Institute of Architecture of Application Systems

Internet of Things Patterns

Lukas Reinfurt[1,2], Uwe Breitenbücher[1], Michael Falkenthal[1], Frank Leymann[1], Andreas Riegg[2]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

[2]Daimler AG, Stuttgart, Germany
{firstname.lastname}@daimler.com

Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2016. Internet of Things Patterns. *EuroPLoP'16*, , Article (), 21 pages.
DOI: 10.1145/3011784.3011789

BibTeX

```
@inproceedings{Reinfurt.2016,
  author    = {Reinfurt, Lukas and Breitenb{\"u}cher, Uwe and
Falkenthal, Michael and Leymann, Frank and Riegg, Andreas},
  title     = {Internet of Things Patterns},
  booktitle = {Proceedings of the 21st European Conference on
Pattern Languages of Programs (EuroPLoP)},
  year      = {2016},
  publisher = {ACM},
  doi       = {10.1145/3011784.3011789}
}
```

© ACM 2016
This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version is available at ACM: http://dx.doi.org/10.1145/3011784.3011789

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Universität Stuttgart
Germany

Internet of Things Patterns

LUKAS REINFURT, Daimler AG
UWE BREITENBÜCHER, University of Stuttgart
MICHAEL FALKENTHAL, University of Stuttgart
FRANK LEYMANN, University of Stuttgart
ANDREAS RIEGG, Daimler AG

The development of the Internet of Things is gaining more and more momentum. Due to its widespread applicability, many different solutions have been created in all kinds of areas and contexts. These include solutions for building automation, industrial manufacturing, logistics and mobility, healthcare, or public utilities, for private consumers, businesses, or government. These solutions often have to deal with similar problems, for example, constrained devices, intermittent connectivity, technological heterogeneity, or privacy and security concerns. But the diversity makes it hard to grasp the underlying principles, to compare different solutions, and to design an appropriate custom implementation in the Internet of Things space. We investigated a large number of production-ready Internet of Things offerings to extract recurring proven solution principles into Patterns, of which five are presented in this paper. These Patterns address several problems. DEVICE GATEWAY shows how to connect devices to a network that do not support the network's technology. DEVICE SHADOW explains how to interact with currently offline devices. With a RULES ENGINE, you can create simple processing rules without programming. DEVICE WAKEUP TRIGGER allows you to get a disconnected device to reconnect to a network when needed. REMOTE LOCK AND WIPE can secure devices and their data in case of loss.

CCS Concepts: • **Computer systems organization** → Embedded and cyber-physical systems; • **Software and its engineering** → Design Patterns

Additional Key Words and Phrases: Internet of Things

ACM Reference format:

Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2016. Internet of Things Patterns. *EuroPLoP'16*, , Article (), 21 pages.
DOI: 10.1145/3011784.3011789

1. INTRODUCTION

In the last years, the *Internet of Things (IoT)* has gathered more and more attention in very different areas. It is driven by several developments, such as decreasing sensor and device sizes, energy consumption, or cost of chips and sensors. Additionally, widespread broadband connectivity and new communication technologies are also pushing the IoT forward. A future where many things will be connected to the internet seems increasingly palpable. This, in turn, would allow us to collect and analyze data about practically all aspects of our lives. The gathered knowledge could then be used for widespread improvements and automation.

Author's address: Lukas Reinfurt and Andreas Riegg: Epplestraße 225, 70546 Stuttgart, Germany; email: [firstname].[lastname]@daimler.com; Uwe Breitenbücher, Michael Falkenthal, and Frank Leymann: Universitätsstraße 38, 70569 Stuttgart, Germany; email: [firstname].[lastname]@iaas.uni-stuttgart.de.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
EuroPLoP '16, July 06 - 10, 2016, Kaufbeuren, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4074-8/16/07…$15.00
DOI: http://dx.doi.org/10.1145/3011784.3011789

There are a few core components that are combined to realize IoT systems, as shown in Figure 1. Central to the IoT are the *things*, which usually resemble some kind of device. These devices are often limited in their capabilities due to cost, size, energy, or technological constraints. The typical device contains a combination of *sensors* and/or *actuators*, a *processing component*, some means of *communication*, and an *energy supply*. Sensors are used to translate changes in the environment to electrical signals, whereas actuators are used to act on the environment by translating electrical signals into some kind of physical action [Anjanappa et al. 2002]. They are controlled by the processing component, which can range from a simple circuit to complex chips. A device can also communicate with other components through wired or wireless communication technologies. These other components could be, for example, other devices or a backend server which runs in a data center or in the Cloud. A *backend server* is usually used to aggregate and process data from many devices. It uses this data to gain new insights and knowledge as well as to send commands to the actuators connected to the devices. It is also used to manage all the connected devices, e.g. for registering new devices, updating software and firmware, or managing security credentials. It might also communicate with *other components*, such as web services for analytics or data storage provided by other companies.

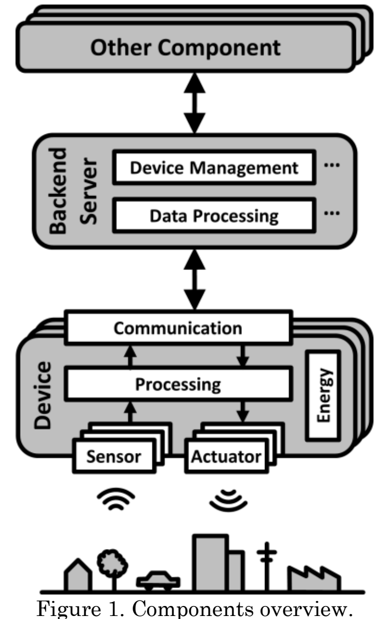


Figure 1. Components overview.

As the IoT is not particular to any specific industry or domain, many different movements or solutions have been developed over time that in some way incorporate the IoT. These include *Smart Homes*, *Smart Offices* [Le Gal et al. 2001; Röcker 2010], *Smart Grids* [Kopp et al. 2015], or the *Smart City* concept [Nam and Pardo 2011; Su et al. 2011], as well as initiatives like *Industrie 4.0* in Germany [Kagemann et al. 2013], or the *Industrial Internet* [Industrial Internet Consortium 2015]. They all do essentially the same on a different scale: They integrate a manifold of yet independent, distributed, and, sometimes, also physically accessible sensors in public environments to achieve two things: (i) to enable analyzing the gathered data jointly and (ii) to use the processed analysis results to automate control of domain specific actuators. All these movements share some significant similarities but have been mainly developed in closed off silos in the past. Several standardization efforts have been initiated that try to break up these silos on different levels. They include network connectivity standards [Bluetooth 2016; Thread Group 2016; ZigBee Alliance 2016; Z-Wave Alliance 2016], protocols [IETF 2014; OASIS 2014; OPC Foundation 2016], device management [Bernstein and Spets 2004, Open Mobile Alliance 2015a, 2015b] or device communication frameworks [Object Management Group 2015; AllSeen Alliance 2016; Open Interconnect Consortium 2016]. It still remains to be seen if all of these efforts can lead to a more unified IoT.

Getting to grips with all these developments is a challenge for companies. Because of the fragmented nature of the IoT space, it is not enough for them to look at different providers, solutions, and technologies in one IoT sector. Instead, they have to look in multiple separate sectors to find the most appropriate solution. Most corporations will come in contact with the IoT on one or multiple levels. A company might realize that it has to produce IoT-enabled products in the future to stay competitive. It might be able to save costs by introducing *Smart Factory* or *Smart Office* capabilities. It might find entirely new business opportunities that are connected to the IoT. When trying to build a good IoT solution, IT architects and developers at these companies are faced with the problems:

- how to conceive application architectures to be robust for IoT challenges, i.e., how to receive and process data from a huge amount of sensors at the same time,
- how to assure security in terms of communication of devices as well as physical access to these devices,
- how to deal with energy and processing limitations of devices, and
- how to integrate multiple proprietary protocols supported by heterogeneous devices, sensors, and actuators into an IoT platform.

However, the prerequisite to tackling these issues is to understand the *core design principles* for developing IoT solutions. It is, therefore, valuable to extract and author a collection of proven design principles from production ready IoT solutions, which are already established in many IoT-platforms and related technologies.

Design Patterns have been used before to describe proven best practices that have stood the test of time in a specific domain. Examples include Patterns for architecture [Alexander et al. 1977], Cloud Computing [Fehling et al. 2014], software design [Gamma et al. 1995], or messaging systems [Hohpe and Woolf 2004]. Their abstraction of very similar and often reoccurring solutions into a structured form can be helpful to dissect and understand complex fields. They are also useful for comparing different solutions and solution providers for suitability for a specific task. Last but not least they can be used as a guideline for new implementations. Therefore, we present in this paper five Patterns for the IoT, as seen in Table 1, aimed at IT architects and developers. We have abstracted them from a systematic information collection process in the area of IoT-platforms and related technologies. We believe that on an abstract level, these Patterns can help companies and individuals to better understand different aspects of the IoT.

