# OPC UA for Plug & Produce: Automatic Device Discovery using LDS-ME

Stefan Profanter, Kirill Dorofeev, Alois Zoitl
fortiss GmbH, An-Institut Technische Universität München, Munich, Germany
{profanter, dorofeev, zoitl}@fortiss.org

***Abstract*—Current manufacturing and production are changing more and more into a flexible and adaptable factory layout that requires rapid changeover and short reconfiguration times of machines. Additionally the setup time for new devices should be as short as possible.**

**In this paper we propose a hierarchical architecture for a multi-level Plug & Produce system and evaluate the proposed structure using open source OPC UA implementations for easy integration of new devices into an existing system. Aside from the requirements for such a system, basic concepts of the OPC UA Discovery Service Set are described and different open source OPC UA implementations for C/C++ and Java are compared.**

## I. Introduction

Flexible and adaptable manufacturing is getting more and more important not only for small- and medium-sized enterprises, but also for large enterprises, which need to cope with increasing customization demands from their customers. In an adaptable manufacturing shop floor new intelligent machines and devices need to be integrated as fast and easy as possible, and the time to setup and configure should be short [1].

The concept of the Administration Shell, as defined in Reference Architecture Model Industrie 4.0 (RAMI) for components[2], embodies such intelligent machines and devices by describing the virtual representation and technical functionality of the component. This embodiment may be coupled or decoupled from the component, i.e., it may be hosted by a higher level IT system. The encapsulation of components with the administration shell provides a standardized interface for other Industry 4.0 (I4.0) components, and also higher level control systems. Additionally such I4.0 components can be logically nested to group multiple sub-components into a bigger component (see Figure 1).

An I4.0 component should provide its functionality through simple skills to abstract implementation and hardware details. I4.0 components can be either passive (sensor), or active (actuator). A skill could be, for example, triggering an action that the component can perform and/or providing some data. Skills can be combined with each other and form more complex skills, which compound the low level skills.

To easily integrate I4.0 components into a production line the overhead for configuring new devices should be kept as small as possible. Comparable to USB devices on a PC System, if a device is plugged in, it should be automatically detected and configured. Therefore a method is required to allow other devices to discover the newly plugged device, while the new device does not need any pre-configuration. The horizontal and vertical integration of new devices can be simplified and the changeover time for new production cycles can be reduced by using a Manufacturing Service Bus (MSB) which implements automatic device discovery functionality, and a common standard for skills.

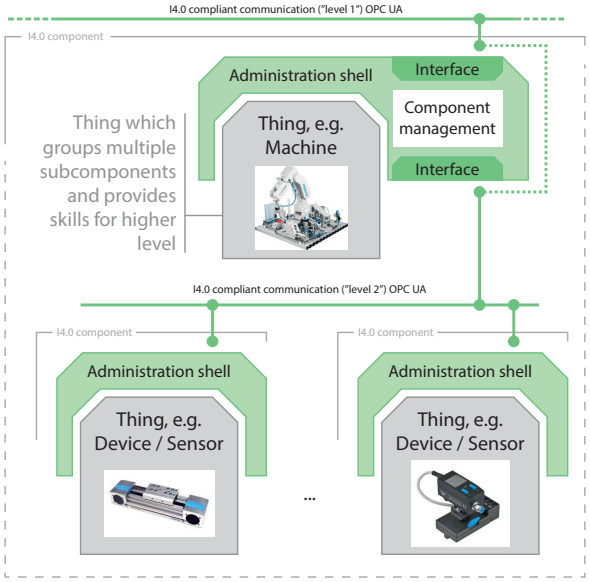


Fig. 1. The administration shell embodies machines, devices or sensors and provides a standardized interface to other Industry 4.0 components. An administration shell may also act as a logically nested group of multiple sub-components[2].

Section II gives a short overview of other Plug & Produce implementations and research. In Section III we propose a hierarchical architecture for a multi-level Plug & Produce system which is able to automatically discover newly plugged-in devices using OPC UA Local Discovery Services with Multicast Extension (LDS-ME). In the following Section IV requirements for the four operational phases — *Discovery*, *Configuration*, *Production*, and *Reconfiguration* — of an intelligent I4.0 component are analyzed. Section V describes the basic concept of the OPC UA Discovery specification, followed by a comparison of different open source C/C++ and Java OPC UA Stacks and an overview on how we implemented

978-1-5090-6505-9/17/$31.00 

automatic discovery using OPC UA in Section VI. In the last Section VII we evaluate our proposed architecture, including its advantages and disadvantages.

## II. Related Work

The need to achieve competitive advantages of manufacturing for fast changing customer requirements using agile and flexible manufacturing was already identified nearly 20 years ago [3]. Especially cost consideration, increased customer choice, and easy integration of new devices were seen as one of the most important drivers of agility and evolvable production systems.

