A

A Scalable Framework for Provisioning Large-scale IoT Deployments

Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar,
{lastname}@dsg.tuwien.ac.at, Distributed Systems Group, TU Wien

Internet of Things (IoT) devices are usually considered as external application dependencies that only provide data, or process and execute simple instructions. The recent emergence of IoT devices with embedded execution environments allows practitioners to deploy and execute custom application logic directly on the device. This approach fundamentally changes the overall process of designing, developing, deploying, and managing IoT systems. However, these devices exhibit significant differences in available execution environments, processing, and storage capabilities. To accommodate this diversity, a structured approach is needed to uniformly and transparently deploy application components onto a large number of heterogeneous devices. This is especially important in the context of large-scale IoT systems, such as in the smart city domain. In this paper, we present LEONORE, an infrastructure toolset that provides elastic provisioning of application components on resource-constrained and heterogeneous edge devices in large-scale IoT deployments. LEONORE supports push-based as well as pull-based deployments. To improve scalability and reduce generated network traffic between cloud and edge infrastructure, we present a distributed provisioning approach that deploys LEONORE local nodes within the deployment infrastructure close to the actual edge devices. We show that our solution is able to elastically provision large numbers of devices using a testbed based on a real-world industry scenario.

CCS Concepts: •**Computing methodologies** → **Distributed computing methodologies;** •**Computer systems organization** → **Distributed architectures;**

Additional Key Words and Phrases: IoT, framework, provisioning, large-scale, resource-constrained, gateway

ACM Reference Format:
Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar, 2015. A Scalable Framework for Provisioning Large-scale IoT Deployments *ACM Trans. Internet Technol.* V, N, Article A (January YYYY), 20 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Traditional approaches for developing and designing Internet of Things (IoT) applications, such as AWS IoT[1] and Bluemix IoT Solutions[2], are based on rigid layered architectures [Agrawal et al. 2010]. The bottom layer, consisting of deployed IoT devices and their communication facilities, is managed using a middleware layer that exposes the underlying hardware in a unified manner for consumption by a top-level application layer, which executes relevant business logic and visualizes processed sensor data [Li et al. 2014]. Such a layered architecture implies that business logic is only executed in the application layer, and IoT devices are assumed to be deployed with appropriate software and readily available [Da Xu et al. 2014]. However, in practice this is not the case. Currently, configuration and provisioning of IoT devices must largely be performed manually, making it difficult to quickly react to changes in application or

[1]http://aws.amazon.com/iot/
[2]https://www.ibm.com/cloud-computing/bluemix/solutions/iot/

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 ACM 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.
© YYYY ACM. 1533-5399/YYYY/01-ARTA $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

infrastructure requirements. Moreover, we see the emergence of IoT devices (e.g., Intel IoT gateway[3], SmartThings Hub[4], and Raspberry Pi[5]) that offer functionality beyond basic connected sensors and provide constrained execution environments with limited processing, storage, and memory resources to execute device firmware. These currently unused execution environments can be incorporated in IoT systems to offload parts of the business logic onto devices. In the context of our work, we refer to these devices as *IoT gateways*.

In large-scale IoT systems, such as in the smart city domain, leveraging the processing capabilities of gateways is especially promising, as their currently untapped processing capabilities can be used to improve dependability, resilience, and performance of IoT applications by moving parts of the business logic towards the edge of the infrastructure. Incorporating edge devices as first-class execution environments in the design of IoT applications allows them to dynamically adapt to inevitable changes, such as new requirements or adjustments in regulations, by modifying their component deployment topology and edge processing logic. System integrators can avoid infrastructure silos and vendor lock-in by implementing custom business logic to be executed on gateways, and even purchase and sell such application components in IoT application markets [Vögler et al. 2015a]. However, the heterogeneity of currently available IoT gateways poses challenges for application delivery due to significant differences in device capabilities (e.g., available storage and processing resources), as well as deployed and deployable software components. Furthermore, the large number of devices in typical IoT systems calls for a scalable and elastic provisioning solution that is specifically tailored to the resource-constrained nature of IoT devices.