Pattern	Description
DEVICE GATEWAY (p. 6)	Some devices cannot directly connect to a network because they do not support the required communication technologies. These devices can be connected through a translating gateway.
DEVICE SHADOW (p. 9)	Other components can interact with currently offline devices by communicating with a persistently stored virtual representation of the device that is synchronized once the device reconnects.
RULES ENGINE (p. 11)	Users can define simple rules without needing to program. These rules tell the system with what action it should react to incoming events.
DEVICE WAKEUP TRIGGER (p. 13)	A device that is not currently connected to the backend server can be informed to do so by sending a message to a low-power communication channel where the device listens for such messages.
REMOTE LOCK AND WIPE (p. 16)	When a device is lost or stolen, its functionality can be remotely locked or data on it can be wiped, either fully or partially, to protect it from possible attacks.

Table 1. Overview of the presented Patterns

The remainder of this paper is structured as follows: Section 2 elaborates on how the Patterns presented in this paper have been identified. Section 3 briefly describes the Pattern format that is used for these Patterns. Section 4 introduces some definitions that are helpful in the scope of the Internet of Things and that are frequently used in the Pattern descriptions. Section 5 presents five IoT Patterns that we identified. Section 6 presents related work in the field of Patterns and the Internet of Things. Section 7 summarizes the paper and gives an outlook on planned future research.

2. PATTERN IDENTIFICATION PROCESS

The Patterns presented in this paper have been identified by collecting and reviewing information from existing products and technologies following the Pattern authoring process defined by [Fehling et al. 2015a]. These solutions were not limited to a specific sector or user group. They include commercial

and open-source solutions for enterprises, developers, and end users alike. The exact sources for each Pattern are detailed in the respective Pattern's example section and include:

- **Product pages** that describe the functionality of IoT solutions
- **User manuals** that explain how to use IoT solutions
- **Technical documentation** of IoT solutions intended for developers
- **Standard documents** of technologies used in IoT solutions
- **Whitepapers** of companies that provide IoT solutions
- **Research Papers** that investigate technologies used in IoT solutions

References from these sources have been collected and grouped when reoccurring solutions became apparent. These groupings represent rough Pattern indicators and ideas. For each grouping, the essence and core principles contained in the considered sources have been abstracted to form a high level, provider independent description of the particular solution. Using these descriptions, Patterns have been authored that conform to the Pattern format described in the next section. The resulting Patterns and Pattern candidates (future Patterns which have not yet been fully formulated) have further been categorized into groups of potentially related Patterns, which form the first step towards a Pattern language for the IoT.

3. PATTERN FORMAT

This section describes the Pattern format that is used to describe the Patterns presented in this paper. It is based on Pattern formats, approaches, and guidelines described in several publications about Pattern writing or publications that contain Patterns [Meszaros and Doble 1996, Harrison 2006b, 2006a; Wellhausen and Fießer 2012; Fehling et al. 2014; Fehling et al. 2015b]. While some elements are required in every Pattern description, others are optional and are only used when necessary.

The **Name** is used to identify the Pattern. Other names by which the Pattern might be known in the industry are listed under **Aliases**. Additionally, the **Icon** adds a graphical representation of the Pattern that is intended to be used in architecture diagrams or sketches [Fehling et al. 2014]. The **Problem** section captures the core problem that is resolved by the Pattern in an abstract manner, i.e., independent from a concrete domain or technology since the general problem might exist in many different use cases. Thus, more technical Patterns [Falkenthal et al.] are out of the scope of this work. The **Context** then further describes the circumstances in which the problem typically occurs, which might impose constraints on the solution. Next, the **Forces** state the considerations that must be taken into account when choosing a solution to a problem. These can often be contradictory.

The **Solution** states the core steps to solve the problem and is often closed with a sketch depicting the architecture of the solution. Then, the **Result** section elaborates the solution in greater detail and describes the situation we find ourselves in after applying the Pattern. **Variants** of the Pattern are listed if they don't differ enough to need their own separate Pattern description. Connections between Patterns, such as Patterns that are often applied together or Patterns that exclude each other can be listed in the **Related Patterns** section. A final **Example** section lists concrete examples that illustrate the application of the Pattern and could also contain links to concrete solution artifacts as conceptually introduced in [Falkenthal et al. 2014b] and validated for different domains [Falkenthal et al. 2014a].

4. TERMINOLOGY AND DEFINITIONS

In this section, we define the basic terminology used to describe the IoT Patterns following Bormann et al. [2014], who presented a terminology for constrained-node networks. The terminology defines different (i) device types, (ii) device energy supply types, and (iii) device operation modes. The following is a short summary to provide a clear understanding of the presented Patterns.

4.1 Device Types

Devices in the IoT can be categorized into groups according to their computational and communication capabilities.

Unconstrained Devices have no significant constraints regarding their computational and communication capabilities. They are able to run arbitrary software and can use communication technology that is not specifically designed for low energy consumption, limited storage, or limited performance.

Semi-Constrained Devices are constrained in their computational power and/or storage space in a way that they cannot use a common full protocol stack to communicate over the internet. However, they can use protocol stacks that are specifically designed for *Semi-Constrained Devices*, such as the Constrained Application Protocol (CoAP)[1], IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN)[2], or Open Platform Communications Unified Architecture (OPC UA) Binary[3]. This enables them to act as fully integrated peers in a network without the help of a gateway or similar components. These nodes often also have a limited energy supply.

Constrained Devices are severely constrained in their computation, storage, and communication capabilities, often caused or accompanied by strong limitations of their energy supply. Therefore, they do not have the resources to support direct internet communication. Consequently, they use communication technology specifically designed for constrained devices, such as Bluetooth Low Energy[4], ZigBee[5], or Z-Wave[6].

4.2 Device Energy Supply Types

The energy supplies available for devices in the IoT can be divided into four groups.

Mains-Powered devices have no direct limitation to available energy, i.e., they are plugged into a wall socket. Unless there is an outage, they can use all the power they need.

Period Energy-Limited devices have a power source that has to be replaced or recharged in regular intervals, such as easily replaceable or rechargeable batteries or fuel in some kind of generator.

Lifetime Energy-Limited devices contain a non-replaceable and non-rechargeable power source, such as a battery that is directly soldered onto the circuit board. Once this power source is depleted it cannot be easily replaced.

Energy Harvesting devices convert ambient energy into electrical energy. Ambient energy can be in form of radiant energy (solar, infrared, radio-frequency), thermal energy, mechanical energy, or

[1] https://tools.ietf.org/html/rfc7252 (last accessed on 07.11.2016)
[2] https://tools.ietf.org/html/rfc4944 (last accessed on 07.11.2016)
[3] https://opcfoundation.org/ (last accessed on 07.11.2016)
[4] https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy (last accessed on 07.11.2016)
[5] http://www.zigbee.org/ (last accessed on 07.11.2016)
[6] http://www.z-wave.com/ (last accessed on 07.11.2016)

biomechanical energy. The energy available to the device depends on the ambient energy available at the location of the device and might vary significantly over time. Energy harvesting can supply a device with perpetual power in some cases, but the available amount of energy is usually very small. Often, these devices will be mostly sleeping while they collect enough energy for short bursts of activity.

4.3 Device Operation Modes

Devices can operate in different modes depending on their communication frequency and their need to save energy.

Always-On devices have no reason to change operation modes to save power. They can stay connected and operational all the time.

Low-Power devices usually need to operate on small amounts of power but are still required to communicate frequently. They will sleep for short periods of time between communicating, but will generally stay connected to the network. This requires optimized hardware and communication solutions.

Normally-Off devices will be asleep most of the time and reconnect to the network at specific intervals to communicate (duty cycling).

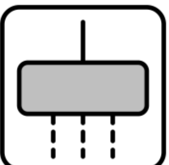
5. INTERNET OF THINGS PATTERNS

In this section, we present five IoT Patterns that were identified following the procedure described in Section 2. The format follows the definition presented in Section 3.

5.1 Device Gateway

Aliases: Gateway, Field Gateway, Intermediate Gateway, Physical Hub, Protocol Converter

Context: A number of devices have to be connected to a network. These might include *Constrained Devices* or *Semi-Constrained Devices* that are limited in their processing power and do not support the communication methods of the network. Or these might also include *Unconstrained Devices* from legacy systems that cannot connect to the network due to outdated technology. A backend server reachable over this network is intended to process data from these devices.



Problem: You want to connect many different devices to an already existing network, but some of them might not support the networks communication technology or protocol.

