Microservices Approach for the Internet of Things

Björn Butzin, Frank Golatowski, Dirk Timmermann
Institute of Applied Microelectronics and Computer Engineering,
University of Rostock, Germany
bjoern.butzin@uni-rostock.de

***Abstract*—The microservice approach has created a hype in the domain of cloud and enterprise application business. Before, grown, monolithic, software has been pushed to the limits of maintainability and scalability. The microservice architecture approach utilizes the service oriented architecture together with best practices and recent developments in software virtualization to overcome those issues. One monolithic application is split up into a set of distributed services. Those are strongly decoupled to enable high maintainability and scalability. In this case an application is split up in a top down manner.**

In the internet of things, applications need to be put together from a set of small and independent services. Thus, creating value added services would require to freely combine services of different vendors to fully make use of the IoT's heterogeneity.

Even though the direction is different, many of the requirements in microservices are similar to those of the internet of things. This paper investigates patterns and best practices that are used in the microservices approach and how they can be used in the internet of things. Since the companies using microservices have made considerations on how services have to be designed to work together properly, IoT applications might adopt several of these design decisions to improve the ability to create value added applications from a multitude of services.

***Index Terms*—internet of things, microservices, patterns, best practices, distributed services, service oriented architecture**

I. Motivation

The number of connected devices is rapidly growing. Gartner predicts about 21 billion devices by 2020 [1] Cisco even predicts about 50 billion connected devices [2]. However, the question arises how these devices or "things" could interact to create an added value to the user. This starts from basic considerations like how to communicate, up to handling the complexity of hundreds of things to cooperate. It is obvious to see that there could not be one monolithic application to handle all data that is produced and consumed by the individual things anymore. This has its reason in the number of device combinations far beyond what we can handle today, but also in the decentralized approach that is intensified by the number of different business stakeholders. Thus, we need an approach to handle systems complexity in a way that allows applications to work independently of each other and only have a loose coupling if there is a necessity to communicate with other things. To achieve loose coupling as well as encapsulation, the service oriented architecture (SOA) already is a solution in the internet of things (IoT) and cyber-physical systems (CPS) that is used today. SOA has been utilized throughout the recent years but it can be seen that this is not sufficient to achieve interoperability between multiple solution providers.

978-1-5090-1314-2/16$31.00

In 2014 the term microservice was coined. Very briefly the term describes a more concrete interpretation of SOA. James Lewis and Martin Fowler describe microservices the following: "In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies." [3]. There are some similarities in the goals of microservices and the internet of things, namely:

- lightweight communication,
- independent deployable software,
- a minimum of centralized management, and
- independent development techniques and technologies.

The microservice architecture was not invented, but emerged from good practice. It has proven to be applicable for highly scalable, fast changing, distributed applications on the cloud. Some successful examples of adopters are Netflix [4], Amazon [5], and Soundcloud [6].

Microservices and the SOA approach in the IoT / CPS have the same goal: building one or multiple applications from a set of different services. However, as depicted in fig.1 these approaches come from different directions. Microservices originate from the enterprise software domain with large monolithic software. Those applications have shown to be hardly maintainable and scalable beyond a certain point. Thus, the idea is to split up the monolith into smaller, modular pieces. In contrast to that, in the IoT the small services are already given as they align with the capabilities of the embedded devices they represent. Hence, the challenge here is to build up an value added application from these heterogeneous services. This in turn requires the individual services to be designed in a way to enable high degree of interoperability. Thus, the internet of things is somewhat bottom up while the microservices approach breaks up one application top down and does not deal with a multitude of vendors. However, in the microservices approach companies have made considerations how these individual distributed services need to be designed to work together properly. Hence, if internet of things services also align to this design, their interoperability could also benefit and enable easier creation of value added applications.

