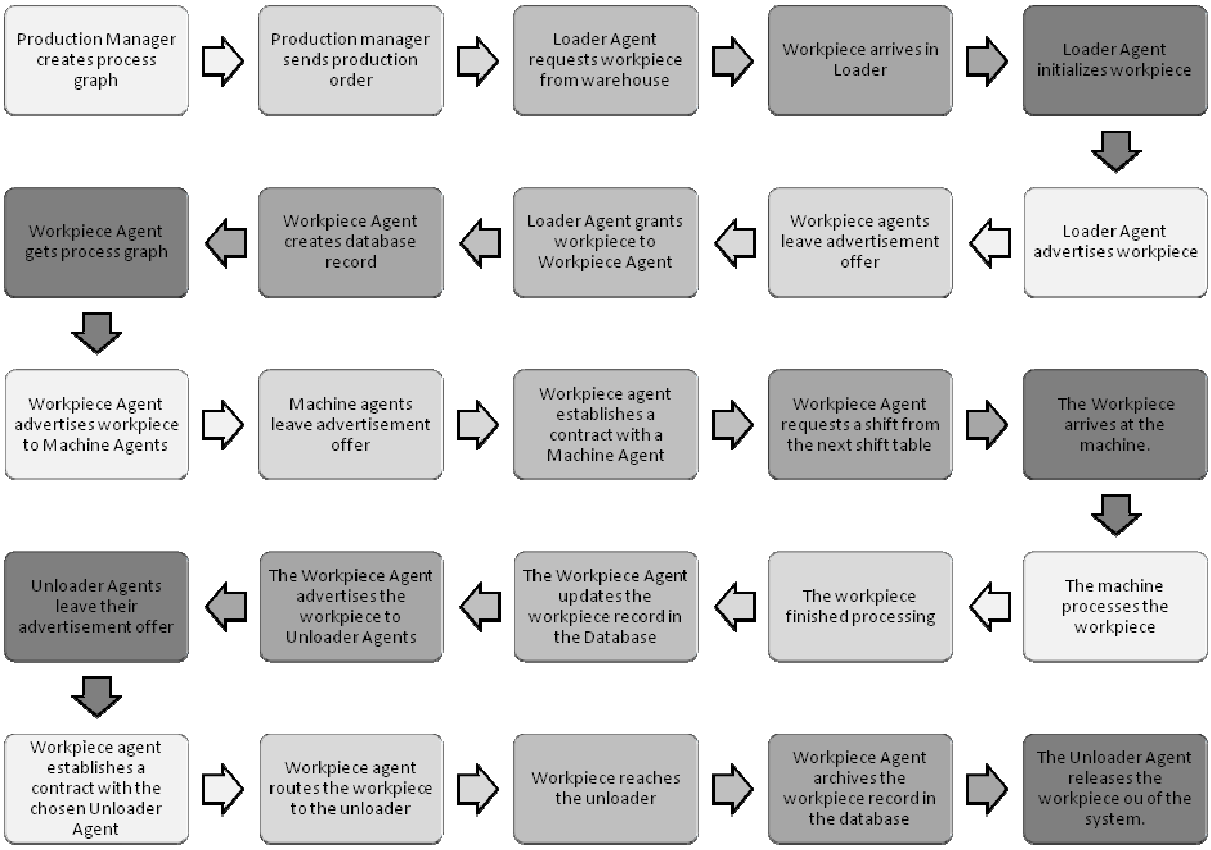


**Figure 4-37 - Single Workpiece Production Overview**

The workpiece production graph is made by the Production Manager entity. The production graph indicates the process names and sequence to produce a workpiece. After making the workpiece process graph and saving it in the database the Production Manager sends a Production Order to the Loader Agent.

The Loader Agent requests the order workpiece type from the warehouse and waits until it arrives at the loader. Upon its arrival the Loader Agent initializes the workpiece and advertises it to Workpiece Agents. Interested Workpiece Agents leave their offer at the Loader Agent. After a predetermined amount of time the Loader agent checks the received offers and grants the workpiece to the Workpiece agent responsible for less workpieces.