Forces:

- **Connectivity:** Devices have to be connected to a network because you want to access their data and functionality regularly. Doing this manually is not an option.
- **Upgradability:** Changing or building up a network so that it supports the communication technology required by the device is often not possible. You might not control the network, or the purpose of the network cannot be realized with the device's technology, e.g. you need a long range network but the device only supports short range communication.
- **Effort:** Adding communication capabilities that are supported by the network to all device types would mean a high investment in time and resources, or might not be possible at all be-

cause of technological limitations.
- **Diversity:** Other devices with different communication technology might also have to be connected to the same network and will face the same problem.
- **Device Numbers:** Your network can only support a certain amount of simultaneous connections. The number of devices you want to connect exceeds this limit. Extending the network is not an option.

Solution: Connect devices to an intermediary DEVICE GATEWAY that translates the communication technology supported by the device to communication technology of the network and vice-versa.

Result: A DEVICE GATEWAY is usually a dedicated hardware appliance that can translate between a number of heterogeneous communication technologies. In many cases, it will be located at the edge of the network, close to the devices which it connects to the backend. It is possible to integrate a DEVICE GATEWAY into the backend, but this is often not practical. It is often used to translate low-power short-range communication to IP communication, so it has to be located close to these devices, whereas the backend is usually located far away.

For communication translation, it has to support at least two, but more commonly multiple communication technologies. On the interface towards the backend it usually supports IP communication over Ethernet, Wi-Fi, or mobile networks. On the interfaces towards the devices, it usually supports some kind of low energy communication technology. Depending on its application, it might also contain additional interfaces supporting other protocols. A translation layer converts messages received from either the backend or the devices to messages that can be sent to the respective opponent interface and vice versa. To be able to route the messages to their intended receivers the messages have to container some kind of identifier.

Benefits:
- **Connectivity:** Devices that do not directly support the networks communication technology can be connected to the network.
- **Separation of Concerns:** Device implementations can focus on only one arbitrary protocol or technology, which makes them simpler. On the other hand, the DEVICE GATEWAY can be optimized for protocol translation.
- **Effort:** One DEVICE GATEWAY can support multiple different communication technologies. The devices don't have to be modified.
- **Cost:** Many devices can be connected to a network via one DEVICE GATEWAY, without needing to support multiple communication technologies over the whole network, which saves costs.
- **Reusability:** It might be possible to reuse existing hardware as a DEVICE GATEWAY, for example, smartphones or routers, which might further decrease the effort and cost needed.
- **Technological Limitations:** Devices can use very limited communication technology in the form of a specifically reduced software stack. So they can exploit their limited power else-

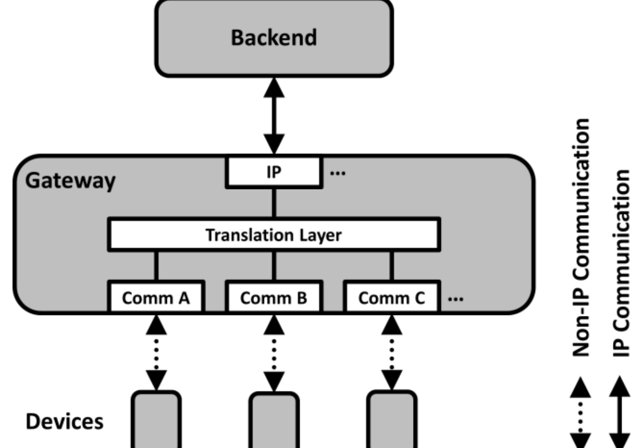


Figure 2. Exemplary sketch of the DEVICE GATEWAY pattern used to transform to and from IP communication.

where, while still being able to connect to a network that requires more sophisticated technology through a DEVICE GATEWAY.

- **Additional Functionality:** A DEVICE GATEWAY might have enough resources to be able to implement additional functionality, such as management or monitoring capabilities, data aggregation or filtering, or enhanced security mechanisms.
- **Resilience:** A DEVICE GATEWAY with additional local functionality, like a RULES ENGINE and a backup battery, can add a layer of resilience and keep local processes running regardless of power or network outages and backend server failures.

Drawbacks:

- **Connectivity:** The DEVICE GATEWAY might become a single point of failure for the network connectivity of the connected devices. Adding redundant DEVICE GATEWAYS with a failover mechanism could alleviate this problem, but at an increased cost.
- **Security:** As a single point of attack the DEVICE GATEWAY also poses a security risk. If compromised, an attacker could gain access to all attached devices or the backend server.
- **Complexity:** Another layer of components is introduced that has to be managed and maintained. This becomes even more difficult if multiple kinds of gateways are used.
- **Cost:** The DEVICE GATEWAY usually has to support multiple communication technologies and, thus, needs more processing power, which makes it expensive. In addition, if devices are distributed, possibly multiple DEVICE GATEWAYS are required to connect all of them. Costs might be reduced by using a modular DEVICE GATEWAY design, where only the required technologies can be added with extension boards.
- **Compatibility:** Some technologies might be incompatible on a conceptual level. A DEVICE GATEWAY might only be able to create a partial translation between these technologies, or it might not be able to translate between certain technologies at all.

Variants: Common variants of a pure DEVICE GATEWAY usually include some kind of local processing power. Some examples are listed below. They are not mutually exclusive and can be combined.

- **Aggregating Device Gateway:** Besides translating communication technologies this gateway also aggregates the messages it receives from the devices in some meaningful way. For example, it might average the temperature readings of several devices and send it on once a minute. This is usually done to reduce the number of individual messages which have to be sent to the backend.
- **Local Processing Device Gateway:** In addition to translating communication technologies, this gateway also contains some local processing functionality which could mirror or replace functionality located in the backend. For example, it could contain a local RULES ENGINE which decides some actions directly on the gateway. This is usually done to minimize communication with the backend and therefore reduce latency or to insulate from connection loss between gateway and backend.

Related Patterns:

- **MESSAGE GATEWAY:** The MESSAGE GATEWAY Pattern is similar to the DEVICE GATEWAY, but describes how one or more gateways can be used to combine several different messaging technologies in a single machine [Eloranta et al. 2014b].

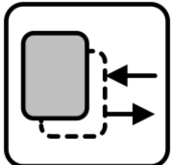
- **ADAPTOR:** The DEVICE GATEWAY can be seen as a physical version of the ADAPTOR Pattern that describes how two incompatible interfaces can work together by converting one interface to the other [Gamma et al. 1995].
- **RULES ENGINE:** A DEVICE GATEWAY might contain a RULES ENGINE to trigger actions locally. This can prevent unnecessary round trips to a remote server and might decrease latency.

Examples: Central hubs are a common occurrence in the product portfolios of home automation companies. Here they often act as an indispensable central point for integrating and managing the actual home automation devices. Examples are the Samsung SmartThings Hub [SmartThings 2015] which supports ZigBee, Z-Wave, and IP, or the Wink Hub [Wink 2015] that additionally supports Bluetooth Low Energy and Lutron Clear Connect[7]. The SmartThings Hub v2 also introduced local processing capabilities, which is also supported by other DEVICE GATEWAYS like the THNGHUB [EVRYTHNG 2016]. Various companies offer development kits and appliances to implement DEVICE GATEWAYS for industrial use, such as Intel, Dell, or Nexcom [Dell 2016; Intel 2016; Nexcom 2016]. The Eclipse Kura project is an Open Source framework for building the software side of DEVICE GATEWAYS [Eclipse Foundation 2016]. Zachariah et al. [2015] proposed to use smartphones with Bluetooth Low Energy as universal gateways for other devices. In a way, smartphones are already used as DEVICE GATEWAYS for many wearable devices, like fitness trackers or smartwatches, which completely rely on the smartphone to communicate the data they collected to the backend. Many IoT platform documentations mention physical hubs or field gateways as a way to connect devices to their platforms that cannot connect to the internet on their own, even though they do not offer any products or solutions in this space [Amazon Web Services 2015a; Bosch Software Innovations 2015; Comarch Technologies 2015a, Microsoft 2015a, 2015b]. Thus, these follow the idea of DEVICE GATEWAYS.

5.2 Device Shadow

Aliases: Thing Shadow, Virtual Device

Context: Devices, such as *Constrained Devices*, *Semi-Constrained Devices*, and *Unconstrained Devices,* might operate in *Normally-Off*, *Low-Power*, or *Always-On* modes. Either because of their operation modes or because of external circumstances, these devices might be offline at various times.



Problem: Some devices will be only intermittently online in order to save energy or because of network outages. Other components want to interact with them but do not know when they will be reachable.

Forces:

- **Availability:** Sending commands to or reading state from offline devices is not possible.
- **Timeliness:** Waiting for currently offline device to come online again to send or receive data in a synchronous fashion can lead to long idle times and should be avoided.
- **Consistency:** Often a slightly out-of-date state is better than no state.