The concept of Plug & Play in IT systems was introduced by Microsoft in Windows 95. It was the first operating system which provided automatic device discovery for USB based device classes and generic drivers for these classes.

In home and office networks, Universal Plug and Play (UPnP) allows to detect specific services on the network automatically. The Devices Profile for Web Services (DPWS) [4] can be seen as the successor of UPnP, which was especially developed for embedded devices. It defines a client/server architecture, where servers provide several services. Such a service can be invoked by the clients using Web Services technology. These services also include the Web Services-Discovery (WS-Discovery) service, which allows automatic detection of DPWS-enabled devices within the network. The service interface is described using the Web Services Description Language (WSDL), thus this description must be known to the client to call a specific function. DPWS uses XML-based SOAP messages encapsulated in HTTP and transproted via UDP or TCP. A possible solution for Ad-hoc field device integration using DPWS is shown in [5].

The comparison of DPWS with OPC UA shows that OPC UA is more suitable for limited hardware due to its flexibility in implementing small OPC UA applications with a specific purpose: It requires less memory by a factor of more than 90% [6], even if those applications do not entirely fulfill the requirements of device-level SOA [7]. For example OPC UA can be implemented on resource limited embedded devices (ARM9, 100Mhz, 64KB) using the Nano Embedded Device Server Profile [8]. Additionally, OPC UA has a well-defined meta model compared to the open approach of DPWS which offers greater extensibility.

In [9] OPC UA is used for autoconfiguration of real-time ethernet systems. The authors' approach is to use a previously defined OPC UA server where the device has to register itself. The specification for automatic discovery of OPC UA devices was released two years later in 2015 [10]. Additionally the focus is on specific real-time ethernet devices. OPC UA is currently in the process of integrating Time Sensitive Networking (TSN) and thus configuration of real time ethernet in combination with automatic device discovery can be simplified to non-real-time ethernet [11].

Since automatic discovery in OPC UA is a fairly new standard and there are not many OPC UA stacks that already support the full discovery service set, most of recent research focused on predefined register servers which implies that devices need to be pre-configured. In our research we also eliminate this additional need of pre-configuration by adding multicast discovery.

## III. Architecture

Our proposed architecture for automatic device discovery using OPC UA Local Discovery Servers with Multicast Extension (LDS-ME) consists of an intelligent Manufacturing Service Bus (MSB) which is responsible of detecting other I4.0 components on the network, and to configure the component when it is plugged in. Such a component may be an intelligent workstation which is embodied by an administration shell to provide the needed services for discovery. This workstation also contains an OPC UA Server which is responsible for discovering newly plugged-in devices on the workstation itself, and provide this information to the MSB. The general architecture is shown in Figure 2.

The MSB acts as the centralized communication mean, which makes it possible to move from the automation pyramid to a more vertically and horizontally integrated automation platform. Its responsibility is also to tell a requesting workstation which other workstations are currently available so that the two workstations can directly communicate with each other. To detect other components on the network, the MSB has to implement LDS-ME.

A workstation can be seen as a super component which encompasses other components in logical terms, to act as a unit and to abstract the underlying components for a higher level. As shown in Figure 1 and described in RAMI 4.0 [2], nestability of I4.0 components requires such components to have more than one communication interface on different abstraction levels and a component management for subcomponents. The interface on the upper level is connected to the MSB and is used for registering the workstation, and to receive corresponding configuration data from the MSB. The component management is implemented as an OPC UA Server which can receive control commands from the MSB, and if necessary forward these commands to the subcomponents, i.e., devices. To be detectable by devices which belong to the workstation, the workstation also needs to implement its own LDS-ME server that listens on the lower level interface for new device announcements.

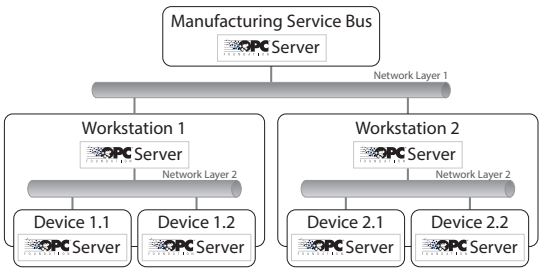


Fig. 2. Proposed architecture for hierarchical Plug&Produce. The Manufacturing Service Bus is able to discover workstations. Each workstation provides discovery functionality for devices within the workstation.

A device can be a single actor, like a motor, or a sensor which delivers specific measurement data for the higher level or other devices on the same workstation. In a more complex system structure a device could be a workstation with subcomponents. In this paper we focus on the case where a device is an actor or a sensor without subcomponents. As soon as the device is plugged in, it is registered through the network on layer 2 with its workstation and receives additional configuration from the workstation or MSB which is notified about the new registration through multicast.

