Microservice Architecture for Automation
Realization by the example of a model-factory's manufacturing execution system

Sebastian BRAUN M.Sc.
Aachen University of Applied Sciences
Goethestraße 1, 52064 Aachen, Germany

Dr. Chi-Tsun CHENG
RMIT University
124 La Trobe Street, Melbourne VIC 3000 Australia

Dr. Chow Yin LAI
RMIT University
124 La Trobe Street, Melbourne VIC 3000 Australia

Prof. Dr.-Ing. Jörg WOLLERT
Aachen University of Applied Sciences
Goethestraße 1, 52064 Aachen, Germany

ABSTRACT

The German government presented the concepts of Industrie 4.0 at the Hannover fair in 2012. Based on this, the German "Hightech-Strategie 2020" was initiated to establish Germany as the leading supplier for cyber-physical systems. This strategy includes implementation recommendations for future technologies as well as promotions for avant-garde innovations. In this context, Industrie 4.0 can be defined as a standardization and digitalization project across companies[1-2]. It defines problem fields on which should be worked upon in future and thus generate a long-term migration strategy towards a homogeneous, reliable and optimized system structure. This takes a continuous development process with various cross-sectional technologies as a base.

This contribution deals with the design and implementation of a framework for automation as well as the investigation of the impacts of this framework on manufacturing executions systems (MES) and in general on the concepts of Industrie 4.0. A migration strategy is shown by the example of porting an existing MES of an Industrie 4.0 model-factory with Open-Source technologies into an edge-cluster. The focus is on the paradigm change for development, deployment, and operation. Furthermore, the draft of a miroservice- and communication-model regarding application containers for automation is shown.

Keywords: Cloud Technologies, Cyber-Physical Systems, Digital Factory, Digital Plant, Manufacturing Execution Systems, Modelling & Simulation, Plant Asset Management

1. Motivation

Expert knowledge is currently concentrated at cloud-service-providers (AWS, Azure, GCE, MindSphere, etc.). Small and medium-sized enterprises (SMEs) are confronted with storing their data at external service-providers which are often connected via a nonredundant internet connection. With current Open-Source technologies, it is possible to realize an operational and maintainable on-premise cloud. The challenge is to configure the cloud's nodes to be plug-and-play capable and to establish best-practices.

The enabling technologies of digitalization offer many best-practices which can be adopted to the questions raised by automation. By combining ICT (standard) solutions with automation technologies, it is possible to create a framework fulfilling the requirements of the Industrie 4.0 initiative. Application virtualization and microservices, which enable agile and elastic software in the IT-world today, are going to play a leading role in automation in the near future.

In 2013, the VDI (Association of German Engineers) proposed a service architecture with decentralized services decomposing the automation hierarchy (Fig. 1).[3] The idea is to enable field-devices to be integrated into a service oriented architecture (SOA). However, this process significantly increases the necessary work in a heterogeneous environment and is therefore not viable.

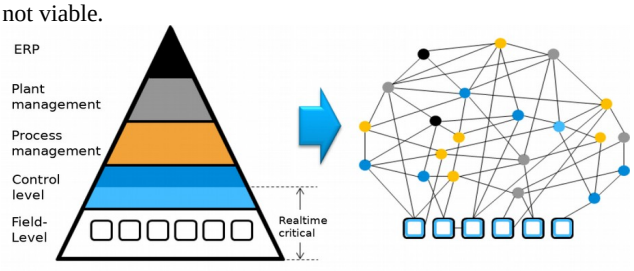

Fig 1: Decomposition of the automation hierarchy by cyber-physical systems with distributed services[3]

Most current implementations of cloud-technology in Industrie 4.0 handle the cloud itself as a simple database or use it for data processing and analysis, missing from major benefits cloud technology is providing[2,4-6]

The ICT-world communicates on a best-effort basis while the automation technologies are mostly following an deterministic and synchronous transfer of information. Those two communication paradigms are conflicting by nature. When controlling a plant from a service in the asynchronous regime, conversions are necessary. In this paper, a new framework is proposed, in which information is kept separated from the plant until it is necessary to transfer them, such that developers can fully utilize the computational power of the cloud and object-oriented programming to create applications when building a SOA. Accordingly, the PLC in the plant can focus on fulfilling the real-time requirements and act on the preprocessed information without the need to have knowledge about the workflow. Smart agents, which contain the process-knowledge, are running in their own application containers, while managing connected plants and themselves. This near decentralized approach uses a concept of multiagent systems and game theory[7] for controlling and is managed by an orchestration-layer. Moreover, the approach goes towards a resilient factory with ability-based and self-optimizing manufacturing processes.