Solution: Store a persistent virtual representation of each device on some backend server. Include the latest received state from the device, as well as commands not yet sent to the device. Do all communication from and to the device through this virtual version. Synchronize the virtual representation with the actual device state when the device is online.

[7] http://www.lutron.com/en-US/Residential-Commercial-Solutions/Pages/Residential-Solutions/IntegrationConnectivity.aspx (last accessed on 07.11.2016)

Result: By storing persistent virtual representations of the devices on the backend server and communicating only through those, device communication can be decoupled. This allows reading device state as well as sending device commands even if the device is offline. Essential to this is a persistent storage on the backend that can store virtual device representations reliably for many devices and that can handle read and write access from multiple sources. If commands are saved they should be queued, unless only the newest command is regarded as relevant. When a device reconnects to the backend, which can happen according to a schedule or based on certain events, it can retrieve and process the stored command and update the last known state. To let other components know that a device is online, a flag can be stored with the device shadow. When a device connects or gracefully disconnects it enables or disables this flag itself. Otherwise, the flag is set to false after a certain time of inactivity or by another mechanism, for example by the last will and testament of the MQTT protocol.

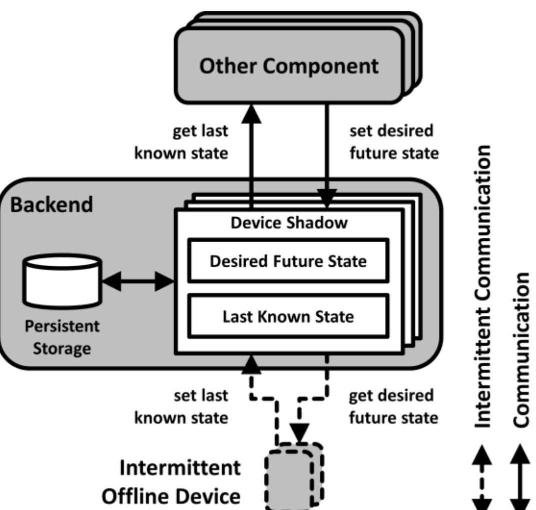


Figure 3. Sketch of the DEVICE SHADOW pattern.

Conceivably, DEVICE SHADOW functionality could also be implemented on DEVICE GATEWAYS to allow localized decoupling between devices connected to one DEVICE GATEWAY. This would bring the benefits of a DEVICE SHADOW to these devices, even if the Gateway might be disconnected from the rest of the network from time to time. A problem here could be that a DEVICE GATEWAY might not be able to reliably provide the persistent storage that is needed.

Benefits:

- **Unified Handling:** The communication with devices can be handled as if they are *Always-On*, even if they really are not. Therefore, time autonomy between backend and devices is established.
- **Additional Functionality:** If all communication goes through a DEVICE SHADOW, additional functionality can be implemented, such as batch messaging, filtering, or caching.
- **Security:** By only communicating with a single, well-known target, security can be increased, because devices can categorically deny communication attempts from any other source.

Drawbacks:

- **Eventual Consistency:** The virtual device representation is only eventually consistent with its actual state.
- **Synchronization Issues:** State updates could be lost if a new state update is written to the device shadow that is based on a state that is older than the current last known state. One way to avoid such issues is versioning the states and using OPTIMISTIC OFFLINE LOCK [FOWLER ET AL. 2002].
- **Obsolescence:** By the time an offline device reconnects and receives stored commands, these commands might have become obsolete. To avoid stale commands, the MESSAGE EXPIRATION Pattern [Hohpe and Woolf 2004] can be used.
- **Quality of Service:** If all communication is forced through the backend server, latency and decreased availability for communication that could be done completely local can be a problem.

Related Patterns:

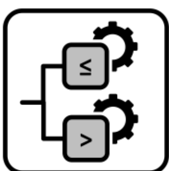
- **Remote Proxy:** Gamma et al. describe remote proxy as one application of the Proxy Pattern. Here, the remote proxy locally represents an object in another address space to hide the fact that the object is remote [Gamma et al. 1995]. Device Shadow can be seen as a device specific version of a remote proxy.

Examples: AWS IoT stores a persistent virtual version of each connected device that includes the last reported state and the desired future state of the device. This allows applications to read and write device state irrespective of the actual availability of the device [Amazon Web Services 2015c, 2015b]. Azure IoT Suite stores device models in a device registry that is an eventually consistent view of device data [Microsoft 2015c]. Kii IoT Platform's Thing Interaction Framework saves the latest state of registered things on the backend server. Applications that request a device's state get the state stored on the server [Kii 2015b].

5.3 Rules Engine

Aliases: Action Engine, Trigger Conditions

Context: A wide range of differing messages from devices and other components are received at the backend server. These might include measurements from sensors, errors, a heartbeat, registration information, etc. These messages can arrive regularly or irregularly. There are different kinds of actions that have to be executed depending on the type of the received message, its content, the time it is received, or other factors.



Problem: Throughout its operation, a system receives a wide range of messages from devices and other components. You want to react in different ways to these messages.

Forces:

- **Flexibility**: The actions to trigger might change over time, new actions might be added, old ones removed, or you might want to temporarily test or disable an action. Hard-coding them into some software component would be possible, but is not flexible enough.
- **Data Sources:** In some cases, additional data apart from the device message might be needed to decide if a particular action should be taken.
- **Diversity:** The type of action to be triggered can vary significantly depending on the circumstances. In some cases, you might want to add an entry into a log file or send an email. In other cases, you might want to route a message to another service for further processing or store it in some kind of database.

Solution: Pass all messages received from devices through a Rules Engine. Allow users to define rules using a graphical user interface that evaluate the content of incoming messages or metadata about the message against a set of comparators. Also allow external data sources to be included in these comparisons. Let users associate a set of actions with these rules. Apply each rule on each message and trigger the associated actions if a rule matches.

Result: A RULES ENGINE contains a set of rules and actions that should be executed if a particular rule is met. Usually, these rules and associated actions are user definable through a graphical user interface on the backend server. During operation, each incoming message is compared against these rules. If a rule matches, the associated action is triggered. RULES ENGINES are often located on a central backend server but can also be located on a DEVICE GATEWAY.

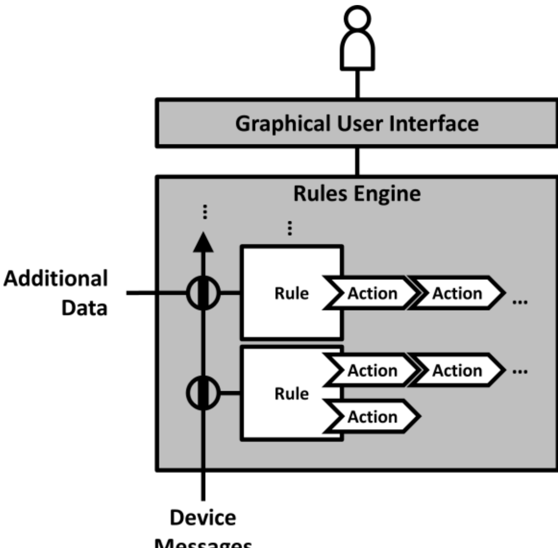


Figure 4. Sketch of the RULES ENGINE pattern.

The rules usually allow comparing incoming data to static values, historical data, data from other sources, or a combination thereof. Different comparators allow a user to check if incoming data is, e.g., equal to, unequal to, larger than, or smaller than a certain value, or if it contains a certain value. Regular expressions or SQL statements might be allowed for more complex comparisons. Rule matching for a particular message could be stopped after the first match, or it could be continued until all rules are evaluated. It could also be possible to let a rule trigger only once and never again, or only once in a specific time window.

Actions can vary in their scope and complexity. Simple actions might trigger some functionality that is built into the platform that is used, such as sending an alert to a user. They might also act as a router that passes data on to services on the same backend server or to external services of other companies for further processing. One rule could only trigger one action, but it could also be possible to associate multiple actions to one rule that then could be executed in serial or in parallel.

Benefits:

- **Flexibility:** Rules can be flexibly added, changed, temporarily disabled, or removed, because they are not hard-coded into software.
- **Ease of Use:** A graphical user interface allows non-programmers to manage rules.
- **Configurability:** The RULES ENGINE usually offers a wide range of options for how to evaluate the rules and trigger the actions, but simple rules can be configured by users without extensive programming knowledge.
- **Automation:** A RULES ENGINE allows creating automatic responses for certain situations.
- **Analytics:** A RULES ENGINE might track certain values to enable monitoring and analytics on the messages it receives and the rules and actions that are or are not triggered.