The Workpiece Agent creates a workpiece record at the database and requests the process graph for the workpiece type. With the process graph it then advertises the workpiece to Machine Agents. Machine Agents leave their offer and after a predetermined time the Workpiece Agent analyses their offers. A Machine Agent is chosen and a contract is established for the workpiece.

The Workpiece Agent then routes the workpiece one shift table at a time until the workpiece reaches the machine. The routing is decided step by step at each node.

When the workpiece reaches the machine the Machine Agent makes all the necessary routing in its conveyor system to guide the workpiece into the processing position. It then processes the workpiece. After processing it announces it to the Workpiece Agent and routes the workpiece out of its conveyor system.

The Workpiece Agent updates the workpiece record in the database and, if the process graph is complete, advertises the workpiece to Unloader Agents. The Unloader Agents leave their offer and wait until the Workpiece Agent makes its decision. After it chooses the Unloader Agent, the Workpiece Agent establishes a contract with it and routes the workpiece to its goal.

When the workpiece arrives at the unloader the Workpiece agent archives the workpiece record in the database a commands the release of the workpiece out of the system.

## 4.9 3D Model Simulation to Real Machines

With the proposed architecture where all interaction in the system is made through a DPWS Interface, agents could either be controlling a 3D model or a real machine with exactly the same set of services providing that the 3D model has the same interface as the real machine.

Considering that the 3D model has the same interface as real machines a system tests with the simulator will work, without any changes in the real system. [52]

## 4.10 WSDL Generator Tool

The Java DPWS Stack implemented by Schneider Electric uses the WSDL Service Descriptor File to generate the necessary code to implement the service. With the large number of entities to implement in the demonstrator, and various changes during implementation, it would not be viable to always be editing the WSDL files. So, the WSDL

Generator Tool was developed to ease this task. With the generated WSDL file the DPWS Stack generator generates the required code to implement the DPWS Interface.

The WSDL Generator Tool helps the DPWS Designer to build the WSDL file that describes the entity services. Easily create and change the service, port types, operations, messages, message information, data types, elements and all associated documentation.

# 5 Conclusion & Future Work

## 5.1 CONCLUSION

The Middleware layer, for all three entity types, was designed, implemented and tested successfully. Agents were able to communicate with each other and with other entities via DPWS as expected. The DPWS Middleware layer provides a transparent means of communication for agents in a system without a centralized communication entity like JADEs' Directory Facilitator.

3D Simulation was also studied and a model was implemented in order to test the approach in a virtual production line. The connection between the system entities and the Simulator was successfully established. The demonstrator was successfully implemented with a significant number of different entities communicating via DPWS with the DPWS Middleware.

In the first attempt to implement the DPWS middleware, the communication layer, including the DPWS stack and middleware was completely independent from the entity. It communicated with it via events and the connection was made by a launcher. This approach was successful and provided a modular implementation approach. While successful this approach had a downside in Request Response type services. In these the implementation had an unnecessary level of complexity and thus was abandoned to a more integrated approach. The DPWS stack and the middleware were then grouped into an object ready to be used by the entities providing a much easier way to implement. This implementation proved as successful as the latter.

The middleware was designed for agent communication. The middleware architecture explores all DPWS features, adds a few others and exposes them to the controlling entity in a simplified functional manner. It is composed of a server and client parts and an extra layer that groups them. Since agents are the most complex entities in the production line they make use of all of its features. Having the DPWS Middleware for agent communication complete, a different middleware had to be designed for other entities present in the production line such as databases and maintenance entities. Only a subset of the agent middleware features are necessary for such entities and two other middleware architectures were devised: a Server and a Client Middleware. The Server Middleware is designed for passive entities that just expose

their services and consume none and the Client for active entities that enter the system without presentation, act upon it and leave, also without notification. Error handling and monitorization are some of the goals for this interface. All three interfaces were designed, implemented and tested successfully.