2. Design

Infrastructure

The suggested infrastructure consists of three components: a database to store the states of all plants and processes within the factory, a cloud-runtime where one microservice per plant (=agent) is created, and an orchestration-layer in-between the database and the microservices for creating, deleting agents, managing permissions, and configuring the agents (Fig. 2). Starting multiple services per agent has no major benefit over keeping only one instance running as availability is achieved by the cloud-orchestration and if needed, states can be stored in a database. The microservices need a standard design for inter-communications and integrating themselves into a framework for orchestration.

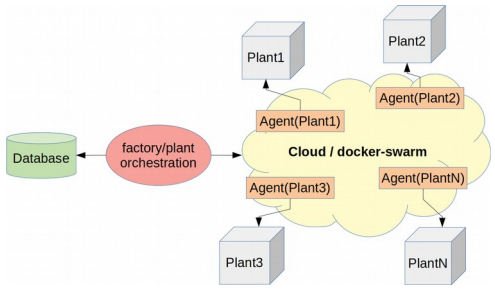

Fig 2: Proposed layout of a microservice architecture based upon docker swarm as a cloud-controller

Moreover, high availability concepts, for example a redundant network, with redundant storage and manager nodes for the provisioning of worker-nodes should be included. To provide a “plug&play” user-experience, worker nodes should boot over network via PXE (Preboot Execution Environment) and integrate themselves into the cloud seamlessly. There are existing viable container operation systems[8-11] and open-source solutions for providing such functionalities on physical clouds[12], thus their details are considered as out-of-scope.

Database For data storage, a NOSQL distributed database is preferred. The main feature that orchestration needs is the detection and notification of client failure. The load on database operations is almost constant and the amount of data stored is relatively light. Thus, a simple key-value storage is sufficient. In this work we choose etcd[13]. It covers the specified requirements and it is used in kubernetes[14-15], the cloud-runtime with the largest community at the time of writing. In etcd, keys and values are stored as binary information. A subscription feature (“watch”) is available to get event-based information about changes and deletes for keys or key-prefixes. Moreover, etcd supports authentication, encryption, and has a mechanism to use distributed locks, which can be used to ensure synchronization between database operations.

Container-Runtime The foundation of application containers was laid with the introduction of cgroups in Linux Kernel 2.6.24 (2007). With containers, it is possible to isolate resources of applications from each other. In 2013, docker[16] was released and provided a developer-friendly interface to cgroups, networking- and file management. This led to a breakthrough for application virtualization. Docker is the de-facto default cloud-runtime today. While the first SOA cloud-solutions with docker required a custom container-orchestration, Docker Inc. extended docker with this missing feature and created a cloud-runtime, called docker swarm[17], which covers the distribution of applications in the cluster and the required networking configuration. It provides load-balancing for services and has a Domain-Name-Service (DNS) running for the containers to easily access services. Two services need to be within the same virtual network to access each other, which adds a layer of security.

kubernetes and docker swarm are similar technologies using docker as a container runtime underneath and supplement it with orchestration and networking functionalities. The major difference between both technologies is the auto-scaling behavior of kubernetes, which makes it applicable for projects with dynamic workloads. However, in most scenarios, a factory is expected to deliver an almost static workload. As a result, docker swarm, which avoids this overhead, is a more suitable candidate. In this work, an additional orchestration layer using docker swarm’s API is added to the proposed framework for factories to ensure the automated starting and stopping of agents and provide configuration management.

Microservice Architecture Modern scalable and high-availability applications are created with orchestration and the concept of microservices, in which monolithic applications are broken down into small isolated and independent running services following the twelve-factor application rules[18] The services use environment variables for configuration, enforce version control and release management. They are stateless, can export services via port bindings, boot fast, have graceful shutdowns, can be used in both development and production, and conduct event and error logging. The same set of rules are for composing the corresponding agents.

There is no restriction on the selection of programming languages, as most common programming languages can be integrated in docker containers. Using high-level programming languages with object-orientation provides modularity and flexibility through polymorphism, and leads to reusable code, which can be distributed to multiple hardware-independent target agents. This provides a major benefit over ICE 61131 runtime environments, where programs are developed in a PLC-manufacturer-specific IDE.