## IV. Requirements

We differentiate the operation of intelligent I4.0 components (workstations, devices) by four different phases: *Discovery*, *Configuration*, *Production*, and *Reconfiguration*. To ensure that each phase can be easily controlled by the MSB, the phases have to meet specific requirements. In this section we will list different requirements for each phase, whereas we mainly focus on the discovery phase.

### A. Discovery

An intelligent Industry 4.0 component has to be able to communicate with other components or the MSB. Standard ethernet based communication is the most recommended communication method in RAMI 4.0 [2]. To be able to call services on other components, the component itself has to know the IP address and port of the other component. One way to achieve this is to pre-configure all the components (i.e., machines, devices, sensors) for a specific factory floor and its specific network setup. Using this approach, new components can not be easily integrated into the factory, as not only new components need to be configured, but also the configuration of existing components needs to be adjusted to accommodate the change, which is an error prone process.

To achieve a more Plug&Produce friendly setup without any factory-specific pre-configuration, the components need to be able to discover other components in an automatic manner. This discovery process can be further subdivided into four stages: *Plug-in*, *Register*, *Operating*, and *Plug-out*. One of the middlewares which is able to handle these stages is the OPC Unified Architecture (UA). More technical details on OPC UA can be found in Section V.

The *Plug-in* stage is the first phase, in which the component needs to set up its communication stack and get a list of all available components in the network, to be able to register itself with the correct component. In ethernet-based communication this means that a device needs to get an IP address via Dynamic Host Configuration Protocol (DHCP), and then send out a multicast message to all network devices in the current subnet, e.g., using Zero-configuration networking (zeroconf). Other devices should then respond with a corresponding message, which contains information on how the network device can be contacted.

After the list of all the network devices is available, the I4.0 component has to select the correct device to *Register* itself and by that telling other components that it is ready for production. If the component has a virtual representation located somewhere in the overall system or in the cloud, this virtual representation has to be connected to the physical instance. The virtual representation is the administration shell and can be seen as an abstraction adapter. The major issue in this stage is to understand the topological structure to register with the correct supervising register server. The task of this component managing server is to keep track of all the available devices and to connect different I4.0 components with each other. To be able to know the type of the component, each Plug & Produce device should have embedded information about its capabilities, skills, data input and output, and different key performance indicators (KPI) it provides.

After registering, the component is in the *Operating* stage, which is the normal running mode where the component should periodically check the current network status and re-register with the register server to indicate that it is still alive. This avoids polling from the register server and ensures that the list of available devices is consistent, e.g., if the network connection is broken or the device itself is not available anymore.

During the *Plug-out* stage the I4.0 component is unregistered from the MSB. Unregistering may occur in a graceful way, where the device is shut down normally, or in an abrupt way, where the network connection is broken. In the latter case the component itself has no way to notify the register server about its plugged out state, therefore the periodic re-register in the operating phase is a mean to detect the broken connection. If the device is shut down in a normal mode, it first issues an unregister service call and with this immediately notifies the server that it is shutting down.

### B. Configuration

After the communication channel for the I4.0 component is set up and it is connected to the corresponding control entities, it needs to be configured to perform its designated task. The component should be able to do a basic pre-configuration and it should provide services for self-configuration such that it can perform product-specific tasks. To bridge this gap between product requirements and machine configuration, the product configuration has to be compiled in a predefined semantic description (e.g., AutomationML[1]), and then the machine configuration can be created out of this description. This concept of configuration using AutomationML is further described in our research conducted at the same time as this paper [12].

Automatic configuration is currently one of the major challenges and a highly researched topic, e.g., in the "Plattform Industrie 4.0"[2] project. Semantic reasoning has to be performed to get from complex product and system requirements to the machine configuration. Additionally the question has to be answered, how a single configuration for a product can be split and deployed to different components or machines.

[1] https://www.automationml.org/
[2] http://www.plattform-i40.de/

Academic experiments such as MGSyn [13] or F++ [14] have applied game-based reasoning or search-based techniques to create distributed recipes or orchestration plans, based on a predefined skill library of every machine and high-level specification.

### C. Production & Reconfiguration

After the component is configured and all required information is gathered, it should provide the services to start the execution and to accomplish its task. Each such service, i.e., a basic action of the device, can be defined as a skill. Depending on the complexity of the component, multiple skills or combination of atomic skills may be presented to the upper control component.

The physical entity may also be represented by a virtual agent on a higher level. Implementing an OPC UA server on a simple temperature sensor which delivers a digital signal for the current temperature may be not affordable. This sensor can have its virtual representation added as an agent to the workstation, which handles the communication between the physical and virtual entity through a proprietary communication channel.