This approach presents a number of advantages over classic implementations. The DPWS interface enables a, much sought, truly distributed production system. Some previous implementations already provided a distributed system but there always was a required centralized entity for it to run. The DPWS Discovery system grouped with the Middleware enables truly independent entities, including agents, in the production system. With this approach every entity from the production level to the business level can enter, search for other entities, request and subscribe to services, post orders and leave seamlessly. The absence of a critical entity without which the system could not function is a major achievement brought by the DPWS Stack and enhanced by the Middleware.

The Schneider Electric agent platform used in this demonstrator presented a number of shortcomings due to its overly simplistic model. These agents were mostly reactive software without the main agent features. JADE agent platform is much more suited to agency and enriching it with this approach could provide very interesting results such as the elimination of the Directory Facilitator.

The development of dedicated hardware that can run the DPWS Stack natively will increase the response time enabling the deployment of DPWS interfaces deep down into the most basic elements of the production line. Development in this direction is being undertaken by Schneider Electric.

During the beginning of this thesis, Semantics were studied as they present a great future improvement to these communication models. To enable a possible future addition of semantics to the approach the DPWS interfaces must be designed with specific non-generic services. These non-generic services can then include semantic description and be used in real-time by, previously unaware of such services, agents. This is discussed more thoroughly in the Future Work Section.

During this project it was noted that current agents, due to the slow response caused by its decision making features are not suitable for lower level control. Until hardware that is

powerful enough to uphold its processing needs, agency should be limited to high level, non time critical, decision making.

The DPWS Stack was thoroughly studied and its flaws identified. Minor changes had to be made to the Stack in the stated use case in order to expose functionality that was hidden. The Stack created the service instances and hid them. In order to listen to their events they had to be exposed so the entities could subscribe to them.

During the implementation of the demonstrator some standards were found lacking such as standard maintenance services and a topology description model. Both are discussed below.

This work was presented to the Vice President Connectivity Architectures & Platforms and to the Manager of European Projects of Schneider Automation GmbH (Germany) and had a very positive reaction.

## 5.2 FUTURE WORK

A number of possible additions to the approach presented in this thesis were identified, such as improvements to the current DPWS Stack, standard DPWS interface port types, semantic notation to service descriptions and FIPA compliant communication model.

### 5.2.1 DPWS STACK

The DPWS Stack available still has to implement some key features to enable a simple interaction between entities. Entity Type as well as Topology Information should be included in the entity metadata. Since more information will be added over time, Custom Metadata should be implemented. With Custom Metadata all information can be added as needed. With Topology Information and Entity Type in the Custom Metadata every entity will have the necessary data when it acknowledges other entities. Friendly names will still have to be unique but no extra data will be extracted from it.

Every message has a sender and the sender friendly name or UUID (Universally Unique IDentifier) should be easily extracted from the message. Any agent always has to know who requested each service and that information should not be in the body of the

message, but in the header. The current JAVA DPWS Stack does not currently expose that feature.

### 5.2.2 STANDARDS

In order to have a system capable of Monitorization a standard port type was implemented in every entity. This port type had just an event that would be sent to subscribers whenever the entity would send a message. This port type had log functionality.

To achieve full Monitorization functionality of a system, as well as error correction, a standard Monitorization port type has to be devised and always implemented in every Entity. Each Entity will then expose every action to subscribers as well as enable other tools to act upon it in order to correct possible system errors at run-time.

### 5.2.3 SEMANTICS

Semantic Web Services could play an important role in automation by having agents acknowledge previously unknown entity services at run-time, and use them.

There are a number of different approaches to bring Semantics into Web Services. Three were presented in [63]:

- Develop a SW version service description language, then map to WSDL (OWL-S)
- Annotate WSDL; provide additional information that defines semantics of a part of the document. (WSDL-S, Data Dictionary Link)
- Transform WSDL into a semantic language (WSDL-RDF Mapping)

To these three approaches the author points down issues and makes comments giving the WSDL-S approach the most positive review.