A docker application consists of a non-writeable image, which provides an execution environment, usually a Linux operation

system, for the application. When starting a docker container, a writeable layer is created upon the image and it is where the application is executed. To ease deployment, docker images are recommended to be initialized with a small base image - like Alpine Linux[19] which has a size of a few MB at the time of writing. To ensure a small image footprint, the usage of go-lang[20] is proposed, which can be compiled into executables and run directly in the basic stock (i.e. "scratch") image.

Additionally, docker images can contain labels, which can be used to store information about the agent's service on the image itself. The labels can easily be retrieved and acted upon before the container is created, which allow creating the features of configuration management while keeping the image stateless.

Overview The proposed framework uses cloud-orchestration, provides configuration management, release-management and an interface for service-integrations. It utilizes many features docker provides to achieve this. Figure 3 displays the different layers and shows unchanged docker and docker swarm functionalities in blue, used docker functionalities in orange and this work in red.

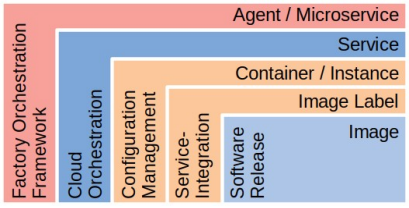

Fig 3: Correlation between docker features and -naming (right) and the components of the proposed framework (bottom)

Communication

The orchestration tool is connected directly to the etcd key-value-storage, where the state of the factory is stored. The factory-orchestration controls the edge-cloud via direct API-calls to docker swarm. The agents contain all business information, run independent containers, and connect themselves to the individual plants (compare to Fig. 2). Following sections describe the details in the communications between orchestration, agents, and plants. Each agent runs as a separate docker service using the integrated DNS. To access a service from another container, an application can resolve the service name into a (load-balanced) list of IP addresses.

Orchestration to agent There is no direct communication between orchestration and agents as to keep communications as stateless as possible. Both parties use etcd as a medium to exchange information. The database-connection supports a watch-feature, which is a default requirement for agents, making it ideal for describing communication interfaces in etcd. In general, process intelligence lies on the agents (and plants). Agents make their own decisions and control themselves. All feedbacks from the orchestration are suggestive rather than instructive – for example a shutdown command can be ignored when the machine state does not fit.

If an agent wants to shut itself down (Fig. 4), it writes on its command-key in etcd. The change in the database creates an update event, which is transmitted to all observers of this key. One observer of changes is the orchestration, which reacts on the event by stopping the container (and thus the agent) and deleting the command topic to signalize its completion.

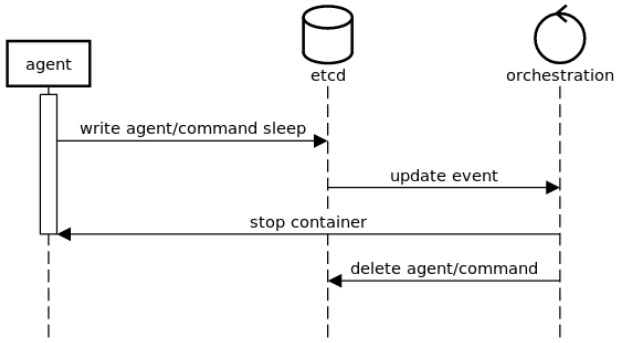

Fig 4: Sequence diagram showing how an agents communicates to the orchestration, that his service is not required to run anymore

The agent also listens to his own topic. For example in Figure 5, where the container-image information is updated in etcd, this information is transmitted to the agent, which schedules the update of its own image. The process is equivalent to a software update. To trigger the update, the agent writes on its command topic. The orchestration then submits the update service command to docker, which stops and restarts the container.

Agent to agent Agents communicate directly to each other. The interface is ideally optimized for speed and should be streaming capable with authentication and encryption. Moreover, the API's should be described through a service definition, which can be used for integrations. Even if the implementation of the communication paradigm could be left to the developer of an agent, gRPC[21] is feature complete with the requirements, widely used, and considered as a best-practice. Therefore all agents are required to run a gRPC-server on a common port, providing the framework with the possibility to specify a common remote procedure call used by the orchestration to manipulate agents.