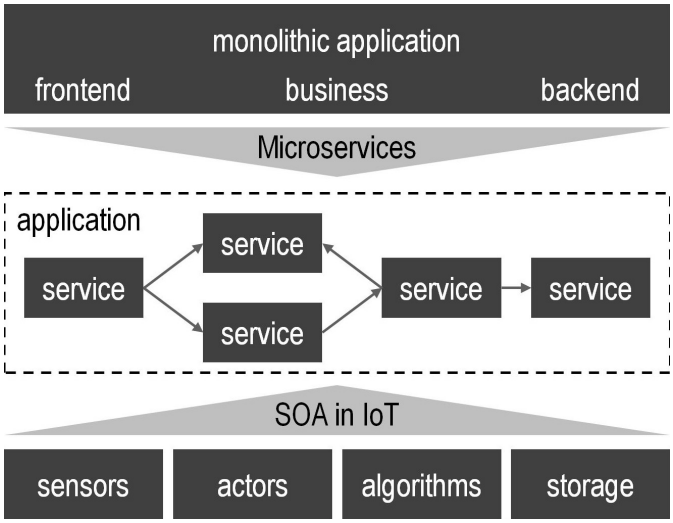

Figure 1. SOA in the IoT and Microservices Overview

In the following we will investigate the state of the art in microservices and internet of things application using service oriented approaches. Afterwards we describe individual aspects of the microservices approach and how they can be incorporated into the internet of things.

II. State of the Art

A. Service Oriented Architecture in the Internet of Things

In the internet of things the service oriented architecture is a natural choice due to the distributed, heterogeneous character of this domain. While typically used in the enterprise domain, full blown web services or RESTful HTTP solutions are often to heavyweight for constrained devices and networks. Hence, the ideas of web services and REST were used to create standards that take into account the constraints of that domain. The devices profile for web services (DPWS) [7] as well as the constrained application protocol (CoAP) [8] are examples of remote procedure call (RPC) and REST derivatives for constrained devices. In the context of CPS and industrial internet of things (IIoT) also OPC UA is used [9]. Even though, the SOA is a well-fitting approach for such applications, SOA leave much things open on how the resulting service should look like. That is where ecosystems and frameworks start to spread to substantiate the use of those technologies.

Frameworks limit the choice of technologies to ease the development of IoT or CPS applications. One could guess that frameworks increase the interoperability on a technical level. However, the behavior of the services is still up to the developer and furthermore, as there are numerous frameworks out there, the gain in interoperability can be neglected. Hence, the use of frameworks can just ease the development of individual services.

Ecosystems make much more specific assumptions about the services and define interfaces to be implemented, thus, enabling much higher interoperability. Nevertheless, they have the drawback that the use case is much more limited and interoperability is just given within that ecosystem. Some examples are Thread[10] driven by Google or HomeKit[11] driven by Apple, both for internet of things applications in the smart home domain.

B. Microservice Architecture

As already mentioned, the microservice term was coined just recently, but the ideas behind were not invented but arise from best practices. Some of them are already several years old. Sam Newman listed some of the basic technologies that led to the microservices architecture in his book [12] which shall be mentioned in the following.

- **domain driven design** [13] - intend to put the business domain knowledge at the core of the software development process. Domain experts are included more closely into the loop of software engineering.
- **continuous delivery** [14] - treating every code repository commit as it could be the next product release. This includes the automatic build, unit-, integration- and performance-testing, as well as the calculation of different software metrics. In advance to continuous integration, continuous delivery also covers the automated deployment of applications.
- **ports and adapters pattern** (see fig. 2) - by Alistair Cockburn, also known as hexagonal architecture [15] provides an approach to separate business logic from external technological considerations. Furthermore, this pattern wants to break up the traditional layers of presentation, business logic and data / integration. Instead applications have several ports e.g. for cloud connection, databases, front-end, integration and further. Each port has a mediation layer that handles crosscutting concerns like monitoring and has several adapters that connect to different technologies e.g. an adapter to a specific MySQL database at the database connection port.
- **machine to machine communication** - data exchange without human intervention.
- **virtualization platforms** - advances in virtualization, especially application containers.