In this paper, we present LEONORE, an infrastructure and toolset for provisioning application components on edge devices in large-scale IoT deployments. To accommodate the resource constraints of IoT gateways, installable application packages are fully prepared on the provisioning server and specifically catered to the device platform to be provisioned. Our solution allows for both, push- and pull-based provisioning of devices. Pull-based provisioning, a common approach in contemporary configuration management systems, allows devices to independently schedule provisioning runs at off-peak times, whereas push-based provisioning allows for greater control over the deployed application landscape by immediately initiating critical software updates or security fixes. We illustrate the feasibility of our solution using a testbed based on a real-world IoT deployment from one of our industry partners. We show that LEONORE is able to elastically provision large numbers of IoT gateways in reasonable time. By deploying application packages with significantly different sizes, we furthermore show that our distributed provisioning mechanism can successfully scale with the size of IoT deployments and can substantially reduce required network bandwidth between edge devices and the central provisioning component. A preliminary version of our approach has been presented in [Vögler et al. 2015c] where we introduce the basic concepts of the LEONORE framework, along with a centralized mechanism for provisioning edge devices. In this work, we extend the framework with a distributed provisioning approach to reduce network overhead, provide a detailed discussion of our prototype implementation, and significantly extend the evaluation of our framework.

The remainder of this paper is structured as follows: In Section 2 we motivate our work and outline the specific problems to be tackled. In Section 3 we introduce the LEONORE infrastructure and toolset to address the identified problems in deploying large-scale IoT systems, and present our distributed provisioning approach in Sec-

[3]https://www-ssl.intel.com/content/www/us/en/embedded/solutions/ iot-gateway/overview.html
[4]http://www.smartthings.com/
[5]https://www.raspberrypi.org/

tion 4. We provide detailed evaluations in Section 5, discuss relevant related research in Section 6, followed by a conclusion and an outlook on future research in Section 7.

2. MOTIVATION

One of the most demanding aspects in the smart city domain is the ability to connect and manage millions of heterogeneous devices, which are emerging from the IoT. The extremely fast-paced evolution within the IoT and the changing requirements in the smart city domain itself make this not only a matter of handling large-scale deployments, but more importantly about supporting the ability to manage this change. A specifically demanding area facing these specific challenges is large-scale Building Management and Operations (BMO). BMO providers not only need to be able to manage and stage large numbers of new devices, they also need to be able to react on rapidly changing requirements to their existing infrastructure. However, current solutions are mostly manual and only deal with fragments of a BMO providers's infrastructure, which leads to the incapability of dealing with the vast amount of devices and changing requirements in an efficient, reliable, and cost-effective way. BMOs dealing with large-scale IoT systems need to be able to handle two distinct stages. The first is the initial deployment and staging of devices, the second is the management of updates of varying frequency and priorities. To illustrate this, we consider the case of a BMO that manages several hundreds of buildings with a broad variety of tenants in a large city. These buildings are equipped with a huge amount of heterogeneous IoT devices including simple sensors to detect smoke and heat, elevator and door controls, as well as complex cooling and heating systems. To reliably operate this infrastructure, the BMO relies on physical gateways [Zhu et al. 2010], which provide constrained execution environments with limited processing, storage, and memory resources to execute the device firmware and simple routines. These gateways are usually installed once in a specific location in a building and then connected and integrated into an infrastructure solution to enable the basic bundling and management of a wide variety of connected devices. The current lack of standardization in this novel field combined with the current market situation leads to a significant heterogeneity in terms of software environments when it comes to these gateways. Initially, the gateways need to be staged with the necessary capabilities to ensure their basic functionality. They need to support the connected sensors, must run the latest firmware and have to be integrated into a specific deployment structure. This is followed by long term evolution requirements like changing deployments, shifting capabilities, as well as updating the software environment or firmware. A special case of updates are security updates and hot fixes that need to be deployed quickly to ensure that the infrastructure stays operational. Delays in these updates can expose severe security risks, which make them time critical. The increasing number of connected devices leads to an increased vulnerability to hacks and exploits, and in the IoT domain, where these devices are connected to the real world, this poses a major threat.