He states that OWL-S needs tools and that it is too complex for non-experts amongst other issues.

WSDL-S is a good approach mainly because WSDL is a widely spread standard and incremental approach. It is easier to be accepted. He also stated that adapting existing tools to

support WSDL-S is easy. WSDL-S is presented by the author to be very flexible accepting any ontology representation language or a combination of multiple representation languages.

The last possible approach presented by the author, WSDL-RDF is not so thoroughly discussed as the previous two maybe because it is the newest of the three and there is not much information on it.

The approaches are displayed in a comparison table:

**Table 5-1 - Semantic Web Service Approach Comparison [63]**

| | OWL-S | SWMO/FLOWS | WSMO | WSDL-S |
|---|---|---|---|---|
| Services | General | | Web Service | |
| How semantics are given | From Semantic Service Description to (Web) Service | | | WS to Semantic |

Comparing WSDL-S with OWL-S it is noticeable that WSDL-S is specifically bringing semantics to Web Services whereas OWL-S first brings Semantics to the Web, and also Web Services.

The WSDL-S approach can be implemented easily in the current DPWS Stacks by including the WSDL request which would be processed by other entities or included in the entity metadata ready to be processed.

Whichever approach is taken the advantages to systems that implement it are numerous. With sophisticated Agents controlling the production few changes would have to be made every time a new product is to be produced by the system.

### 5.2.4 FIPA COMPLIANT COMMUNICATION

FIPA is the standard in Agent Communication. With DPWS made available for agents, the communication protocols should follow the same route.

FIPA compliant communication could prove valuable as it would make testing of new features easier since there are already some implementations using its standard.

To make the current DPWS Stack FIPA compliant, it would have to undergo a serious number of changes. The result would include a communication suitable for agents with all

necessary data in each message and possible interaction with other agent based infrastructures such as JADE [32].

### 5.2.5 MIDDLEWARE GENERATOR

The implementation of the approach presented in this thesis provided the means to create a generator that could automatically, given some configuration, generate the middleware for any entity. A generator could be made which would require the input of the service interface and few other parameters and would output the full middleware ready to be added to the entity. This generator added to a DPWS JADE would provide a full featured and easy development of multi agent systems.

# 6 Bibliography

[1]. SIRENA, "Service Infrastructure for Real-time Embedded Network Applications". [Online] 2006. http://www.sirena-itea.org.

[2]. SOCRADES, "Service-Oriented Cross-layer infRAstructure for Distributed smart Embedded deviceS". [Online] 2006. http://www.socrades.eu.

[3]. **K.Ueda.** *"A concept for bionic manufacturing systems based on DNA-type information".* s.l. : in PROLAMAT Tokyo: IFIP, 1992.

[4]. **L.Gou, P.B.Luh, and Y.Kyota.** *"Holonic Manufacturing Scheduling Architecture, Cooperation Mechanism and Implementation".* s.l. : Computers in Industry, vol. 37, pp. 213-231, 1998.

[5]. **S.Bussmann, and D.C.Mcfarlane.** *"Rationales for Holonic Manufacturing".* s.l. : Second International Workshop on Intelligent Manufacturing Systems, Leuven, Belgium, 1999, pp. 177-184.

[6]. **H.Van Brussel, J.Wyns, P.Valckenaers, L.Bongaerts, and P.Peeters.** *"Reference Architecture for Holonic Manufacturing Systems: PROSA".* s.l. : Computers in Industry, vol. 37, pp. 255-274, 1998.

[7]. **R.Babiceanu, and F.Chen.** *"Development and applications of holonic manufacturing systems: a survey".* s.l. : Journal of Intelligent Manufacturing, vol. 17, pp. 111-131, 2006.

[8]. **M.G.Mehrabi, A.G.Ulsoy, and Y.Koren.** *"Reconfigurable Manufacturing Systems and their Enabling Technologies".* s.l. : International Journal Manufaturing Technology and Management, vol. 1, pp 113-130, 2000.