Building on top of these, the microservice architecture has become an approach to support highly scalable, fast changing business models in the cloud. Several big players like Netflix or Amazon have already adopted this approach and have shown that this architectural pattern fits their needs.

The intentions of companies to move to microservice architecture are manifold, but the most common reasons are better scalability and maintainability. Furthermore, reduced time to deployment and the ability to choose better technologies for individual application parts play an important role. During the attempts to solve these issues, best practices have emerged from the experiences made. Those shall be investigated in the following section.

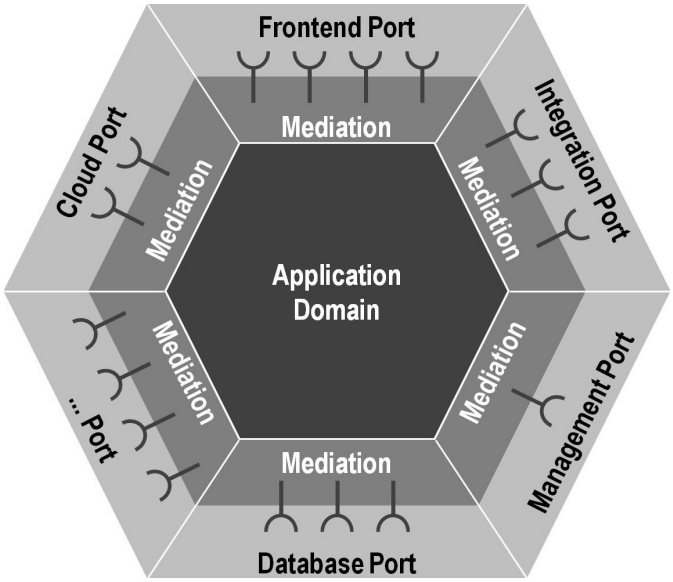

Figure 2. Hexagonal Architecture

III. Utilizing the Microservice approach for the Internet of Things

In the following we will investigate different patterns and best practices of the microservice approach and compare them with the practices in IoT. We will cover the self-containment of services, monitoring and the prevention of fault cascading, choreography and orchestration, container technologies and the handling of different service versions.

A. Self-Containment

In the microservice approach the property of self-containment is one of the core aspects. The term self-containment is used to describe, that services should contain everything they need to fulfill their task on their own. This includes not just its business logic but also its front- and back-end, as well as required libraries. Keeping it this way the services can be scaled individually by starting multiple instances. Furthermore, dependencies to other services are kept small and thus, services can evolve independently. Nevertheless, services should not become too big in order to stay maintainable. The question arises that always comes with the term microservices: "How small should it be?". The answer is not perfectly defined but originates in the domain driven design idea. In fact the answer is a trade-off between fulfilling the business case with as less dependencies as possible, and still having a concise business case to handle.

In the internet of things many SOA implementations already stick close to this principle. Especially services representing physical devices like sensors and actors are built around very brief areas of business. This is mainly driven by the constrained nature of the devices. Typically data is also kept local at the sensors / actors for a short term, as long as they can be stored. However, it is very uncommon today to have those services to offer a front-end to users. Other services in the internet of things, like optimization, aggregation, control, etc., that might run on less- or un-constrained devices have more freedom of design. Nevertheless, they should also stick to the principle of self-containment, having in mind the size of the service.

Adopting self-containment in the internet of things might create the following benefits.

- By having the back-end as part of the service, we can neglect dependencies for data storage. The data kept by a service should not directly be accessible from outside the service. This enforces the use of the service API and thus, decouples external data consumers from the internal representation of the data. Hence, this enables the independent evolution of services. In this case the internal data model can be freely changed, while maintaining interoperability.
- Having each service providing its own user interface would also enable independent evolution. This also omits the need for a centralized front-end that has to be aware of every possible device that might show up. Independent services might provide their e.g. HTML5 fragments, which can be put together into one dynamic panel.
- Providing the required libraries together with the service makes the deployment much easier, as the installation of dependencies is not required. Furthermore, if separated e.g. by containers (see sect.III-D), required libraries of different services do not interfere with each other.
- The limitation to have as less dependencies as possible leads to a better decoupling between services, an increase in autonomy and reduces the amount of required communication in the overall network. In contrast to that, limiting services to one concise business case leads to better independent evolvability, reduction in the services complexity, and a gain in freedom to compose services, but might raise the number of dependencies. So there is always a trade-off to consider.