We therefore identify the following requirements in the context of this scenario: (1) A provisioning framework must consider that participating gateways are resource-constrained in terms of their processing, memory, and storage capabilities. (2) Scenarios dealing with large-scale deployments comprising thousands of gateways with a wide variety of different execution environments must be supported. (3) Requirements of deployed applications change over time, which makes updates necessary. These updates can either be non-time-critical or time-critical. (4) In order to sustain operations, updates need to be efficient and fast, and therefore have to be performed at runtime.

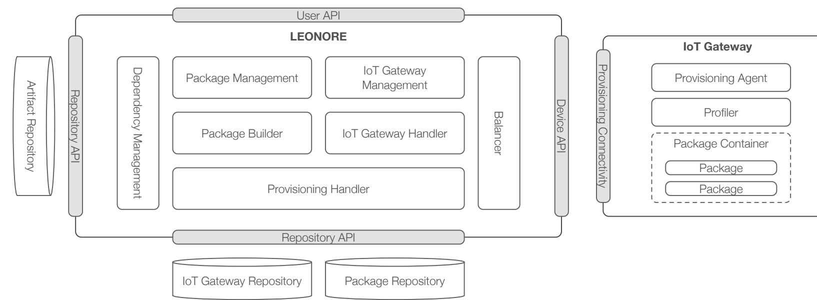


Fig. 1: LEONORE – Overview

3. APPROACH

In order to address the previously defined requirements, we present LEONORE, an infrastructure to provision application components on gateways in large-scale IoT deployments. The overall architecture of our approach is depicted in Figure 1 and consists of the following components: (i) Application Packages, (ii) IoT gateways, and (iii) LEONORE, the provisioning framework. In the following, we discuss these components in more detail.

3.1. Application Packages

Usually an application in the IoT domain consists of different application components and supporting files (e.g., libraries and binaries), which we refer to as artifacts. To enable automatic provisioning of these artifacts, LEONORE builds gateway-specific application packages, which are a compound of various artifacts and have the following structure. First, each package has an `id`, which uniquely identifies the package. Second, each package contains a `binary` folder, to store required artifacts. Furthermore, it also contains the resolved application dependencies to avoid expensive dependency resolution on the gateway. Finally, in the `control` folder all instructions for installing, uninstalling, starting and stopping this package are included. Additionally, a `path` file defines the installation paths and the order of installing/uninstalling artifacts. With this approach the heavy lifting is done by the framework, and gateways only have to unpack the package and execute the provided installation instructions, which usually just copy artifacts in place without any additional processing.

3.2. IoT Gateway

To efficiently provision edge devices, we first need a general and generic representation of such devices. We analyzed the capabilities of several gateways that are commonly applied by our industry partners in the domain of Building Management Systems. Our findings show that in general such gateways have limited hardware components and use some rudimentary, tailored operating system (e.g., a BusyBox[6] user land on a stripped down Linux distribution). Installing or updating software components is a tedious manual task, since there are no supporting packaging or updating tools in place, as known from full-featured operating system distributions[7]. Furthermore, due to limited resources in terms of disk space, adding new capabilities often requires the removal of already installed components. Taking all these limitations into account, we derived the final representation of a gateway for our approach as depicted on the right-hand side in Figure 1. The IoT gateway has the following components: (i) a *container*,

[6]http://www.busybox.net
[7]e.g., apt (https://packages.qa.debian.org/a/apt.html) or rpm (http://www.rpm.org/)

hosting application packages, (ii) a *profiler*, monitoring the current status of the gateway, (iii) an *agent*, communicating with the provisioning framework, and (iv) a *connectivity layer*, supporting different communication protocols and provisioning strategies.

3.2.1. Profiler. As mentioned above, gateways are usually resource-constrained, which means that they only provide limited disk space, memory and processing power. Therefore, keeping track of these resources is of utmost importance. In order to do that the profiler uses pre-defined interfaces to constantly monitor the underlying system (e.g., static information like ID, MAC-address, and instruction set, or dynamic information like disk- and memory-consumption). The profiler sends the collected information either periodically or on request to the provisioning framework. Based on this heartbeat information the provisioning framework can detect failures (e.g., gateway has a malfunction), which allows notifying the operator. Once the framework receives the heartbeat again, the gateway is considered back and running.