[9]. **Y.Koren, U.Heisel, F.Jovane, T.Moriwaki, G.Pritchow, A.G.Ulsoy, and H.Van Brussel.** *"Reconfigurable Manufacturing Systems".* s.l. : CIRP Annals, vol. 48, 1999.

[10]. **M.Onori, H.Alsterman, and J.Barata.** *"An Architecture development approach for evolvable assembly systems"* . s.l. : International Symposium on Assembly and Task Planning: From Nano to Macro Assembly and Manufacturing, Montréal, Canada, pp. 19-24, 2005.

[11]. **M.Onori, J.Barata, and R.Frei.** *"Evolvable Assembly Systems Basic Principles".* s.l. : Conference on Information Technology for BALANCED AUTOMATION SYSTEMS in Manufacturing and Services, Ontario, Canada: Springer, 2006.

[12]. **J.Barata, P.F.Santana, and M.Onori.** *"Evolvable Assembly Systems: A Development Roadmap.* s.l. : IFAC Symposium on Information Control Problems in Manufacturing, Saint-Etienne, France, 2006.

[13]. **R.M.Frei, L.Ribeiro, J.Barata, and D.Semere.** *"Evolvable Assembly Systems: Towards User Friendly Manufacturing".* s.l. : International Symposium on Assembly and Manufacturing, Ann Arbor, USA:IEEE, 2007.

[14]. **M.Onori.** *"Evolvable Assembly Systems - A New Paradigm?"*. s.l. : 33rd International Symposium on Robotics, Stockholm, 2002.

[15]. **J.Barata, R.Frei, and M.Onori.** *"Evolvable Production Systems Context and Implications"*. s.l. : International Symposium on Industrial Informatics, Vigo:IEEE, 2007.

[16]. **J.Barata, M.Onori, R.Frei, and P.Leitão.** *"Evolvable Production Systems: Enabling Research Domains"*. s.l. : International Conference on Changeable, Agile, Reconfigurable and Virtual Production, Toronto, Canada, 2007.

[17]. **R.Frei, J.Barata, and G.Di Marzo Serugendo.** *"A complexity theory approach to evolvable production systems"*. s.l. : International Conference in informatics and control, automation and robotics, Angers, France, 2007.

[18]. **P.Leitão.** *"An Agile and Adaptive Holonic Architecture for Manufacturing Control", PhD Thesis*. University of Porto : s.n., 2004.

[19]. **J.Lastra.** *"Reference Mechatronic Architecture for Actor-Based Assembly Systems, PhD Thesis"*. s.l. : Tampere, 2004.

[20]. **J.Barata.** *"Coalition Based Approach for Shop Floor Agility, PhD thesis"*. Monte da Caparica, 2003 : s.n.

[21]. **W.Shen, S.Lang, and L.Wang.** *"iShopFloor: An Internet-Enabled Agent-Based Intelligent Shop Floor"*. s.l. : IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS - PART C: APPLICATIONS AND REVIEWS, vol.35, 2005 .

[22]. **EUPASS.** "Evolvable Ultraprecision Assembly Systems". [Online] 2006. http://www.eupass.org/.

[23]. **SODA.** "Service Oriented Device and Delivery Architecture". [Online] 2006. http://www.soda-itea.org.

[24]. **Hewitt, C.** *Control structure as patterns of passing messages.* [ed.] R.H. Brown (Eds.), Artificial intelligence: An MIT Perspective P.H. Winston. s.l. : MIT Press, 1979. pp. 435-465.

[25]. **N.R.Jennings and M.L.Wooldridge.** *Applications of intelligent agents.* s.l. : N.R.Jennings, M.J.Wooldridge (Eds.), Agent Technology: Foundations, Applications, and Markets, Springer, 1998. pp. 3-28.

[26]. **L.M.Camarinha-Matos, M.Vieira.** *"Intelligent Mobile Agents in Elderly Care"*. s.l. : Journal of Robotics and Autonomous Systems, 27(1-2), 59-75, 1999.