B. Monitoring and Prevention of Fault Cascading

In order to provide monitoring capabilities, each service should provide an interface to hand over monitoring information. In particular the health status e.g. ok and broken, is of high importance. This is due to the fact that other services can check for this health status and can prevent to call broken services as far as possible. If the calling service has no other service that can be called, it can set its own status to broken too.

In order to deal with the health status but also with unexpected failures the "Circuit Breaker" pattern [16] has evolved from practice. In short the circuit breaker checks health status or remember the number of unsuccessful calls and trips if a certain threshold is reached. If the circuit breaker is triggered it will return an error instead of sending the call to the remote service to prevent the broken service to be penetrated with additional requests. After a certain amount of time the circuit breaker tries to reach the service again to test if the services

has recovered or checks for the health status and enters an half open state if the test is successful again. It is not completely open immediately to prevent the called service to become unavailable again due to high incoming traffic.

The circuit breaker pattern furthermore perfectly works together with the ”Load Balancer” pattern [17]. In this case the load balancer distributes workload on a set of equal services. The circuit breaker enables the load balancer to put work only on services that are in a good health state. Services to which the circuit breaker is only half open the number of routed requests is lowered. Broken services will, for the time being, not be used.

In the internet of things, both, the load balancer and the circuit breaker, can be used either on its own or in combination. Both patterns have proven to be a good way to handle the fault of remote services. With regard to the constrained nature of many internet of things appliances these patterns have additional benefits. The circuit breaker prevents unnecessary messages sent to broken services. This reduces the overall traffic in the constrained network and saves energy that was otherwise spent for (re-)transmission. The load balancer can increase the lifetime of wireless sensors as the workload is shared among several devices, which enables them to stay longer in low power modes. As the circuit breaker can be used by every service (as needed) with no regard of the called service, this pattern is always possible even if the called service is provided by another vendor. Thus this is a good way to provide resilience to IoT applications.

Related to the detection and prevention of faults is the logging of services. **In the microservice architecture** it is recommended to use one logging format throughout all services. This helps to aggregate individual logs to get a whole picture of the overall system. **In the internet of things** however, this is not suitable as probably no one is in control of all services hosted in an IoT scenario. Nevertheless, developers of IoT services should be emphasized to use a common logging format e.g. with internationalized time stamps to ease integration with other logs.

C. Choreography over Orchestration

When talking about putting several services together, there are multiple ways to do that. The grater concepts of doing so are orchestration and choreography. Orchestration means, that one instance, the conductor, is in control of the services to be called and does that in a centralized way.

In a choreography instead, each participant does its part on its own and the resulting application is created by the sum of the individuals. The individual parts take up their activity on an event driven basis.

From the microservice architecture point of view both is possible. From practice one should favor choreography, except there are good reasons to use orchestration. Choreography implies a higher degree of freedom in the way things can be handled. As example we can imagine a set of services that is triggered by a single service in an orchestration manner. To add an additional service, the conductor needs to be changed. This however, is only possible if the conductor is not a product of another vendor in an IoT scenario. When utilizing the choreography approach, things look different. When a certain event occurs, each service that listens to the event is triggered to do its part. Now to add something is pretty easy. The new service just needs to listen for this event and also do its part if the event occurs.

However, there is one thing to keep in mind when using the choreography style. There is no instance that can track if all required actions are done successfully in the first place. To overcome this issue there might be an additional service that only monitors (not triggers) the services that need to be executed. This way we can to monitor for successful execution, but also can add new services on demand.