3.2.2. Application Package and Container. All packages that are not pre-installed on the IoT gateway have to be provisioned by the framework at runtime. Therefore, the IoT gateway uses a runtime container to store and run application packages. By using a separate container we ensure that installing or removing packages does not interfere with the underlying system, and avoids expensive reboots or configuration procedures.

3.2.3. Provisioning Agent. An essential part of the overall provisioning framework is the provisioning agent. The pre-installed agent runs on each IoT gateway and manages application packages that are locally hosted and stored. The management tasks of the agent comprise installing, uninstalling, starting, and stopping packages. Furthermore, the agent is responsible for handling requests from the framework and triggers the respective actions on the IoT gateways (e.g., gather latest information via the profiler or trigger the provisioning of an application package).

3.2.4. Connectivity Layer. Since gateways usually use different software communication protocols in large real world deployments (e.g., oBIX[8] or CoAP[9]), our approach provides a pluggable connectivity layer. This layer can either reuse the deployed software communication protocols or extend services provided by the underlying operating system. Additionally, this layer provides extensible strategies to provision the gateway. In the current implementation, we provide two strategies: (i) a *pull-based* approach where the provisioning agent queries the framework for provisioning tasks, and (ii) a *push-based* approach where the framework pushes new updates to the gateway and the agent triggers the local provisioning.

3.3. The LEONORE Provisioning Framework

The enabling framework to provision edge devices in large-scale deployments is depicted on the left-hand side in Figure 1. LEONORE is a cloud-based framework and the overall design follows the microservice architecture [Newman 2015]. This approach enables building scalable, flexible, and evolvable applications. Especially the flexible management and scaling of components is important for LEONORE when dealing with large-scale deployments. In the following, we introduce the main components of LEONORE and discuss the balancer-based scaling approach.

3.3.1. Repositories. To manage all relevant information for LEONORE, the framework relies on a number of repositories:

[8]http://www.obix.org
[9]http://coap.technology

Artifact repository. Usually, an application consists of multiple artifacts that are linked together to fulfill specific requirements. To handle these artifacts and make them reusable, a repository is used. The repository manages artifacts by storing source code, pre-built binaries, dependencies, possible configurations, and further necessary information that is required for the application building process. Furthermore, the repository provides a mechanism to store different versions of an artifact.

IoT gateway repository. This repository stores relevant gateway specific information that is needed for creating the deployable application package. This information includes: hardware configuration (e.g., disk space, memory, processor), software (e.g., kernel version, installed components/tools), as well as supported provisioning strategies and communication protocols. Additionally, for each IoT gateway the repository stores the provisioned application packages, which is important in case a different version of an installed package needs to be provisioned, since this might require the removal of an already installed version.

Package repository. Application packages specifically built for a set of IoT gateways are stored in the package repository. This approach guarantees that packages are only built once, and all compatible gateways are provisioned with the same package. Furthermore, by storing the packages in a repository it is easier to scale the framework, since no data is stored in memory and therefore components can be easily replicated. After IoT gateways are successfully provisioned, the package is removed after a configurable amount of time to avoid storing unnecessary data.

3.3.2. Package Management. To provision application packages with LEONORE, users have to add artifacts via the package management component. This component is responsible for retrieving all necessary information (e.g., name and version), required binaries, available source files, configurations, policies, and dependencies on other artifacts, from the user. After the user has provided this information along with the artifacts, the package management stores them in the respective repository. The structure of the repository follows the layout of conventional software package management systems (e.g., Maven[10]).