Drawbacks:

- **Suitability:** Depending on the functionality offered by the inbuilt rules and actions, a RULES ENGINE might not be suitable for certain complex transformation or routing tasks. A possible way to mitigate this drawback is to support user defined rules and actions via some scripting language.
- **Configurability:** While simple rules are easy to configure, complex rules might require more insight or special training.
- **Security:** A compromised or misconfigured RULES ENGINE can be a security risk.
- **Single Point of Failure:** If all messaged are passed through a RULES ENGINE it becomes a single point of failure.

- **Effort:** Creating and maintaining good rules might be a lot of work. Creating a marketplace for rules could be one solution to decrease effort and duplication and increase efficiency.

Related Patterns:

- **PRODUCTION RULE SYSTEM:** As described by Fowler [2011], a PRODUCTION RULE SYSTEM organizes logic into a set of rules, where each rule has a condition and an action. While the PRODUCTION RULE SYSTEM is just a formalism to represent and organize logic into rules and conditions, the RULES ENGINE is the component that controls the execution of these rules.
- **CONTENT BASED ROUTER:** A RULES ENGINE can be seen as an extended CONTENT BASED ROUTER as described by Hohpe et al. [Hohpe and Woolf 2004]. A CONTENT BASED ROUTER examines only the message content and then routes the message to exactly one system. A RULES ENGINE can route to multiple systems based on the message content, other data, or a combination thereof.

Examples: The AWS IoT Platform includes a RULES ENGINE that can transform and deliver inbound messages to other devices or Cloud services. Its rules can be applied to multiple data sources at once and multiple actions can be triggered in parallel. The rules can be created in an SQL-like syntax [Amazon Web Services 2015c]. IBM IoT Real-Time Insights has an action engine that lets users define automated responses to detected conditions. Inbuilt actions include sending an email, triggering an IFTTT[8] recipe, or executing a Node-RED[9] workflow. Arbitrary other web services can be included with webhooks [IBM 2015b]. Many other IoT Platforms also include a Rules Engine [Ayla Networks 2015; Comarch Technologies 2015b; EVRYTHNG 2015; Kii 2015b; myDevices 2015; Wind River 2015]. There are also standalone services like Waylay, IFTTT, and Zapier or apps like Stringify that offer RULES ENGINE functionality without a complete IoT platform [IFTTT 2016; Stringify 2016; waylay.io 2016; Zapier 2016]. Some RULES ENGINES, like EVRYTHNG's Reactor, can be located on DEVICE GATEWAYS to enable low latency message processing close to the devices [EVRYTHNG 2016].

5.4 Device Wakeup Trigger

Aliases: Update Trigger, Device Triggering

Context: You have a *Constrained Device* or *Semi-Constrained Device* that is *Lifetime Energy-Limited* or *Period Energy-Limited* and operates in a *Low-Power* or *Normally-Off* mode. You have a backend server where the device is registered, i.e. its identity and other metadata is known to the server. From time to time you have a situation where you want to immediately contact the sleeping device. For example, this could be the case if a critical security fix has to be applied, if you need current sensor values or send commands for one-off time critical situations, or if the device has been lost or stolen and you want to use REMOTE LOCK AND WIPE immediately.

Problem: Some devices might go into a sleep mode to conserve energy and only wake up from time to time to reconnect to the network. During sleep, they are not reachable on their regular communication channels. In some instances, other components might have to contact sleeping device immediately.

[8] https://ifttt.com/ (last accessed on 07.11.2016)
[9] http://nodered.org/ (last accessed on 07.11.2016)

Forces:

- **Irregularity:** You need to establish a connection at non-regular times.
- **Predictability:** You do not know the point in time when you need to connect to the device in advance.
- **Timeliness:** The device might reconnect on its own, but you can't wait that long.
- **Power Consumption:** The device has to maintain low power consumption in terms of entering *Low-Power* or *Normally-Off* operation modes to save energy.

Solution: Implement a mechanism that allows the server to send a trigger message to the device via a low energy communication channel. Have the device listening for these triggering messages and immediately establish communication with the server when it receives such a message.

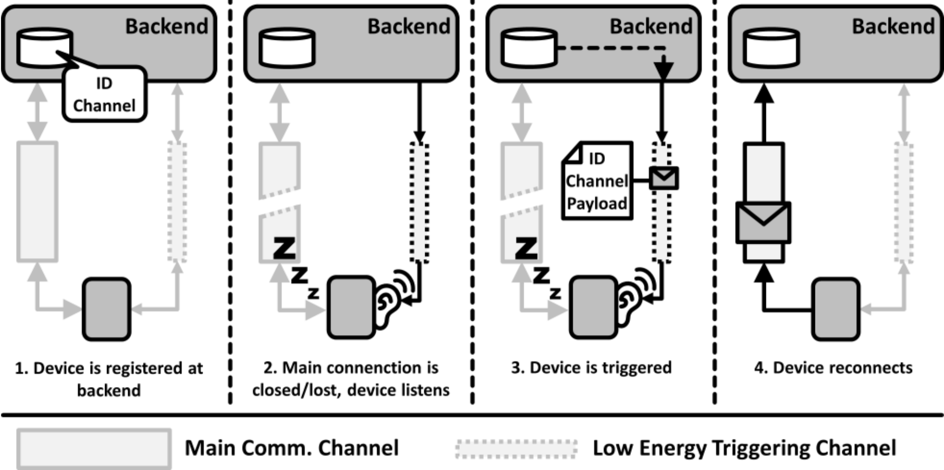


Figure 5. Sketch of the DEVICE WAKEUP TRIGGER Pattern.

Result: A triggerable device can be in a *Low-Power* or *Normally-Off* operation mode, where most of its functionality is dormant. But it is still listening on a specific communication channel for triggering messages using a low energy communication module. When a server wants to wake up a device, it has to know the device and this channel in advance. So a prerequisite for the DEVICE WAKEUP TRIGGER is that the device has previously registered some kind of identifier and its listening channel with the backend server. This can either be done manually when the server or the device is provisioned, or it could be done automatically by the device when it communicates with the server.

If the server wants to initiate communication with a triggerable device, it looks up the device in its registry and uses the stored information to send a trigger message to the channel that the device is listening on. The trigger message can contain a payload, e.g., to trigger some specific action on the device after the wake-up. Depending on the content and the existence of a payload, the triggered device might react in two ways: (i) If a payload was sent, it can process it and send a response to the server without establishing a long lasting connection (piggybacked response). (ii) If no payload was sent or the payload indicates that further communication is needed, the device can establish a long lasting connection to the server and wait for further instructions. The maximum time to wait for further instructions can be configured by a timeout, either directly on the device or in the payload of the wake-up message.

Benefits:

- **Efficiency:** If no constant connection has to be kept alive, it allows the device to operate in a *Low-Power* or *Normally-Off* mode where the only active component is a low energy communication module listening for trigger messages.
- **Responsiveness:** Even though the device can be in a *Low-Power* or *Normally-Off* operation mode most of the time, it can be triggered to reconnect at any time if needed.

Drawbacks:

- **Efficiency:** At least the communication module has to be active to listen for triggering messages. Therefore, the device cannot be turned off completely. To maximize efficiency, a very low power communication module should be used to listen for trigger messages.
- **Cost:** There might be costs associated with sending a trigger message, for example, when using SMS to trigger devices.
- **Infrastructure:** New infrastructure might be needed on the server side for a low energy communication channel which is only used for device triggering.
- **Effort:** The device needs a second communication circuit which increases cost and complexity.

Related Patterns:

- **CORRELATION IDENTIFIER:** A CORRELATION IDENTIFIER can be used when sending and replying to a DEVICE WAKEUP TRIGGER so that the server from which the trigger message originated knows to which trigger message the answer it received belongs [Hohpe and Woolf 2004].

Examples: Device Triggering was introduced in release 11 of the 3rd Generation Partnership Project (3GPP) as a way to allow server initiated communication with UMTS or LTE devices when their IP address is not known. SMS is used as triggering mechanism, but a direct response to the payload is not supported [ETSI 2015]. 3GPP2 also supports Device Triggering using SMS, broadcast SMS, or IP transport [3GPP2 2014]. OneM2M uses these mechanics to trigger devices to wake them up, to force them to establish a connection to the server, or when their IP address is not known [oneM2M 2015]. Starsinic et al. [2015] argue that LTE devices always have an IP address and using SMS as triggering mechanism makes applications using a DEVICE WAKEUP TRIGGER more platform dependent, because they always need to support SMS. Additionally, the lack of direct response to a trigger message requires devices to always establish a connection, which may be inefficient in cases where a simple reply to the trigger messages would have been sufficient. They propose an IP-based triggering method that is LTE backwards-compatible and utilizes UDP packages. It supports direct responses to triggering messages, for example by using CoAP confirmable data packages. Open Mobile Alliance Lightweight Machine to Machine (OMA LWM2M) supports an update trigger mechanism where the server can wake up devices via SMS. An LWM2M client can disconnect if it doesn't receive a message after a certain time but stays reachable via SMS. The LWM2M server queues operations for the client while it is offline. The server can send an update trigger message via SMS to the client. After the client received the SMS it reconnects and receives the queued operations [Open Mobile Alliance 2015a]. The CPE WAN Management Protocol, also known as TR-069, includes a mechanism called asynchronous auto-configuration server-initiated notifications. It allows a configuration server to instruct a device to establish a connection with the server when a new configuration is available [Bernstein and Spets 2004]. An example of products is the PawTrax pet trackers. They stay in a sleep mode to save energy until activated by SMS. As a piggyback response, they send the current location of the pet, but they can also be switched to periodically send the location to an app or web platform [PawTrax 2016].