During execution of a skill, the component has to cope with different events, e.g., if a new job is started, while the old job is still running. Additionally, the component has to be able to handle errors which may occur on the device itself during execution of the skill, on the upper level, or if another collaborating component fails to execute its task.

If a new component is inserted and discovered in the system and there is currently a production running, all involved components need to be able to handle reconfiguration during runtime. Therefore an intelligent I4.0 component has to handle switching jobs when a new production task arrives, including the case where the hardware is altered or a collaborating device changes. Additionally the communication channel may change and thus needs to be updated accordingly. Moreover in the reconfiguration phase the I4.0 component can receive new parameters for executing its existing skills, e.g., for optimizing the execution speed or power consumption, and new product variants can be also introduced to the system.

## V. OPC UA Discovery Services

For our research we are using OPC Unified Architecture (UA), as it already solves and handles many of the requirements described in Section IV. It is a service-oriented machine-to-machine communication protocol mainly used in industrial automation and defined in the IEC 62541 specification. The main goals are to provide an open, freely available cross-platform implementation, while using an information model to describe the data. The various features and components of OPC UA are described in different specification parts released and publicly available by the OPC Foundation[3].

Part 12 and partially Part 4 of the specification describe how OPC UA applications can be discovered on a computer or network [10]. It includes the general discovery process, Local Discovery Server (LDS) and Global Discovery Server (GDS) concepts, including certificate management within GDS.

In this section we mainly focus on the discovery process and LDS functionality, and summarize the concept of application discovery in OPC UA. This summary provides a basic understanding on what OPC UA Discovery can do, and which additional functionalities are required to fulfill our use-cases. A more detailed technical explanation of the concepts can be found in the aforementioned specification.

The discovery process defines how OPC UA Clients can find OPC UA Servers on the network and then discover how to connect to the Server. Therefore the mechanisms can be separated into two parts: how can clients discover servers and how can servers make themselves discoverable.

A client can use different methodologies to find a server. The most basic concept is a list of predefined hard-coded discovery URLs of other components which the client queries. This is clearly not optimal for adaptable I4.0 components, which are used in different environments. A more flexible way is to contact a locally running server on the same host with a well known port number. OPC UA specific methods like *FindServers* or *FindServersOnNetwork* can then be called to get a list of other running servers on the network. This local server is listening for server announcements on the multicast subnet, or is searching for servers on the Global Discovery Server. The client itself can also listen for these server announcements and thus find other servers without calling the local server, which makes the system even more flexible.

OPC UA servers need to make themselves discoverable so that other clients and servers can contact them. This is achieved by implementing one of the following server types:

The *Local Discovery Server (LDS)* is an OPC UA instance running on a host. It keeps track of all the other OPC UA servers on the same host. This means that only OPC UA servers which are running on the same host as the LDS are known to the LDS. These have to explicitly register with the LDS, which is by default listening on port 4840.

The *Local Discovery Server with Multicast Extension* (LDS-ME) is an extension of the LDS, and additionally keeps track of all the servers that announce themselves on the local multicast subnet. Servers on the same host can register themselves directly, whereas other LDS-ME servers are detected by multicast announcements. This avoids the necessity to define specific IP addresses beforehand.

The *Global Discovery Server* (GDS) is another type of discovery server and can be used for discovery among multiple subnets, and in subnets where host names can not be discovered directly (for the multicast announcements hostnames have to be discoverable). A GDS may also implement certificate management services for distribution and central trust management of certificates for other OPC UA applications. Certificates are used in OPC UA for encryption and authentication to ensure clients and servers are talking to the expected entity.

[3] https://opcfoundation.org/developer-tools/specifications-unified-architecture/

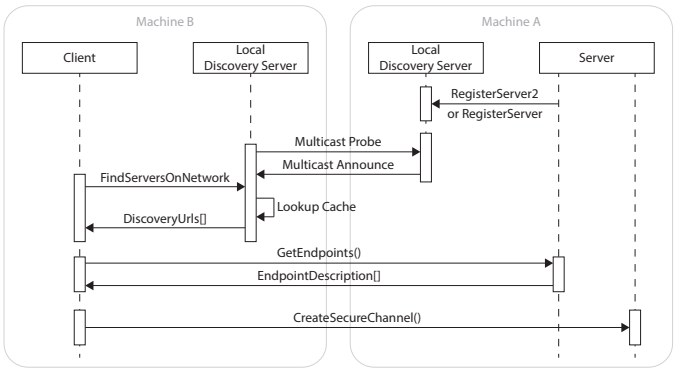


Fig. 3. Using the Multicast Subnet discovery process, the secondly attached machine B queries a predefined or discovered server for all known servers on the network and can then contact the desired endpoint on machine A.

Since our goal is to avoid pre-configuration of components, our focus lies on the case where a client uses the LDS-ME server on the same host to find other servers and other Industry 4.0 components correspondingly so that no component needs to be configured with specific IP adresses beforehand.