3.3.3. Dependency Management. Since many applications depend on libraries or other applications, LEONORE utilizes the following mechanism to resolve these application dependencies. The data model for the dependency management consists of artifacts, releases, and dependencies between these releases. Each artifact has a set of releases, and each release has a set of dependencies to other artifacts. Thus, the releases and dependencies create a well-structured directed graph where releases are nodes and dependencies are directed edges. This model allows us to reuse well-known graph algorithms (i.e., depth-first search) to find all dependencies for a specific release. Therefore, according to the desired artifact, the dependency management finds a list of suitable artifacts and provides a plan that can be used to build the actual application package. The plan includes a dependency tree and all needed artifacts. The dependencies are represented as a directed graph, with nodes representing artifacts like applications, libraries, operating system tools, and hardware components, whereas edges represent dependencies between nodes. As an example, let us consider a Java application, where the application code is packaged as a jar file. In order to execute this application, it has a dependency on the JVM 1.8 for ARM runtime.

3.3.4. Package Builder. To create the actual application package that can be provisioned, the package builder is used. In order to build an application package, the

[10]http://maven.apache.org

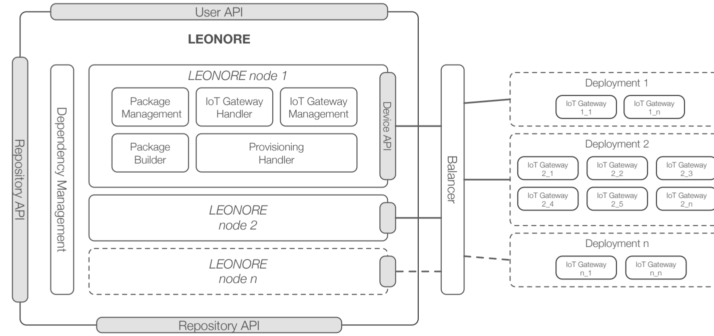


Fig. 2: LEONORE – Balancer

builder performs the following steps: (i) retrieve gateway-specific information from the IoT gateway management, (ii) gather a list of suitable plans using the dependency management, (iii) build an application package based on the plan, (iv) notify the provisioning handler to trigger the actual provisioning if the build was successful, (v) try next plan in list if the build failed, (vi) store application package in package repository.

3.3.5. IoT Gateway Management and IoT Gateway Handler. In order to deal with the bootstrapping problem, i.e., to know which IoT gateways are available for provisioning, LEONORE follows the following approach. When an IoT gateway starts for the first time, the local provisioning agent registers the gateway with the framework by providing its unique identifier (e.g., derived from name, ID and mac-address) and the gathered profile data. Based on this information the IoT gateway management creates an entry in the IoT gateway repository and stores the provided information. The registration process is finalized by negotiating the supported provisioning strategy and communication protocol. This is possible, since each IoT gateway is pre-configured and provides some already installed communication protocols and provisioning strategies. Next, a suitable IoT gateway handler is assigned to this gateway. The handler is responsible for handling any further communication with the gateways, by providing the required communication protocols and provisioning strategies. IoT gateways that use the same protocols and strategies are grouped together and managed by a designated IoT gateway handler. This assures more flexibility and avoids mediating between protocols. Once the registration process is successful, the IoT gateway can be provisioned via the framework.

3.3.6. Provisioning Handler. To provision application packages, the provisioning handler first chooses a suitable provisioning strategy according to the information provided by the IoT gateway management. Then the handler checks if the respective package is already present in the package repository. If it is available it will be used, if not the handler triggers the building of gateway-specific application packages by invoking the package builder. Then, the provisioning handler executes the provisioning strategy. This means that the IoT gateway can either query the framework for application packages or the handler delegates the provisioning request to the respective IoT gateway handler, which pushes the update to the gateway and triggers the provisioning.

3.3.7. Balancer. Since LEONORE needs to provision large-scale deployments of IoT gateways, scalability is essential. Therefore, we provide several strategies to deal with the immense workload. First, the framework's design follows the microservice architecture principle. Thus, optimizing single components is relatively easy by moving them from one host to a more powerful host. Additionally, it is possible to scale components