Today, IoT services are often combined using an orchestration style, because it is easier to implement and protocols like HTTP do not have native support of event based communication. An exception is the Message Queue Telemetry Transport protocol (MQTT) which enforces to use event based communication. The Idea of choreography in microservices can serve as blueprint for internet of things services and applications. Basic services are modeled as independent, using event based communication. Value added services might take care, that all services for a particular application are present and listen to the corresponding events. Hence, the application can monitor, if an initial event causes all related services to execute. If a service does not respond to that event, the value added service can take remedial actions. However, new services can easily be plugged in, because the value added service only makes sure that required services are executed, but not limits which services are executed beyond.

D. Container

When talking about microservices a term that can often be found is Docker. When looking at Google trends since 2013 this term is of increasingly growing interest. However, Docker is one of several solutions for containers, also called operating-system-level-virtualization [18]. Just to mention a few there are openVZ [19], lxc / lxd [20] and rkt [21] for Linux, and ”Windows Server Containers” or ”Hyper-V Containers” for Windows [22]. The basis for all of those is the concept of namespaces. In short namespaces wrap system resources to appear to processes as they would have their own instance. Examples would be routing tables, process ids, network interfaces or mounting points [23]. Changes to that resource are only visible to the processes within the namespace. However, the processes are still running on the host system. This results in the limitation that all containers use the same kernel and thus e.g. a Windows container could not run on a Linux host or vice versa. This method of virtualization is way more lightweight compared to hypervisor virtualization and thus, allow better performance, lower start-up times (seconds instead of minutes) and less storage requirements.

For the microservice architecture this is helpful as the individual services can be hosted as a single container. Those containers enclose the microservice itself, including all re-

quired libraries and data, which also supports the requirement of self-containment. This in turn provides several advantages:

- **better testability** - Tests can be run against the whole container. Thus it is tested already in the environment in which it will run during operation. This prevents problems when putting the software into operation.
- **ease of service deployment** - Each container includes the service and all of its dependencies. Thus, they can be deployed without the need to care about different libraries to be installed. Also one does not need to deal with different versions of libraries that interfere, as each service is packed together with its libraries into a single container. Thus, the used libraries of one service are invisible to other services. Furthermore the individual solutions, e.g. Docker, provide tools to automatically deploy containers; hence, deployment can be fully automated.
- **better scalability** - As services are highly decoupled, each service can scale individually by just start / stop multiple instances, i.e. containers of one image. This is feasible since the overhead of container virtualization is much lower and start-up happens much faster.

If the internet of things can also make use of containers, depends mainly on the requirements of those technologies and the constraints in the scenario. There are two projects that try to create a minimal host for Docker containers, Boot2Docker [24] and RancherOS [25]. Both require around 23MB of ROM and at least 512MB of RAM to run. This already limits the use cases of containers in the internet of things. Small embedded device like sensors and actors with about less than 512kB of ROM available are out of scope for container technologies.

A scenario in which we can use containers is at edge computing [26] or in the fog [27]. The idea of fog and edge computing is mainly to reduce latency and other network related drawbacks by putting computational power at network edges. The devices on those levels could be routers, smart phones or small PCs which would be capable of running containers. Most gateways are even powerful enough to host several containers at once. Besides technical reasons fog or edge computing might also be used to keep private data close to the user / device.

If the devices at the network edges are not able to run containers but are still not strongly constrained, there is an alternative to operating-system-level-virtualization. Instead of using containers to host multiple services one could use use OSGi. OSGi doesn't separate the applications as much as containers, but can help to manage and deploy services. As of course most OSGi implementations require Java SE to be present this doesn't help much on constrained devices. However, there are some implementations taking care of memory usage. Concierge [28] is an OSGi implementation that requires 250kb ROM and runs on Java2ME CDC which has a footprint 2 MB RAM and 2.5 MB ROM and requires a 32-bit CPU. An even smaller implementation is nOStrum (former nOSGi) [29] which requires 233kB ROM. Additionally, nOStrum does not use Java. Instead it is an OSGi implementation for ELF binaries, thus, we can run arbitrary Linux binaries with it.