5.5 Remote Lock and Wipe

Aliases: Remote Factory Reset, Remote Locking, Remote Wiping

Context: A device is connected to a backend server and is in danger of being lost or stolen. This might be the case because it is installed at an easily accessible public location, or a remote and unmonitored location. The device might have functionality that must not be accessed by a thief. It might also contain classified data that has to be kept protected. The data might or might not be encrypted. The device might be retrievable when it is lost or stolen, but it might also vanish forever.

Problem: Some devices might be lost or stolen. You want to prevent attackers from misusing the functionality of the device, or from gaining access to the data on the device or to the network through the device.

Forces:

- **Long-term Data Security:** If the device is irretrievably stolen, an attacker might have ample time to break encryptions if data on the device is encrypted.
- **Fine-grained Control:** Depending on the situation, the type of device and the content on the device, different actions might be necessary in the case of loss or theft.
- **Reversibility:** A lost or stolen device might eventually be returned, so any actions taken should be reversible if possible.
- **Remote Control:** Since the device is no longer physically available, the activation of additional security mechanisms has to work remotely.

Solution: Make the device a managed device that can receive and execute management operations from the backend server. Allow authorized users to use the backend server to trigger functionality on the device that can delete files, folders, applications or memory areas, revoke or remove permissions, keys, and certificates, or enable additional security feature. Execute triggered functions as soon as the device receives them and provide an acknowledgment to the backend.

Result: To be able to offer REMOTE LOCK AND WIPE functionality, a device has to be a managed device that is connected to a management backend, which is a component on the backend server that can remotely execute management functionality on the device. Once an authorized user successfully authenticated to the backend, he or she can choose between different lock or wipe options depending on the circumstances. Which exact options are provided depends on the particular device. The device should provide a list of lockable or deletable data and functionality to the backend server.

In some circumstances, it might be enough to only disable some functionality but leave on location tracking to facilitate the retrieval of a lost or stolen device. The user might also only erase certain sensitive data to prevent data theft. In more severe cases, he or she might reset the device to its factory state, which would leave it operational but without any data on it. He or she might also completely disable the device to make it unusable.

Wiping data can be done by utilizing existing functionality to delete files and folders, or by directly deleting certain memory areas. Data can also be encrypted with a key stored on the device which is used by applications to access this data. When this key is deleted, access to this data is effectively revoked. Functionality can be locked by revoking permissions, keys, or certificates that are required for execution, or by enabling security checks that were previously not enabled. Functionality could also be

completely removed by deleting the associated code from the device. Once the requested operations are executed, the device should send back an acknowledgment to the backend server if possible.

Benefits:

- **Long-Term Data Security:** Wiping sensitive data from the device prevents an attacker from stealing the data, even when he has enough time to circumvent some kind of encryption.
- **Fine-grained Control:** Partially or fully locking or wiping and full factory reset allow reactions appropriate to the situation and the sensitivity of the data on the device, or its functionality.
- **Reversibility:** Locked device functionality can be unlocked if the device is retrieved.
- **Remote Action:** To execute lock and wipe functionality the device doesn't have to be under physical control. It only has to be connected to the backend so that the lock and wipe functionality can be triggered.

Drawbacks:

- **Reversibility:** Wiped data and a factory reset cannot be reversed. A backup mechanism could be used to be able to restore at least some data.
- **Connectivity:** The device has to be connected to receive the REMOTE LOCK AND WIPE commands. A DEVICE WAKEUP TRIGGER could be used to get the device to connect to the backend server.
- **Security:** If attackers gain access to the REMOTE LOCK AND WIPE functionality they could lock devices for ransom or wipe or disable them to cause damage. Proper authentication and authorization mechanisms, as well as end-to-end encryption, should be used at all times.

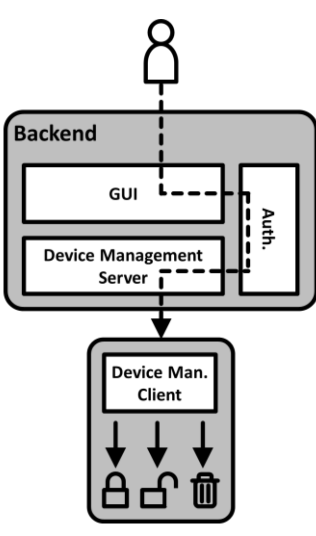


Figure 6. Sketch of the REMOTE LOCK AND WIPE pattern.

Related Patterns:

- **DEVICE WAKEUP TRIGGER**: A DEVICE WAKEUP TRIGGER could be used to get the device locked or wiped as soon as possible if it is currently not connected to the backend server.

Examples: Functionality to remotely locate, lock or wipe a phone is common on modern smartphones. Android phones can be located, set to ring, locked, or erased remotely with the Android Device Manager website or app [Google 2016]. Apple offers similar functionality through the iCloud [Apple 2016b, 2016a]. Options for other kinds of devices do also exist. The OMA LWM2M standard specifies a Lock and Wipe object. It supports functionality for partially or fully locking a device, for partially or fully wiping data on a device, and for doing a factory reset. These operations can be performed with or without user confirmation or notification [Open Mobile Alliance 2015a]. The Kii IoT Platform allows users to lock and unlock devices over their web interface. When locked, the device is not able to access its data resources in the Cloud, while the owner and admin users still have access to these resources [Kii 2015a]. TR-069 and the IBM IoT Foundation Platform both support remote factory reset functionality [Bernstein and Spets 2004; IBM 2015a].

6. RELATED WORK

The concept of Patterns, as introduced by Alexander et al. [1977] is of course nothing new. Over the years, many publications were made that either include new Patterns for a specific field or talk about the Pattern creation process in general. A selection of the latter was already mentioned in Section 3

and includes [Meszaros and Doble 1996, Harrison 2006b, 2006a; Wellhausen and Fießer 2012; Fehling et al. 2014; Fehling et al. 2015b]. Additional publications include [Reiners et al. 2013; Fehling et al. 2015a; Falkenthal et al. 2016]. Further, research about efficient Pattern application via Pattern refinement and concrete solutions organized in solution repositories emerges [Falkenthal et al; Falkenthal et al. 2014a].

Some Patterns for topics in IoT or related areas exist. Eloranta et al. describe Patterns for building distributed control systems for moving machinery used for foresting, mining, construction, etc. These Patterns focus on aspects of reliability and fault-tolerance within these large machines but are not concerned with communication between small, *Constrained Devices* [Eloranta et al. 2014a]. Qanbari et al. present four design Patterns for edge application provisioning, deployment, orchestration, and monitoring. In addition to their narrow focus on edge applications, these Patterns use existing technologies like Docker and Git which are not suited for *Constrained Devices* [Qanbari et al. 2016].

Publications in other contexts exist that contain Patterns that are applicable in the IoT domain. The *Messaging Patterns* by Hohpe et al. [2004] contain several Patterns that can be used to describe communication aspects in the IoT. For example, the COMMAND MESSAGE and EVENT MESSAGE Patterns fit neatly with the two types of messages that are exchanged in the IoT, namely messages that are sent to devices that contain a command, e.g., to activate some kind of actuator, and messages that are sent from devices to the backend for further processing by other components, e.g., sensor values. Other Patterns that are applicable include EVENT-DRIVEN CONSUMER, PUBLISH-SUBSCRIBE CHANNEL, or GUARANTEED DELIVERY. But these Patterns only cover some aspects of IoT communication.

The *Cloud Computing Patterns* by Fehling et al. [2014] also contain some Patterns that are applicable in the IoT domain. For example, a variant of the WATCHDOG Pattern can be found on DEVICE GATEWAYS where it resets the system when it detects a problem with a critical component [Eclipse Foundation 2015]. The EXACTLY-ONCE DELIVERY and AT-LEAST-ONCE DELIVERY Patterns apply to device communication, for example when the Message Queue Telemetry Transport (MQTT)[10] protocol is used. The different workload Patterns could be used to describe workloads generated by device messages and the LOOSE COUPLING Pattern discusses principles to decouple devices from other components that consume their data or trigger some actuator functionality of the device, respectively. Again, these Patterns only cover some aspects which are relevant for IoT.