by replicating them and therefore distributing the workload across multiple computing resources. Following this approach, components of LEONORE are classified in scalable and not scalable. Components that should be scaleable are grouped together in so-called LEONORE nodes. These nodes comprise all components that are required to handle and provision IoT gateways. The classification in scalable and not scalable is flexible and can be adapted depending on the requirements. Now that LEONORE provides the ability to replicate components via the notion of nodes, we further need a component that is responsible for creating and destroying these nodes, as well as distributing incoming requests to them. To this end, we introduce a balancer. In general, a balancer aims to optimize resource usage, to minimize response time and to maximize throughput. Figure 2 depicts how LEONORE scales up with a growing number of deployments by using the balancer. In Figure 2 we see that the balancer receives incoming requests from IoT gateways deployed in different areas. Based on a pluggable strategy the balancer gathers a suitable node from the pool of available LEONORE nodes and assigns the gateway to this node. The node is then responsible for handling any further interaction with the respective IoT gateway. LEONORE nodes are deployed using a $N + 1$ strategy with one active node and one hot standby initially. As load increases above the capacity of one node, the framework will immediately start to use the standby node for handling device provisioning requests, and furthermore start another LENORE node to again maintain a hot standby node. Currently, LEONORE scales nodes based on the number of gateways to be provisioned. In the future, we will provide additional strategies, such as a location-aware strategy that aims at deploying nodes close to affected IoT gateways to reduce network overhead.

3.4. Provisioning of Application Packages

Whenever an artifact is requested for a certain deployment, LEONORE performs the following steps: (1) check if the requested artifact is available; (2) resolve the given deployment to retrieve the set of IoT gateways that need to provisioned; (3) find the responsible LEONORE nodes, group the gateways according to their node assignment and delegate the provisioning task to the node; (4) on each node: analyze if the requested artifact is compatible with every IoT gateway, and group gateways that require the same application package (e.g., equal hardware or installed packages); (5) on each node: for each group of IoT gateways resolve dependencies and create application package; (6) on each node: execute required provisioning strategy for each IoT gateway; (7) on each node: wait until IoT gateways successfully provisioned the package to complete the provisioning task; (8) check if all nodes have completed their provisioning task to finalize the overall provisioning.

4. DISTRIBUTED PROVISIONING

After presenting the overall approach and the realization in the previous section, we now want to discuss certain limitations of our approach and propose an optimization addressing these shortcomings. In the approach presented so far, we assumed that the communication between the cloud and edge infrastructure is always available, reliable, and cheap. However, real world deployments use wireless communication links like 3G or GPRS that are not only slow and unreliable [Shrestha and Jasperneite 2012], but also expensive as they are usually charged based on transferred data. Additionally, the current approach puts the server-side framework under heavy load, which we already partly addressed by introducing a scalable LEONORE node concept. Nevertheless, by scaling LEONORE across several nodes and therefore provisioning more resources in the cloud, operating expenses increase along with the additional overhead of managing the provisioning and releasing of these nodes. In order to tackle these limitations, we apply the core notion of offloading business logic to the infrastructure edge

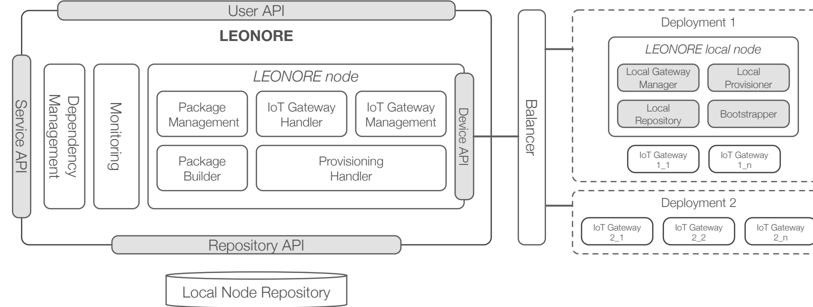


Fig. 3: LEONORE – Local Provisioner

to LEONORE itself, moving parts of the provisioning logic to suitable gateways in the field.

4.1. Server-side Extensions

To allow for the new concept of LEONORE local nodes, we extend LEONORE by adding several new components, which we describe in the following and are depicted on the left-hand side in Figure 3.