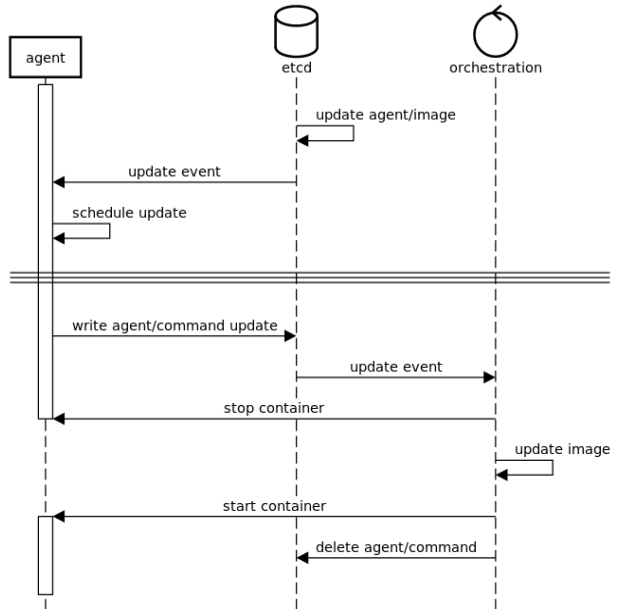

Fig 5: Sequence diagram showing an update request to update the agent's image version.

The difficult part of the agent-agent communication is that it is not certain if both involved parties are active and running, because agents are allowed to kill their processes if they are not needed anymore. A starting mechanism must be in place so that

agents are able to summon one another. To achieve this (compare to Fig. 6), the agent writes on his own "boot" key the hostname(s) that it cannot access at the moment. This key is then read by the orchestration, which starts the corresponding agent and waits for a status change. If the change-event is received, the orchestration deletes the boot topic of the requesting agent.

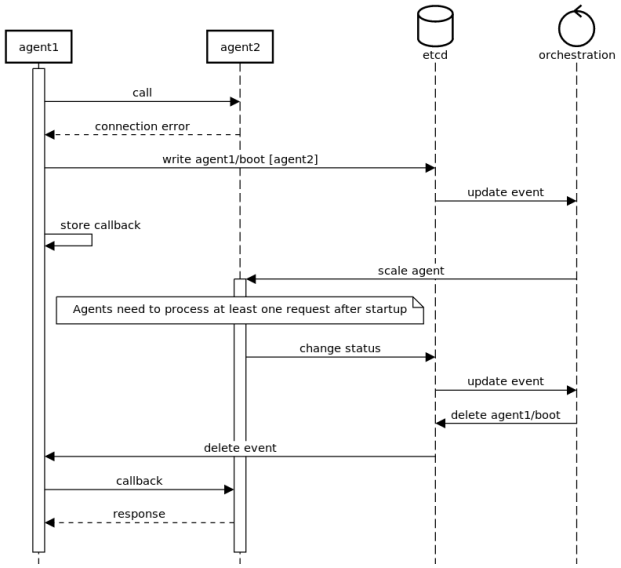

Fig 6: Sequence diagram showing the communication between an active and inactive agent. After detecting that agent2 has no process running, agent1 requests the startup of agent2 at the agent-orchestration.

Agent to plant Communications between an agent and a plant can be achieved with TCP-based technologies (OPC-UA[22], TCP-Sockets). The data-interpretation is done on the agent's side and the plant should only receive pre-processed instructions, which it needs to work on. The agents, rather than the plants, are responsible for initializing the connections and integrating themselves into other services. Data generated from the plants can be distributed by the agents. Agent to plant communication is not designed to fulfill realtime-requirements.

Security and privacy protection All communications within the cloud can be done fully encrypted with client certificates. The encryption keys must be accessible to the container runtime, so that the container configuration can be written accordingly. The key is then visible in the container configuration, which needs to be secured by user privileges. The microservice application can be shipped as a pre-compiled binary for protecting the source-code. The container images are delivered with docker's registry to the nodes, which provides user authentication and encryption. No information is required to leave the datacenter. Nevertheless, agents and services can be configured to send their data to external services. The reliability for security lies on the developers of the agents.

3. Implementation

Manufacturing Execution System

The original MES is a monolithic database application with management of stock, buffer, plants, carrier and production. It receives orders for the model factory generated by an ERP-System. According to the requirements (i.e. if the plant has the required parts available and knows how to handle the recipe), the orders are distributed by the MES to a plant. The plants connect directly or via connectors to the MES, read their orders, update their states, and execute commands directly through a REST or CoAP[23] interface.