7. SUMMARY AND OUTLOOK

The vision of the Internet of Things has been around for some years, but the widespread realization of this vision is just now starting. The field of IoT solutions is still very much in a state of flux, as different companies, research institutes, and other entities try to bring the IoT into reality. This adds another level of complexity to the already pretty tangled mess that the IoT is today, with its numerous technologies, standards, organizations, sectors, and flavors.

To manage this overall complexity, we presented five Patterns in order to help the reader understand some key aspects of the IoT. What these Patterns hint at in some places is that there are more Patterns to be added to this catalog. We have already identified a few more Pattern candidates and more will certainly be found over time. So, we work on expanding this selection of Patterns to a comprehensive Pattern language for the IoT, by also investigating relations between the Patterns to additionally support and guide readers towards the usage of typical Pattern combinations and Pattern refine-

[10] http://mqtt.org/ (last accessed on 25.01.2016)

ments. This would give companies a tool to evaluate different IoT providers, provide developers with guidance when implementing new IoT solutions, and help other interested individuals with understanding different aspects of the IoT.

ACKNOWLEDGEMENTS

We would like to thank our shepherd, Marko Leppänen, for the discussions and comments that helped to improve this paper. This work was partially funded by the BMWi projects NEMAR (03ET4018B), SmartOrchestra (01MD16001F) and SePiA.Pro (01MD16013F).

REFERENCES

3GPP2. 2014. *Network Enhancements for Machine to Machine (M2M)*. (2014). Retrieved January 20, 2016 from http://www.3gpp2.org/public_html/specs/X.S0068-0_v1.0_M2M_Enhancements_20140718.pdf.

Alexander, C., Ishikawa, S., and Silverstein, M. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York.

AllSeen Alliance. 2016. *AllSeen Alliance*. (2016). Retrieved January 28, 2016 from https://allseenalliance.org/.

Amazon Web Services. 2015a. *AWS IoT FAQs*. (2015). Retrieved November 10, 2015 from https://aws.amazon.com/iot/faqs/.

Amazon Web Services. 2015b. *Device Shadows Documents*. (2015). Retrieved November 10, 2015 from http://docs.aws.amazon.com/iot/latest/developerguide/thing-shadow-document.html.

Amazon Web Services. 2015c. *How the AWS IoT Platform Works*. (2015). Retrieved November 9, 2015 from https://aws.amazon.com/iot/how-it-works.

Anjanappa, M., Datta, K., and Song, T. 2002. Introduction to Sensors and Actuators. In *The Mechatronics Handbook*. CRC Press, Boca Raton, Florida, 327–340.

Apple. 2016a. *iCloud: Erase your device*. (2016). Retrieved January 27, 2016 from https://support.apple.com/kb/PH2701.

Apple. 2016b. *iCloud: Use Lost Mode*. (2016). Retrieved January 27, 2016 from https://support.apple.com/kb/PH2700.

Ayla Networks. 2015. *Ayla Architecture. Focusing on the 'Things' and Their Manufacturers*. (2015). Retrieved December 10, 2015 from https://www.aylanetworks.com/wp-content/uploads/2015/06/Ayla_Architecture_White_Paper_preview.pdf.

Bernstein, J., and Spets, T. 2004. *DSL Forum TR-069. CPE WAN Management Protocol*. DSL Forum. (2004). Retrieved December 11, 2015 from https://www.broadband-forum.org/technical/download/TR-069.pdf.

Bluetooth. 2016. *Bluetooth Technology Website*. (2016). Retrieved January 28, 2016 from https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy.

Bormann, C., Ersue, M., and Keranen, A. 2014. *Terminology for Constrained-Node Networks*. IETF. (2014). http://www.rfc-editor.org/rfc/pdfrfc/rfc7228.txt.pdf.

Bosch Software Innovations. 2015. *The Bosch IoT Suite. Technology for a Connected World*. (2015). Retrieved December 3, 2015 from https://www.bosch-si.com/media/en/bosch_software_innovations/documents/brochure/products_2/bosch_iot_suite/brochure.pdf.

Comarch Technologies. 2015a. *Comarch IoT Platform. In the pursuit of becoming smart*. (2015). Retrieved November 20, 2015 from http://technologies.comarch.com/wp-content/uploads/2015/10/CT_IoT-white-paper_22092015_WEB.pdf.

Comarch Technologies. 2015b. *Digital Lifestyle & IoT Solutions*. (2015). Retrieved November 20, 2015 from http://www.comarch.com/files-com/file_91/Comarch-Digital-Lifestyle-and-IoT-Solution-283522.pdf.

Dell. 2016. *Dell IoT solutions*. (2016). Retrieved January 20, 2016 from http://www.dell.com/learn/us/en/04/oem/oem-internet-of-things.

Eclipse Foundation. 2015. *Kura Documentation - Introduction*. (2015). Retrieved January 21, 2016 from http://eclipse.github.io/kura/doc/intro.html.

Eclipse Foundation. 2016. *Kura - Open Source Framework for IoT*. (2016). Retrieved January 20, 2016 from http://www.eclipse.org/kura/.

Eloranta, V.-P., Koskinen, J., Leppänen, M., and Reijonen, V. 2014a. *Designing distributed control systems. A pattern language approach*. Wiley series in software design patterns. Wiley, Hoboken, NJ.

Eloranta, V.-P., Koskinen, J., Leppänen, M., and Reijonen, V. 2014b. Patterns for the Companion Website.

ETSI. 2015. *3GPP TS 23.682. Architecture enhancements to facilitate communications with packet data networks and applications*. (2015). Retrieved January 20, 2016 from http://www.etsi.org/deliver/etsi_ts/123600_123699/123682/12.04.00_60/ts_123682v120400p.pdf.

EVRYTHNG. 2015. *Evrythng Platform Overview*. (2015). Retrieved April 28, 2016 from https://evrythng.com/wp-content/uploads/EVRYTHNG-IoT-Platform-Overview.pdf.

EVRYTHNG. 2016. *THINGHUB Local Cloud Gateway*. (2016). Retrieved April 28, 2016 from https://evrythng.com/wp-content/uploads/THNGHUB-data-sheet.pdf.

Falkenthal, M., Barzen, J., Breitenbücher, U., Brügmann, S., Joos, D., Leymann, F., and Wurster, M. 2016. Pattern Research in the Digital Humanities: How Data Mining Techniques Support the Identification of Costume Patterns. In *Proceedings of the 10th Symposium and Summer School On Service-Oriented Computing (SummerSOC 2016)*. Springer.

Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., and Leymann, F. 2014a. Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains. *International Journal on Advances in Software 7*, 3&4, 710–726.

Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., and Leymann, F. 2014b. From Pattern Languages to Solution Implementations. In *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. IARIA, Wilmington, DE, 12–21.

Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., Leymann, F., Hadjakos, A., Hentschel, F., and Schulze, H. Leveraging Pattern Application via Pattern Refinement. In *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC 2015)*. (to appear).

Fehling, C., Barzen, J., Breitenbücher, U., and Leymann, F. 2015a. A Process for Pattern Identification, Authoring, and Application. In *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2721956.2721976.

Fehling, C., Barzen, J., Falkenthal, M., and Leymann, F. 2015b. PatternPedia - Collaborative Pattern Identification and Authoring. In *PURPLSOC (In Pursuit of Pattern Languages for Societal Change): The Workshop 2014*. epubli GmbH, Berlin, 252–284.

Fehling, C., Leymann, F., Retter, R., Schupeck, W., and Arbitter, P. 2014. *Cloud Computing Patterns. Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Wien.

Fowler, M. 2011. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ.

Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., and Stafford, R. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, Massachusetts.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

Google. 2016. *Remotely ring, lock, or erase a lost device - Accounts Help*. (2016). Retrieved January 27, 2016 from https://support.google.com/accounts/answer/6160500.

Harrison, N.B. 2006a. Advanced Pattern Writing. Patterns for Experienced Pattern Authors. In *Pattern languages of program design 5*. Addison-Wesley, Upper Saddler River, NJ, 433–452.

Harrison, N.B. 2006b. The Language of Shepherding. A Pattern Language for Shepherds and Sheep. In *Pattern languages of program design 5*. Addison-Wesley, Upper Saddler River, NJ, 507–530.

Hohpe, G., and Woolf, B. 2004. *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, Boston, Massachusetts.

IBM. 2015a. *Device Management Operations - Device Actions*. (2015). Retrieved December 14, 2015 from https://docs.internetofthings.ibmcloud.com/device_mgmt/operations/device_actions.html.