4.1.1. Monitoring. We introduce a central (i.e., not replicated) monitoring component that collects the following information of the overall framework: (i) The number of provisioned packages. (ii) The consumed bandwidth based on the number of provisioned packages and incoming pulling requests. (iii) The overall time that is needed for provisioning the edge infrastructure. (iv) Information about the provisioned gateways, e.g., used disk space, memory and processing power. (v) For each deployed LEONORE node, relevant metrics of the node (e.g., average response time, uptime, and load average). Based on this monitoring data, collected by LEONORE, it is possible to decide that a LEONORE local node needs to be provisioned and where in the edge infrastructure it is feasible to do so.

4.1.2. Service API. Since in the current version LEONORE does not automatically decide when and where to provision a LEONORE local node, we created a service API that allows operators to retrieve the collected data from the monitoring component. Additionally, operators query for gateways that are suitable for hosting a LEONORE local node. Finally, the service API allows operators to trigger the provisioning and releasing of LEONORE local nodes in the edge infrastructure.

4.1.3. Local Node Repository. To keep track of provisioned LEONORE local nodes, we added a separate repository. For each LEONORE local node deployment, we store in this repository the ID of the node and the gateway in the edge infrastructure that is provisioned with the respective node. The combination of node ID, gateway ID, and gateway IP uniquely identifies the node deployment. Furthermore, the repository stores the current status of a LEONORE local node using the monitoring component.

4.2. LEONORE Local Node

In essence, the LEONORE local node provides the same capabilities as the server-side LEONORE node, but is specifically catered to be more lightweight in terms of memory consumption and CPU usage. This approach allows to execute LEONORE local nodes on gateways residing in the edge infrastructure, which only provide a fraction of the processing power of cloud resources. The architecture of the LEONORE local node

is depicted on the right-hand side in Figure 3. In the following, we outline the basic components of a LEONORE local node.

4.2.1. Local Gateway Handler. Like the gateway handler on the server-side LEONORE node, this component manages a local cluster of gateways. In order to know which gateways need to be handled, how to communicate with them, and what kind of provisioning strategy needs to be used, LEONORE provides this information when provisioning a LEONORE local node. Compared to the server-side manager, this approach is not as flexible, but since the local gateway handler is specifically built for this local cluster of gateways, we keep the overall footprint of the node small by avoiding resource expensive protocol mediation and bootstrapping of gateways.

4.2.2. Local Repository. In order to save bandwidth, the application package that needs to provisioned is not transferred to each gateway, but only sent to the LEONORE local node, which then takes care of provisioning the respective gateways. The node stores the package in a local cache repository using available RAM and/or disk resources if available. This allows for fast read and write access, while explicitly considering the underlying resources of the gateway. After successful provisioning, the cache is cleared to save memory on the gateway.

4.2.3. Local Provisioner. In contrast to the provisioner on the server-side LEONORE node, the local provisioner is more lightweight, since it does not have to deal with the process of building application packages, but only uses the already transferred application package to provision the gateways. Furthermore, the local provisioner provides an optimized local provisioning strategy, which solely supports the push mechanism, since it consumes less resources and puts the node under less load compared to polling in short intervals.

4.2.4. Bootstrapper. Once a LEONORE local node gets provisioned, the bootstrapper component of the local node takes care of the following two tasks. First, it registers the respective local node at LEONORE, which guarantees that the framework is aware of all deployed local nodes. Second, once the registration was successful and the framework accepted the registration request, the bootstrapper gathers health and status information about the local node. This information is then periodically published at a configurable interval, which is then collected by LEONORE.

4.3. LEONORE Local Node Deployment

In order to deploy a LEONORE local node in the edge infrastructure, the operator uses the previously described LEONORE service API to retrieve a list of suitable gateways that are capable to run a local node. Next, the operator chooses how to distribute the local nodes across the edge infrastructure. Since the distribution of nodes can depend on various factors, such as available connectivity or logical location, LEONORE provides a pluggable distribution mechanism that can be easily extended. Following our microservice architecture, this can be done by adding an additional microservice to the framework that specifically implements a new distribution approach. In the current implementation, we form clusters of gateways based on physical proximity (e.g., all gateways that are residing on the same floor). Based on these clusters, the distribution mechanism elects a suitable gateway to host the LEONORE local node. Next, after selecting the gateways that will host the local nodes, LEONORE provisions them using the same approach we use for ordinary artifacts. This means that the local node artifacts, which are already residing in the artifact repository, get bundled to an application package and then transferred to the gateway. On the gateway the application package is installed and started by the provisioning agent. Finally, after the startup of

the local node the bootstrapper registers the local node at LEONORE. After registration, the LEONORE local node is ready for serving provisioning requests.