[27]. **L.Monostori, J.Váncza, S.R.T.Kumara.** *"Agent-Based Systems for Manufacturing"*. s.l. : Annals of the CIRP vol.55, 2006.

[28]. **P.Wooldridge.** *"An Introduction to Multiagent Systems"*. s.l. : Addison-Wesley, Reading, MA, 2000.

[29]. **R.G.Smith.** *"The Contract Net Protocol: High-Level Communication and Control in Distributed Problem Solving"*. s.l. : IEEE Trans. on Computers, C-29/12: 1104-1113, 1980.

[30]. FIPA, The Foundation for Intelligent Physical Agents . [Online] http://www.fipa.org/.

[31]. **F.Jammes, H.Smit.** *Service-Oriented Paradigms in Industrial Automation.* s.l. : IEEE Transactions on Industrial Informatics, 2005.

[32]. *JADE technical description.* [Online] http://jade.tilab.com/description-technical.htm.

[33]. ZEUS. [Online] http://labs.bt.com/projects/agents/zeus/.

[34]. **M.Huhns, et al.** *Research Directions for Service-Oriented Multiagent Systems.* s.l. : IEEE Internet Computing, 2005.

[35]. **S.Bussmann, N.R.Jennings, M.Wooldridge.** *"Multiagent Systems for Manufaturing Control: A Design Methodology".* s.l. : Springer, Berlin, 2004.

[36]. **V.Marik, D.C.Mcfarlane.** *"Industrial Adoption of Agent-based Technologies".* s.l. : Intelligent Systems, 20(1), 27-35, 2005.

[37]. **N.R.Jennings.** *"And Agent-Based Approach for Building Complex Software Systems".* s.l. : Communications of the ACM, 44/4: 35-41, 2001.

[38]. **J.Hatvany.** *"Intelligence and Cooperation in Heterarchic Manufacturing Systems".* s.l. : Robotics and Computer-Integrated Manufacturing, 2/2:101-104, 1985.

[39]. **T.Vámos.** *Co-operative Systems - An Evolutionary Perspective".* s.l. : IEEE Continuous Systems Magazine, 3/2:9-14, 1983.

[40]. **X.F.Zha.** *"A Knowledge Intensive Multi-Agent Framework for Cooperative / Collaborative Design Modeling and Decision Support of Assemblies".* s.l. : Knowledge-Based Systems, 15:493-506, 2002.

[41]. **G.H.Anthes.** *"Agents of Change".* s.l. : Computer World, January 27, 2003.

[42]. **M.P.Singh, M.N.Huhns.** *"Service Oriented Computing - Semantics, Processes, Agents".* s.l. : John Wiley, 2005.

[43]. **D.Booth, H.Hass, F.MacCabe, E.Newcomer, M.Champion, C.Ferris, D.Orchard.** *"Web Services Architecture, W3C Working Group Note 11 February 2004".* s.l. : 2004.

[44]. **D.Box, D.Ehnebuske, G.Kakivaya, A.Layman, N.Mendelsohn, H. Nielsen Frystyk, S.Thatte, D.Winer.** *"Simple Object Access Protocol (SOAP) 1.1, W3C Note 8 May 2000".* s.l. : 2000.

[45]. **T.Bray, J.Paoli, C.M.Sperberg-McQueen, E.Maler, F.Yergeau.** *"Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation 16 August 2006".* s.l. : 2006.

[46]. **M.N.Huhns, M.P.Singh.** *"Service-Oriented Computing: Key Concepts and Principles".* s.l. : Internet Computing, 9(1), 75-81, 2005.

[47]. **L. Ribeiro, J. Barata, and P. Mendes.** *"MAS and SOA: Complementary Automation Paradigms".* s.l. : Innovation in Manufacturing Networks. vol. 266/2008, A. Azevedo, Ed.: Springer Boston, 2008, pp. 259-268.

[48]. **E.Christensen, F.Curbera, G.Meredith, S.Weerawarana.** *"Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001".* s.l. : 2001.