This is also the concept which we use for our evaluation and automatic discovery for Plug & Produce. Figure 3 shows the sequence of the discovery process: In the first step the Server on machine A registers with its LDS. When machine B is plugged in, its LDS issues a multicast probe on which the LDS on machine A responds with a multicast announcement. The LDS on machine B now knows all the available servers on the network. Finally the client on machine B queries these servers and selects the desired remote server on machine A. Using this information the connection between both machines can be established without any pre-configuration. The only necessary piece of information is the port on which the local LDS is running, which is normally port 4840, as defined in the OPC specification.

The OPC UA specification already provides basic concepts for device discovery, i.e., how the multicast messages are handled and how the registration on the LDS-ME is done. OPC UA does not include the concept of hierarchical LDS-ME servers, which is one of the concepts we describe in this paper. The next section compares different open-source OPC UA stacks and gives an overview how we implemented LDS-ME in C/C++ and Java.

## VI. Implementation

To implement Plug & Produce in OPC UA a software stack has to be selected which supports the discovery service set. More importantly it has to support the Local Discovery Server with Multicast Extenstion (LDS-ME). There exist many commercial OPC UA stacks in different programming languages but unfortunately only very few already provide an implementation for LDS-ME. There are even less open source implementations which support LDS including multicast discovery.

For evaluating the discovery functionality (see Section VII) and meeting different project-internal requirements like support for embedded devices or platform independability, we were searching for open source stacks for C/C++ and Java. There also exist other open source implementations for C#, Python, and JavaScript.

### A. Comparison of C/C++ OPC UA Stacks

In this section we compare different open source OPC UA stacks. Commercial OPC UA stacks are not part of this comparison since we only want to rely on open source software. All of the following OPC UA stacks can be compiled under Linux and Windows.

**open62541** (MPL-2.0, similar to LGPL, but includes static linking) [15]. It provides an API for server and clients, and supports nearly all features of the different discovery sets, except (initially) the Discovery Service Set. The interface is compliant with the OPC Foundation Compliance Test Tool (CTT). The information model can be automatically generated out of XML files. This project is actively developed and new features are constantly being added. Aside of Linux and Windows many different platforms are supported, e.g., OS X, QNX, Android and embedded systems.
https://github.com/open62541/open62541

**OpenOpcUA** (CECILL-C, similar to GPL with no fork option). It provides an API for server and client development, and allows to dynamically load UA information models from XML files. It is also tested with the CTT. Its main disadvantage is that a one time fee is requested to access the codebase.
http://www.openopcua.org

**ASNeG OpcUaStack** (Apache License, 2.0). It provides an API for server and clients with only basic functionality, i.e., reading, writing, and monitoring OPC UA Variables.
http://asneg.de

**FreeOpcUa** (LGPL). It is a server and client library with support for most of the basic OPC UA service sets, except the discovery service set. Since 2015 contributions to this project rapidly decreased and only a few features have been added.
https://github.com/FreeOpcUa/freeopcua

**UAF Unified Architecture Framework** (LGPL). It only implements the OPC UA client side and does not support OPC UA servers. Additionally it is based on the commercial C++ OPC UA Software Developers Kit from Unified Automation, which is required to develop applications.
https://github.com/uaf/uaf

**OPC Foundation AnsiC** provides an official reference implementation under a dual-license, proprietary for OPC Foundation Members and GPL for everybody else. This stack is the most complete one. In March 2017 the LDS-ME server implementation was released as Beta for Windows only.
https://opcfoundation.github.io/UA-AnsiC/

When starting our project in 2016 none of the above stacks fully supported the Discovery Service Set, especially the Local Discovery Server with Multicast Extension. The OPC Stack

from the OPC Foundation has now (May 2017) a beta release of a stand-alone LDS Server with Multicast Extension, but due to its GPL license and the requirement to disclose the application code built upon the LDS-ME code it did not meet our requirements. Therefore we chose open62541 for our project. It has an open license and many of the required features are already implemented. One of the most important features, the LDS-ME Server, was not yet included in the stack as it is the case for all other open source stacks. Additionally signing and encryption is currently not yet supported by open62541 but is currently under development and will be supported soon.

### B. Comparison of Java OPC UA Stacks

Besides the most commonly used commercial stacks from Unified Automation, Prosys, and AscoLab there are only a few open source implementations of OPC UA in pure Java. It is possible to encapsulate a C stack from previous subsection using Java Native Interface (JNI), which would result in a less portable but more performant data serialization. Here we compare the only two well-known open source stacks, which use pure Java and implement a majority of the OPC UA features.