IBM. 2015b. *Getting started with IoT Real-Time Insights*. (2015). Retrieved December 14, 2015 from http://www.ng.bluemix.net/docs/services/iotrtinsights/index.html.

IETF. 2014. *The Constrained Application Protocol (CoAP)*. (2014). Retrieved January 20, 2016 from https://tools.ietf.org/html/rfc7252.

IFTTT. 2016. *IFTTT*. (2016). Retrieved August 15, 2016 from https://ifttt.com/.

Industrial Internet Consortium. 2015. *Overview*. (2015). Retrieved January 25, 2016 from http://www.iiconsortium.org/pdf/IIC-Overview-11-24-15.pdf.

Intel. 2016. *Intel IoT Gateways*. (2016). Retrieved January 20, 2016 from https://www-ssl.intel.com/content/www/us/en/embedded/solutions/iot-gateway/overview.html.

Kagemann, H., Wahlster, W., and Helbig, J. 2013. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0*. acatech. (2013). Retrieved January 25, 2016 from http://www.acatech.de/fileadmin/user_upload/Baumstruktur_nach_Website/Acatech/root/de/Material_fuer_Sonderseiten/Industrie_4.0/Final_report__Industrie_4.0_accessible.pdf.

Kii. 2015a. *Disable/Enable Things*. (2015). Retrieved December 18, 2015 from http://documentation.kii.com/en/guides/thingifsdk/thingsdk/thing-client/things-status/.

Kii. 2015b. *State Registration and Retrieval*. (2015). Retrieved December 17, 2015 from http://documentation.kii.com/en/starts/thingifsdk/model/states/.

Kopp, O., Falkenthal, M., Hartmann, N., Leymann, F., Schwarz, H., and Thomsen, J. 2015. Towards a Cloud-based Platform Architecture for a Decentralized Market Agent. In *INFORMATIK 2015*. Gesellschaft für Informatik e.V. (GI), Bonn, 69–80.

Le Gal, C., Martin, J., Lux, A., and Crowley, J.L. 2001. SmartOffice: Design of an Intelligent Environment. *IEEE Intelligent Systems 16*, 4, 60–66. DOI:http://dx.doi.org/10.1109/5254.941359.

Meszaros, G., and Doble, J. 1996. Metapatterns: A Pattern Language for Pattern Writing. In *Third Pattern Languages of Programming Conference*. Addison-Wesley.

Microsoft. 2015a. *Azure and IoT*. (2015). Retrieved November 10, 2015 from https://azure.microsoft.com/en-us/documentation/articles/iot-hub-what-is-azure-iot/.

Microsoft. 2015b. *Azure IoT Hub guidance*. (2015). Retrieved November 10, 2015 from https://azure.microsoft.com/en-us/documentation/articles/iot-hub-guidance/.

Microsoft. 2015c. *IoT device management using Azure IoT Suite and Azure IoT Hub*. (2015). Retrieved November 10, 2015 from https://azure.microsoft.com/en-us/documentation/articles/iot-hub-device-management/.

myDevices. 2015. *myDevices Connected Device Platform for the Internet of Things*. (2015). Retrieved November 9, 2015 from https://www.mydevices.com/platform.

Nam, T., and Pardo, T.A. 2011. Conceptualizing Smart City with Dimensions of Technology, People, and Institutions. In *Proceedings of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times*. ACM, New York, NY, 282–291. DOI:http://dx.doi.org/10.1145/2037556.2037602.

Nexcom. 2016. *IoT Gateway*. (2016). Retrieved January 20, 2016 from http://www.nexcom.com/Products/industrial-computing-solutions/iot-solutions/iot-gateway.

OASIS. 2014. *MQTT Version 3.1.1*. (2014). Retrieved January 28, 2016 from http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf.

Object Management Group. 2015. *Data Distribution Service (DDS)*. (2015). Retrieved August 9, 2016 from http://www.omg.org/spec/DDS/1.4/PDF/.

ONEM2M. 2015. *Functional Architecture*. (2015). Retrieved November 25, 2015 from http://www.onem2m.org/images/files/deliverables/TS-0001-Functional_Architecture-V1_6_1.pdf.

OPC Foundation. 2016. *Unified Architecture - OPC Foundation*. (2016). Retrieved January 28, 2016 from https://opcfoundation.org/about/opc-technologies/opc-ua/.

Open Interconnect Consortium. 2016. *Open Interconnect Consortium*. (2016). Retrieved January 28, 2016 from http://openinterconnect.org/.

Open Mobile Alliance. 2015a. *Lightweight M2M - Lock and Wipe Object (LwM2M Object - LockWipe)*. (2015). Retrieved December 4, 2015 from http://technical.openmobilealliance.org/Technical/Release_Program/docs/LWM2M_LOCKWIPE/V1_0-20150217-C/OMA-TS-LWM2M_LockWipe-V1_0-20150217-C.pd.

Open Mobile Alliance. 2015b. *OMA Device Management Protocol*. (2015). Retrieved January 28, 2016 from http://technical.openmobilealliance.org/Technical/Release_Program/docs/DM/V2_0-20150122-C/OMA-TS-DM_Protocol-V2_0-20150122-C.pd.

PawTrax. 2016. *Welcome to PawTrax*. (2016). Retrieved August 10, 2016 from http://www.pawtrax.co.uk/.

Qanbari, S., Pezeshki, S., Raisi, R., Mahdizadeh, S., Rahimzadeh, R., Behinaein, N., Mahmoudi, F., Ayoubzadeh, S., Fazlali, P., Roshani, K., Yaghini, A., Amiri, M., Farivarmoheb, A., Zamani, A., and Dustdar, S. 2016. IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications. In *Proceedings of the First International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 277–282. DOI:http://dx.doi.org/10.1109/IoTDI.2015.18.

Reiners, R., Falkenthal, M., Jugel, D., and Zimmermann, A. 2013. Requirements for a Collaborative Formulation Process of Evolutionary Patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Programs (EuroPlop)*. ACM, New York, NY. DOI:http://dx.doi.org/10.1145/2739011.2739027.

Röcker, C. 2010. Services and Applications for Smart Office Environments - A Survey of State-of-the-Art Usage Scenarios. *International Journal of Social, Behavioral, Educational, Economic, Business and Industrial Engineering 4*, 1, 51–67.

SmartThings. 2015. *Architecture*. (2015). Retrieved November 23, 2015 from http://docs.smartthings.com/en/latest/architecture/index.html.

Starsinic, M., Mohamed, A.S.I., Lu, G., Seed, D., Aghili, B., Wang, C., Palanisamy, S., and Murthy, P. 2015. An IP-Based Triggering Method for LTE MTC Devices. In *2015 Wireless Telecommunications Symposium (WTS)*. IEEE. DOI:http://dx.doi.org/10.1109/WTS.2015.7117251.

Stringify. 2016. *Home - Stringify*. (2016). Retrieved May 3, 2016 from https://www.stringify.com/.

Su, K., Li, J., and Fu, H. 2011. Smart City and the Applications. In *2011 International Conference on Electronics, Communications and Control (ICECC)*. IEEE, Piscataway, NJ, 1028–1031. DOI:http://dx.doi.org/10.1109/ICECC.2011.6066743.

Thread Group. 2016. *Home*. (2016). Retrieved January 28, 2016 from http://www.threadgroup.org/.

waylay.io. 2016. *Waylay.io Documentation*. (2016). Retrieved April 27, 2016 from http://docs.waylay.io/Tasks-and-Templates.html.

Wellhausen, T., and Fießer, A. 2012. How to write a pattern? A rough guide for first-time pattern authors. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*. ACM, New York, NY.

Wind River. 2015. *Wind River Helix Device Cloud*. (2015). Retrieved November 12, 2015 from http://www.windriver.com/products/product-overviews/wr-device-cloud_overview.pdf.

Wink. 2015. *Wink Hub*. (2015). Retrieved January 20, 2016 from http://www.wink.com/products/wink-hub/.

Zachariah, T., Klugman, N., Campbell, B., Adkins, J., Jackson, N., and Dutta, P. 2015. The Internet of Things Has a Gateway Problem. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications - HotMobile '15*. ACM, New York, NY, 27–32. DOI:http://dx.doi.org/10.1145/2699343.2699344.

Zapier. 2016. *Connect Your Apps and Automate Workflows*. (2016). Retrieved August 15, 2016 from https://zapier.com/.

ZigBee Alliance. 2016. *Control your World*. (2016). Retrieved January 28, 2016 from http://www.zigbee.org/.

Z-Wave Alliance. 2016. *The Internet of Things is powered by Z-Wave*. (2016). Retrieved January 28, 2016 from http://z-wavealliance.org/.