[49]. **Jammes, F. and Smit, H.** *Service-Oriented Architectures for Devices - the SIRENA View.* s.l. : 3rd IEEE International Conference on Industrial Informatics, 2005.

[50]. **InLife.** "Integrated Ambient Intelligence and Knowledge Based Services for Optimal Life-Cycle Impact of Complex Manufacturing Assembly Lines". [Online] 2006. http://www.uninova.pt/inlife/.

[51]. **J. Barata, L. Ribeiro, and A. W. Colombo.** *"Diagnosis using Service Oriented Architectures (SOA)".* s.l. : International Conference on Industrial Informatics Vienna: IEEE, 2007.

[52]. **D.Cachapa, A.Colombo, M.Feike, A.Bepperling.** *An approach for integrating real and virtual production automation devices applying the service-oriented architecture paradigm.* s.l. : IEEE Conference on Emerging Technologies & Factory Automation, 2007. pp. 309-314.

[53]. **L.Ribeiro.** *"A Diagnostic Infrastructure for Manufacturing Systems".* s.l. : in Electrical and Computer Science Engineering. vol. MSC Lisbon: New University of Lisbon, 2007.

[54]. **L. Ribeiro, J. Barata, A. W. Colombo, and F. Jammes.** *"A Generic Communication Interface for DPWS-based Web Services".* s.l. : IEEE International Conference in Industrial Informatics INDIN Daejeon, Korea: IEEE, 2008.

[55]. **R.S.Cost, T.Finin, Y.Labrou, X.Luan, Y.Peng, I.Soboroff, J.Mayfield, A.Boughannam.** *An agent-based infrastructure for enterprise integration.* s.l. : Proc. of ASA/MA'99, Palm Springs, CA, 1999. pp. 219-233.

[56]. **A.Koestler.** *The ghost in the machine.* Arkana Books, London, UK : s.n., 1967.

[57]. **E.H.Van Leeuwen, D.H. Norrie.** *Intelligent manufacturing: holons and holarchies.* s.l. : Manufacturing Engineer 76(2), 1997. pp. 86-88.

[58]. **S.Bussmann.** *An agent-oriented architecture for holonic manufacturing control.* s.l. : Proc. of IMS1998, Lausanne, Switzerland, 1998. pp. 1-12.

[59]. **W.Shen, Y.Li, Q.Hao, S.Wang, H.Ghenniwa.** *"Implementing Collaborative Manufacturing with Intelligent Web Services".* s.l. : Proceedings of the 2005 The Fifth International Conference on Computer and Information Technology, 2005.

[60]. **A.W.Colombo, R.Shoop, R.Neubert.** *An agent-based intelligent control platform for industrial holonic manufacturing systems.* s.l. : IEEE transactions on industrial electronics, Vol. 53, No.1, 2006.

[61]. **W3C.** Web Services Description Language (WSDL). [Online] http://www.w3.org/TR/wsdl.

[62]. —. XML Schema Data Types. [Online] http://www.w3.org/TR/xmlschema-2/.

[63]. **Song, Zhexuan.** *Bring Semantics into Web Services.* s.l. : CMSC 828W Class Presentation, 2005.

[64]. Schneider Electric. [Online] http://www.schneider-electric.com.

[65]. **S.M.Deen.** *Agent-Based Manufacturing - Advances in the Holonic Approach.* Springer-Verlag, Heidelberg, Germany : s.n., 2003.

[66]. **I.M.Delamer, J.M.Lastra.** *Loosely-couple automation systems using device-level SOA.* s.l. : 5th IEEE International Conference on Industrial Informatics, Vol 2, 2007. pp. 743-748.

[67]. **J. Barata, L.Ribeiro, M.Onori.** *"Diagnosis on Evolvable Production Systems".* s.l. : International Symposium on Industrial Electronics Vigo: IEEE, 2007.

[68]. **Ribeiro, L.** *"A Diagnostic Infrastructure for Manufacturing Systems".* s.l. : Electrical and Computer Science Engineering. vol. MSC Lisbon: New University of Lisbon, 2007.