**OPC Foundation Java** is the official implementation by the OPC Foundation and released under a dual-license, proprietary for OPC Foundation Members and GPL for everybody else. It provides the basic tools to implement OPC UA servers and clients, but currently only includes example code for the Nano profile which does not support the discovery service set.
https://opcfoundation.github.io/UA-Java/

**Eclipse Milo** is a project under the Eclipse Foundation and therefore licensed under EPL-1.0. It includes a fully functional stack, client, and server SDK, however it is missing certain functionality, especially LDS-ME. It already supports signing and encrypting messages, but is missing the possibility to load information models from XML files.
https://github.com/eclipse/milo

Due to the major development stage and its more open-source friendly license we decided to use the Eclipse Milo project for our implementation and testing of Plug & Produce for operating system independent Industry 4.0 components.

### C. Implementing LDS-ME

Since neither open62541, nor Eclipse Milo did support the full Local Discovery Server with Multicast Extenstion (LDS-ME), we first had to implement this feature set according to the OPC UA Specification Part 12 [10], to be able to create configuration-less Plug & Produce Industry 4.0 components.

In the open62541 C/C++ stack, the services for LDS without multicast were already implemented. This includes *RegisterServer* to allow other server instances to register themselves with the LDS, *FindServers* to allow clients to get a list of registered servers, and *GetEndpoints* which returns the list of available connection endpoints of the LDS. The fist step was to evaluate different open source multicast DNS (mDNS) implementations, which can be used within the open62541 project in consensus with the MPL-2.0 license[4]. For easy integration and support on embedded devices the mDNS library has to be self contained without any external dependencies. The only fully functional library which meets this requirement and is less restrictive than the MPL-2.0 license is the *mdnsd* library, which we adapted to be compilable under Linux, Windows and OS X and fixed various bugs[5]. Based on this library we implemented the multicast mechanism for automatic detection of other running instances within the same subnet as shown in Figure 3. Finally the *RegisterServer2* service was implemented and added to open62541, which allows other instances to register with additional multicast information, and *FindServersOnNetwork* which returns not only explicitly registered OPC UA servers, but also the ones detected through multicast messages. These changes were compiled as a pull request with more than 3000 lines of code modifications and contributed to the base repository of open62451[6], where it is now integrated into the master branch.

For the Eclipse Milo project the same amount of work had to be performed: After extending the basic methods *RegisterServer*, *FindServers*, and *GetEndpoints*, the *jmDNS* library was used to add mDNS support to Java. This enables Milo to detect other multicast enabled OPC UA instances on the network and to keep a list of known OPC UA servers. Additionally, the *RegisterServer2* and *FindServersOnNetwork* were implemented, and then again compiled into a pull request with around 3000 lines of code modifications, to be submitted to the Eclipse Milo project on GitHub[7].

### D. Discovery Process

Our overall demonstration setup for automatic discovery functionalities is described in Section III, and consists of the MSB and two workstations with two devices each. An OPC UA Server can support multiple service sets at the same time, therefore it can be a standard OPC UA Server which provides methods and data to clients and at the same time offer the LDS-ME functionality. This is shown in Figure 4. Additionally, an OPC UA Server cannot call a method on another OPC UA Server directly and thus needs to use an OPC UA Client to call the corresponding configuration methods on the lower-level server.

The detailed process for automatic device discovery on plug-in is as follows:

1) When a workstation or device is plugged in, it gets assigned an IP address and the corresponding subnet using DHCP. After network initialization, the OPC UA server on the plugged in device issues a multicast probe. All LDS-ME servers within the same subnet respond with a multicast announcement, including information how the LDS-ME server can be contacted. With this information

[4] https://github.com/open62541/open62541/issues/701
[5] https://github.com/pro/mdnsd
[6] https://github.com/open62541/open62541/pull/732
[7] https://github.com/eclipse/milo/pull/89