Feature	Microservices	IoT / CPS
self containment	wrap around business domain and reduce dependencies, libraries packed with application	often around device capabilities, libraries not packed with application
orchestration vs. choreography	choreography preferred	often orchestration (esp. when using HTTP, DPWS or CoAP)
container virtualization	yes, Docker, lxc, etc. for separation, scalability and ease of deployment	no, but similar, OSGi
continuous integration	yes, test overall application	partly, e.g. in single vendor scenarios
continuous delivery	yes, short release cycles	no, rare updates
protocols	HTTP	HTTP, MQTT, CoAP, DPWS

Table I
COMPARISON OF MICROSERVICE AND IOT APPROACH

Of course, in internet of things scenarios incorporating a cloud e.g. to reason about data, the requirements to run containers are easily met as well.

E. Handling Different Service Versions

In the microservices approach, with container technology in mind, deploying applications is greatly simplified.

A pattern that makes use of this fact is the *immutable server pattern* [30]. After an application was tested and put into operation this specific artifact is not altered anymore. This can be emphasized by not providing any user credentials to the container. Instead when something needs to be changed one just replaces the application-container with a new version of it. This makes sure, that every artifact in operation was tested before. Furthermore if something in the new service turns out to be faulty, the old version of the application can easily be redeployed by replacing the new version again.

Another pattern that can be used with regard to multiple versions of a service is the so called *blue green deployment* [31]. Blue green deployment deals with the problem of replacing applications by new versions in place which would cause a downtime of the service. Instead when deploying a new version of a service the old and new versions are running in parallel. At the beginning all requests are routed to the old version of the service and the new service can be started and configured. After everything works fine the routers e.g. load balancer are triggered to route the requests to the newly deployed services. This allows the introduction of new versions without downtime. Furthermore, in case of a rollback we just need to reroute traffic back to the old version.

The *canary release* [32] pattern is a slightly modified version of the blue green deployment. Instead of routing any traffic immediately to the new service, the fraction of traffic that is routed to the new services is iteratively raised. This keeps the impact of a faulty new service even smaller as not everyone is immediately affected by the new version.

Another topic is the coexistence of different versions for longer periods of time. Basically there are two possibilities [12]. Maintaining two versions in parallel which means there is e.g. a service of version 1 and one of version 2. This is highly discouraged as both code bases have to be maintained in parallel. The other way would be to have one service of the new version that is accessible by the new and the old interface. In this way the new interface just needs to internally redirect to the new interface. If the old interface is not required anymore the interface and redirection parts can just be removed. If there are even more interface versions alive, those can be chained this way. With this approach changes in the implementation can be maintained in a single code base.

In the internet of things, the pattern immutable server, blue green deployment, and canary release are not yet employed. However they could be, together with container technology, to enable updates with a minimum of risk and downtime. How the version handling is done in the IoT domain is hard to find out. Nevertheless, the considerations in the microservice architecture remain valid in the IoT as well.

IV. Conclusion

In this paper we had a brief overview on some new patterns and best practices that have emerged from the microservice approach or have made it possible. We covered the aspect of self-containment, dealing with service versions, monitoring and fault handling. The container technology was investigated as well as if orchestration or choreography should be used to put services together. As a result of this paper we can see that the architectural goals of both, microservices and the internet of things, are quite similar. The practice instead sometimes is different as shown in table I. The best practices and patterns that can be found in the microservices approach are partially already part of the SOA in the internet of things. Some, like to favor choreography, might already be known, but are in many cases not adopted in the internet of things, especially when using RPC or REST based protocols. The operating-system-level-virtualization is not yet adopted in the internet of things and might show a new possibility for the deployment and update of IoT services and applications. When operating-system-level-virtualization would be used, the already existing patterns for the roll out of new versions can be used.

Altogether, the microservice approach comes from another direction than the internet of things but both have the same architecture goal. People have made detailed considerations on how services can be composed to create applications of them. Many of those considerations can also be incorporated into the internet of things to enable the creation of applications out of distributed services even if they are provided by different vendors.