MES in use today need to undergo a transformation in the context of Industrie 4.0, as they are typically monolithic systems which integrate only horizontally and vertically. MES are supposed to evolve into systems which provide a multi-disciplined (business) process integration with smooth interactions between partial systems, as they will generate valuable information from data.[24] The proposed framework is able to fulfill the key requirements that the ZVEI's position paper on MES[24] points out. Simply by implementing the management of orders and the state of agents, it demonstrates a new decentralized realization of MES.

The implemented MES provides a REST interface to access and manipulate data stored in a SQL-database. The same REST interface is also exposed to a web-interface for visualization. The MES creates an order-agent for every order, which consumes the the REST interface from the MES as a service and provides a gRPC interface for other agents that have no connection to the virtual network of the MES. On one hand, this generates a marketplace where the order can ask (available) agents for the time and cost of production. On the other hand, it expands the digital twin with the production workflow and the necessary compute-power to act on it.

A manager for logistical services and another for storage with a material-database can create - in combination with basic MES designs, the microservice architecture, and a multi-agent approach - a self-controlling and smart acting shop floor.

Services

The service architecture is described in Fig 7. Orders are read from an ERP system by the MES. Next, the MES writes entries into an etcd database, which is watched by the orchestration. The orchestration creates or updates agents for each order. Plant-agents write their status and abilities into etcd database in keys which are watched by the MES. Thus, the MES knows the status and abilities of the factory plants.

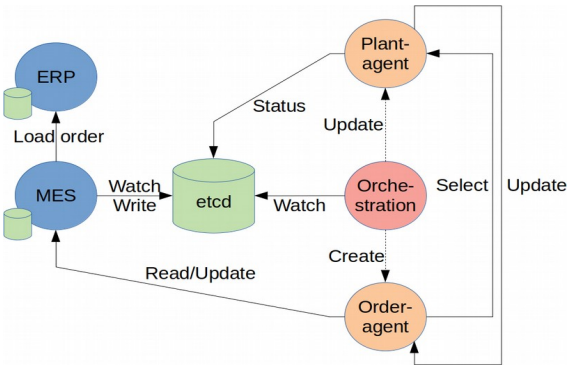

Fig 7: Integration of the MES into the automation framework

If an order-agent is created by the orchestration, the order -agent accesses the MES to get the instructions describing the build process and a list of available plants with the required abilities. The order-agent talks to plant-agents to select the best plant to execute the order. The production process is tracked by the plant-agent and transmitted over a gRPC interface back to the order-agent, which updates its database entry in the MES-

database. If an order is completed (Fig. 8), the MES updates the database entry in the ERP-system and deletes the order-key in the etcd-database. The delete-event is transmitted to the orchestration which then stops the service via the docker API.

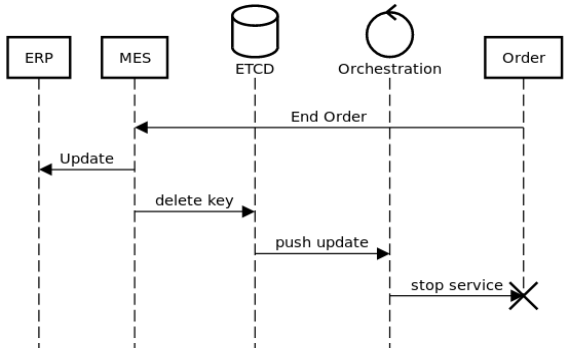

Fig 8: Sequence for completing an order

4. Migration Strategy

Agents are designed to be integrated into services. Common business logic can be abstracted in software libraries, which keeps the development effort low. Ideally the libraries should use object-oriented programming which enables parts to be modified or exchanged. Even plants with heterogenic control hardware can share those libraries and their agents can receive software updates by updating the common library.

On the hardware side, a migration strategy for legacy plants is straight forward, as agent-to-plant communications are recommended, but not required. Thus, it is possible to integrate machines via a generic agent into the cloud environment. Machines which can open a TCP socket and are programmable or have an API can be fully integrated into the MES via an agent.

5. Conclusions & future work

Cloud-based control is possible with ICT technologies and smart agents. Utilizing microservices and multiagent systems lead towards flexible, ability-based, and self-optimizing manufacturing processes. The framework presented in this work is a building block of cyber-physical systems in production and provides viable solutions for problems described by the Industrie 4.0 initiative.

One advantage of this distributed approach is the possibility to have high-availability and self-healing processes. It provides a strategy to deliver software and configuration management. Furthermore, the configuration management can be extended to field devices, which can be configured through the agents. Also, it is possible to use device independent libraries and object-oriented programming, which can greatly shorten the development process.