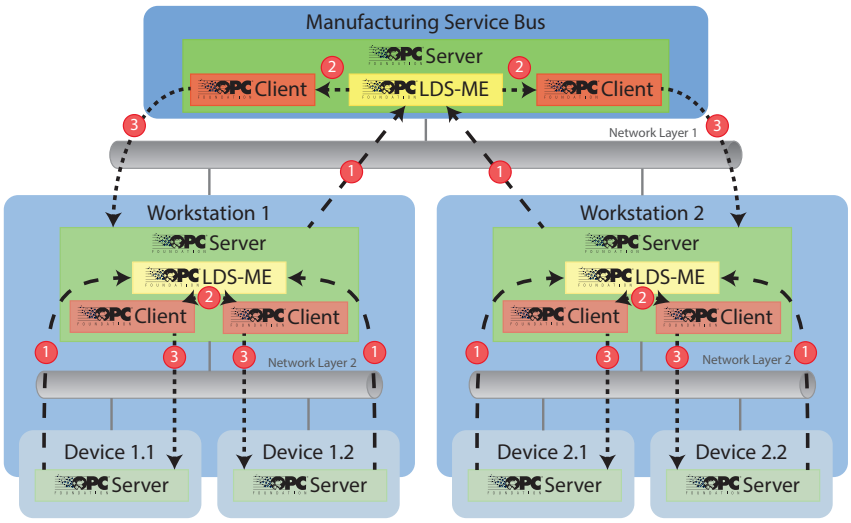


Fig. 4. Discovery process with multiple levels of hierarchy. The Manufacturing Service Bus controls Workstations which in return control Devices. ① The server detects the LDS-ME Server through multicast and registers itself with the LDS. ② The LDS-ME Server creates a new client to communicate with the OPC UA Server. ③ The Client calls the configuration method on the Server and controls its actions.

the newly plugged in OPC UA server can register itself with the LDS-ME server. This step completes the Plug-In and Register stages described in Section IV.

2) The new server is stored in a list and an OPC UA client is created to be able to talk to the newly registered server. An OPC UA server can not directly communicate to another OPC UA server.
3) The newly instanciated client then calls a predefined configuration method on the server with configuration parameters, e.g., how often status updates should be sent (Configuration pahse). This client instance is also used in the Operating phase for controlling the workstation or device respectively.
4) If a device or workstation is gracefully shut down, it has to unregister itself from the upper LDS-ME server. If it is disconnected by a network or software fault, the controlling server either detects the downtime through its controlling client or through the LDS-ME server which checks if the underlying component periodically re-registers itself.

If the network link is down or the LDS-ME server is not running when a new component is plugged in, the multicast probe should be sent after a short retry interval to make sure the registering succeeds as soon as possible.

## VII. Evaluation

OPC UA defines how data should be serialized on the wire, therefore different OPC UA stacks are able to communicate with each other. This is also the case for the discovery services: LDS-ME servers from different vendors should be able to find each other and to exchange data. We evaluated the compatibility of our implementations in open62541 and Milo in a basic way by creating dummy workstations and devices which need to register with the corresponding LDS-ME server. Different combinations of open62541 and Milo are used to show their compatibility.

Additionally, Milo is tested against the official OPC Foundation reference implementation of the LDS-ME server. The open62541 stack is currently incompatible with this server since it requires encryption for register services, which is as of now (June 2017) currently in final stage of development and not yet fully supported in open62541.

For the setup we are using a standard Desktop PC with Ubuntu Liunx where the LDS-ME server is directly started. Additionally, two Virtual Machines are set up, which simulate the two workstations from Figure 4. Each workstation contains a device simulated by directly starting an application within the virtual machine. Since the virtual machine only has a virtual network interface, the plugging in of a component is simulated by the starting of the application. The Device will then query the LDS-ME server on the workstation for other known OPC UA Servers. This query should return the OPC UA Server from the other workstation, which provides a pass-through method call so that the method on the other device can be called. Additional more detailed tests on real hardware (Festo MPS Stations and industrial robots) will be included in a later publication.

The tests have shown that both implementations are able to discover each other. In open62541 the discovery process is significantly faster, as it takes less than a second for the workstation and device to find the LDS-ME servers, register with them and then query for known OPC UA servers, to finally call the method on the other device. In Eclipse Milo the multicast probe is slower and it takes up to 7 seconds until the device or workstation detects its counterpart LDS-ME. This is due to the fact that jmDNS is first checking if there is already a DNS-SD service announcement with the same name, using a hardcoded timeout value of 6 seconds. Only afterwards the mDNS probe is sent out and the response

from the LDS-ME server is processed. In open62541 such a name conflict is handled my immediately announcing itself, and if at the same time another server with the same name exists, the mDNS implementation within open62541 detects it and can change its name to a more unique one to retry.

If different network layers are used, as shown in Figure 4, a device on one workstation can not simply connect to a device on another workstation. Either the network segments have to be connected and corresponding routing tables have to be set up, or servers on higher level have to act as proxies, e.g., a device has to call its parent workstation server, which calls the server on another workstation, which then can call a method on its underlying device. This introduces additional delays and it is desirable to keep communication of devices isolated on the same workstation. If additional data is required from other workstations or subcomponents, this data should be delivered through the MSB.

If there are multiple LDS-ME servers running within the same subnet, additional intelligence and pre-configuration has to be added to devices. If a device receives multiple LDS-ME announcement messages, it has to select the correct LDS-ME to register itself. To solve this issue, the device can be configured to only connect to LDS-ME servers with a specific ID, or the main LDS-ME server has some specific nodes within its information model, which a device can query and then decide if it is the LDS-ME it is looking for.

Due to time constraints we were not able to implement and evaluate the use of Global Discovery Servers (GDS) in our setup. GDS can be used as an enterprise wide Discovery Server which is able to collect information on other OPC UA servers even across multiple subnets.