References

[1] Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015. [Online]. Available: http://www.gartner.com/newsroom/id/3165317

[2] D. Evans, "The internet of things: How the next evolution of the internet is changing everything," in *CISCO white paper*, vol. 1, 2011, pp. 1–11.

[3] M. Fowler. Microservices guide. [Online]. Available: http://martinfowler.com/microservices/

[4] S. Tilkov, "The modern cloud-based platform," *IEEE Software*, vol. 32, no. 2, pp. 112–116, Mar. 2015.

[5] Jim Gray and Werner Vogels, "A conversation with Werner Vogels," *ACM Queue*, vol. 4, no. 4, pp. 14–22, May 2006.

[6] Phil Calçado. (2015, Sep.) How we ended up with microservices. [Online]. Available: http://philcalcado.com/2015/09/08/how_we_ended_up_with_microservices.html

[7] *Devices Profile for Web Services (DPWS)*, OASIS standard, Rev. 1.1, Jul. 2009. [Online]. Available: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01

[8] *The Constrained Application Protocol (CoAP)*, Internet Engineering Task Force (IETF) proposed standard RFC 7252, Jun. 2014. [Online]. Available: https://datatracker.ietf.org/doc/rfc7252/

[9] G. Cândido, F. Jammes, J. B. d. Oliveira, and A. W. Colombo, "SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications," in *2010 8th IEEE International Conference on Industrial Informatics*, Jul., pp. 598–603.

[10] Thread. [Online]. Available: http://www.threadgroup.org/

[11] iOS 9 - HomeKit. [Online]. Available: http://www.apple.com/ios/homekit/

[12] S. Newman, *Building Microservices*. O'Reilly Media, 2015.

[13] E. J. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison Wesley, 2003.

[14] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, Jul. 2010.

[15] A. Cockburn. Hexagonal architecture. [Online]. Available: http://alistair.cockburn.us/Hexagonal+architecture

[16] Circuit breaker pattern. [Online]. Available: https://msdn.microsoft.com/de-de/library/dn589784.aspx

[17] Cloud computing patterns — design patterns — service load balancing. [Online]. Available: http://cloudpatterns.org/design_patterns/service_load_balancing

[18] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, Jul. 2014.

[19] OpenVZ. [Online]. Available: https://openvz.org

[20] Linux containers. [Online]. Available: https://linuxcontainers.org/

[21] rkt, A security-minded, standards-based container engine. [Online]. Available: https://coreos.com/rkt/

[22] (2016, May) Windows containers quick start. [Online]. Available: https://msdn.microsoft.com/en-us/virtualization/windowscontainers/quick_start/quick_start

[23] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support PaaS," in *2014 IEEE International Conference on Cloud Engineering (IC2E)*, Mar. 2014, pp. 610–614.

[24] Boot2docker by boot2docker. [Online]. Available: http://boot2docker.io/

[25] RancherOS. [Online]. Available: http://rancher.com/rancher-os/

[26] W. Shi and S. Dustdar, "The Promise of Edge Computing," *Computer*, vol. 49, no. 5, pp. 78–81, May 2016.

[27] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. ACM, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[28] Maven - concierge OSGi - an optimized OSGi r3 implementation for mobile and embedded systems - overview. [Online]. Available: http://concierge.sourceforge.net/

[29] S. Kächele, J. Domaschka, H. Schmidt, and F. J. Hauck, "nOSGi: a posix-compliant native OSGi framework," in *Proceedings of the 5th International Conference on Communication System Software and Middleware*, New York, NY, USA, 2011, pp. 4:1–4:2.

[30] bliki: ImmutableServer. [Online]. Available: http://martinfowler.com/bliki/ImmutableServer.html

[31] Using blue-green deployment to reduce downtime and risk | cloud foundry docs. [Online]. Available: https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html

[32] bliki: CanaryRelease. [Online]. Available: http://martinfowler.com/bliki/CanaryRelease.html