Processes can be abstracted, analyzed, and optimized with artificial intelligence-based approaches. With the abstraction layer provided by the agent, it is possible to collect metric data for analyzing business information, like the overall equipment effectiveness. Moreover, the framework idea presents a best-practice for automation and shows a clear separation between ICT and automation technology – business logic belongs to the agent, while process logic belongs to the plant. The associated split between the low- and high-level programming introduces the possibility to utilize continuous-integration, continuous-delivery, and agile software development in automation.

Besides, the complexity and dynamics of a distributed system introduces many sources for errors and makes debugging difficult. For API designs, a comprehensive error handling is required by the developer of libraries and agents. It is recommended to practice log-aggregation (i.e. ELK-Stack[25]) and error-tracking (i.e. Sentry[26]) in a production system. While it is relatively easy to deploy a cluster, its operations, especially at transition phases, and software development are more challenging.

[1] J. Wollert, Industrie 4.0 – warten bis die Revolution vorbei ist? Ängste und Chancen rund um Industrie 4.0 **Kommunikation und Bildverarbeitung in der Automatisierung**, Springer Verlag, 2017, pp 117-186.
[2] M. Wollschlarger, T. Sauter. J. Jasperneite, The Future of Industrial Communication IEEE **Industrial Electronics Magazine**, 2017
[3] VDI/VDE Gesellschaft Mess- und Automatisierungstechnik **Cyber-Physical Systems: Chancen und Nutzen aus Sicht der Automation**, Thesen und Handlungsfelder, 2013
[4] R. Drath, A. Horch, Industrie 4.0: Hit or Hype, **IEEE Industrial Electronics Magazine**, 2014
[5] A. Rojko, Industry 4.0 Concept: Background and Overview, **iJIM**, Vol. 11, No. 5, 2017
[6] S. Heymann, L. Stojanovic, K. Watson, S. Nam, B. Song, H. Gschossmann, S. Schriegel, J. Jasperneite, Cloud-based Plug and Work architecture of the IIC Testbed Smart Factory Web, **IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)**, 2018
[7] Y. Shoham, **Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations**, 2009
[8] CoreOS, https://coreos.com/ [Accessed: 30 Dec. 2018]
[9] Project Atomic, https://www.projectatomic.io/ [Accessed: 25. Jan. 2019
[10] RacherOS, https://rancher.com/rancher-os/ [Accessed: 25. Jan. 2019]
[11] Ubuntu Core, https://www.ubuntu.com/core [Accessed: 25. Jan. 2019]
[12] MAAS – Metal as a Service, https://maas.io/ [Accessed: 25. Jan. 2019
[13] etcd - Distributed reliable key-value store for the most critical data of a distributed system, https://coreos.com/etcd/docs/latest/ [Accessed: 25. Jan. 2019]
[14] Kubernetes – Production-Grade Container Orchestration, https://kubernetes.io/ [Accessed: 25. Jan. 2019]
[15] K. Hightower, **Kubernetes: Up and Running Dive into the Future of Infrastructure**, 2017
[16] D. Merkel, Docker: Lightweight Linux Containers for Consistent Development and Deployment, **Linux J.**, April 2014
[17] F. Soppelsa, **Native Docker Clustering with Swarm**, 2016
[18] A. Wiggins, **The Twelve-Factor App**, https://12factor.net/ [Accessed: 25. Jan. 2019]
[19] Alpine Linux, https://alpinelinux.org/ [Accessed: 25. Jan. 2019]
[20] The Go Programming Language, https://golang.org/ [Accessed: 25. Jan. 2019
[21] gRPC - A high performance, open-source universal RPC framework, https://grpc.io/ [Accessed: 25. Jan. 2019]
[22] OPC Unified Architecture, **IEC 62541**, 2015
[23] Shelby, Z., Hartke, K., and C. Bormann, The Constrained Application Protocol (CoAP), **RFC 7252**, June 2014
[24] German Electrical and Electronic Manufacturers' Association (ZVEI), **Industrie 4.0: MES – Prerequisite for Digital Operation and Production Management Tasks and Future Requirements**, Position Paper, 2017
[25] V. Sharma, **Beginning Elastic Stack**, Apress Berkley, 2016
[26] Sentry - Error Tracking Software, https://sentry.io [Accessed: 25. Jan 2019]