## VIII. Conclusion

Our research shows that it is possible to implement the Plug & Produce concept on the network side using OPC UA Local Discovery Servers with Multicast Extension (LDS-ME). It allows easy integration of new devices into the network without any network-specific pre-configuration. Using multiple hierarchies of LDS-ME, it is possible to create modular workstations which abstract underlying devices to the middleware. Due to the common interface described in the OPC UA specification, different hardware from different vendors can be combined, which may lead to cost reduction.

This common interface is only possible if all the manufacturers agree on the same standard and way of implementation, thus every device or sensor has to implement a basic OPC UA stack. As compared in this paper there are many open source implementations for OPC UA, whereas open62541 for C/C++ and Eclipse Milo for Java provide a manufacturer-friendly license with most of the OPC UA features already implemented. In our future research we will evaluate different OPC UA discovery implementations in more detail on separate hardware devices. Additionally we will extend our research on the Configuration phase as described in Section IV and published in[12] which is another important factor to provide a real Plug & Produce concept for flexible manufacturing.

If it is not possible to implement an OPC UA stack on an embedded device, the corresponding Administration Shell has to be implemented by system integrators and could be deployed as a small App onto a central server or server running on a workstation, which acts as an adapter between the proprietary communication with the device and other OPC UA instances.

## Acknowledgment

The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 680735, project openMOS (Open Dynamic Manufacturing Operating System for Smart Plug-and-Produce Automation Components). We also thank Chih-Hong Cheng for assisting us in the implementation and testing of LDS-ME in Eclipse Milo.

## References

[1] M. G. Mehrabi, A. G. Ulsoy, and Y. Koren, "Reconfigurable manufacturing systems: key to future manufacturing," *Journal of Intelligent Manufacturing*, vol. 11, no. 4, pp. 403–419, 2000.

[2] ZVEI, "The Reference Architectural Model Industrie 4.0 (RAMI 4.0)," Zentralverband Elektrotechnik- und Elektronikindustrie e.V. (ZVEI), Tech. Rep. July, 2015.

[3] Y. Y. Yusuf, M. Sarhadi, and A. Gunasekaran, "Agile manufacturing: the drivers, concepts and attributes," *International Journal of Production Economics*, vol. 62, no. 1, pp. 33–43, 1999.

[4] OASIS, "Devices Profile for Web Services Version 1.1," OASIS, Tech. Rep., 2009.

[5] S. Hodek and J. Schlick, "Ad hoc field device integration using device profiles, concepts for automated configuration and web service technologies," *International Multi-Conference on Systems, Signals and Devices, SSD 2012 - Summary Proceedings*, pp. 1–6, 2012.

[6] L. Dürkop, J. Imtiaz, and H. Trsek, "Service-Oriented Architecture for the Autoconfiguration of Real-Time Ethernet Systems," *Iot-At-Work.Eu*, 2012.

[7] G. Cândido, F. Jammesy, J. B. De Oliveira, and A. W. Colomboz, "SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications," *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 598–603, 2010.

[8] J. Imtiaz and J. Jasperneite, "Scalability of OPC-UA down to the chip level enables "internet of Things"," *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 500–505, 2013.

[9] L. Dürkop, J. Imtiaz, H. Trsek, L. Wisniewski, and J. Jasperneite, "Using OPC-UA for the autoconfiguration of real-time Ethernet systems," *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 248–253, 2013.

[10] OPC Foundation, "OPC UA Specification Part 12: Discovery," OPC Foundation, Tech. Rep., 2015. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-12-discovery/

[11] S. Nsaibi and L. Leurs, "Time Sensitive Networking," *atp edition*, vol. 58, pp. 40–47, 2016.

[12] K. Dorofeev, C.-H. Cheng, P. Ferreira, M. Guedes, S. Profanter, and A. Zoitl, "Device Adapter Concept towards Enabling Plug&Produce Production Environments," in *Emerging Technologies And Factory Automation (ETFA)*, Limassol, 2017.

[13] C. H. Cheng, M. Geisinger, H. Ruess, C. Buckl, and A. Knoll, "MGSyn: Automatic synthesis for industrial automation," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7358 LNCS, pp. 658–664, 2012.

[14] N. Keddis, G. Kainz, and A. Zoitl, "Product-Driven Generation of Action Sequences for Adaptable Manufacturing Systems," in *International Federation of Automatic Control*, vol. 28, no. 3. Elsevier Ltd., 2015, pp. 1502–1508.

[15] F. Palm, S. Grüner, J. Pfrommer, M. Graube, and L. Urbas, "open62541 der offene OPC UA Stack," *5. Jahreskolloquium "Kommunikation in der Automation" (KommA 2014)*, 2014.