4.4. Application Provisioning with LEONORE Local Nodes

As described in Section 3.4, when an artifact is requested for a specific deployment, LEONORE checks if the artifact is available, finds the responsible LEONORE node according to the retrieved set of gateways and delegates the provisioning to the respective server-side node. On the server side, the gateways are grouped based on available capabilities and application packages are created. Additionally, each server-side node now clusters gateways based on their physical proximity. For each cluster, the server-side node queries the local node repository for an available LEONORE local node. If no local node is present, the server-side node follows the original approach described in Section 3.4 and executes the required provisioning strategy for each gateway. However, if a local node is available, it transfers the application package and the directive to provision the cluster of gateways to this local node. The local node then takes care of provisioning these gateways by executing the optimized push-based approach.

5. EVALUATION

To evaluate our provisioning framework we created a test setup in the cloud using CoreOS to virtualize devices as Docker containers. IoT gateways in our experiments use two types of provisioning strategies – a pull and a push based approach.

When an IoT gateway uses the pull approach, the gateway's agent polls the provisioning framework for new tasks in a configurable interval (e.g., every second). The framework only provides new provisioning tasks for the IoT gateway, which collects and executes these tasks. With short polling intervals, this approach generates increased load on the framework, consumes more bandwidth, and uses more resources on the IoT gateways, but is more fault-tolerant in case of connectivity problems due to inherently frequent retries. For the push-based approach, the IoT gateway's agent only registers the gateway once at the framework and then remains idle until the framework pushes an update. When the agent gets pushed by the framework, it collects the provisioning task, executes it and returns to the idle state. In general, the push-based approach generates less load on both the IoT gateway and framework, but is more vulnerable to connectivity problems and operators need to take care to not inadvertently disrupt gateway operations by placing additional load on it.

To simulate real-world provisioning requests, we use the following two application packages. The first package uses the Sedona Virtual Machine[11] (SVM). SVM is written in ANSI C and is highly portable by design. It allows to execute applications written in the Sedona programming language and is optimized to run on platforms with less than 100KB of memory. For our experiments we developed a small sample application and used SVM Version 1.2.28. The final application package created by LEONORE has approximately 120KB – including the application code (.sab, .sax, .scode and Kits-file) and the required SVM binary. As second package we use Java 8 for ARM[12] (JVM). In general, using Java on an embedded device is a challenging task, since the JVM binary is quite big and often does not fit due to limited disk space. However, for our experiments we created a compact[13] Java package specifically for our gateway. Additionally, we developed a small sample application that pushes temperature readings to a web server. In total, the JVM application package created by LEONORE has approximately 12MB – including the application code (compiled .class files), JVM binary,

[11] http://www.sedonadev.org
[12] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-arm-downloads-2187472.html
[13] http://docs.oracle.com/javase/8/embedded/develop-apps-platforms/jrecreate.htm

and libraries. In the remainder of this section we give an overview of the used cloud setup, present four scenarios, and analyze the gathered results.

5.1. Setup

To see how LEONORE deals with large-scale deployments, we created an IoT testbed in our private OpenStack[14] cloud. In order to simulate large-scale deployments, we first created a snapshot of a real-world gateway that is used by our industry partner. Based on this snapshot, we created an image that can be run in Docker[15]. The running image (Docker container) is then used to virtualize and mimic the physical gateway in our cloud. Since for our evaluation we want to use several thousand virtualized gateways, we employed CoreOS[16] clusters. In general, CoreOS is a lightweight Linux distribution designed for security, consistency, and reliability. Instead of installing packages via a package management system like apt, CoreOS uses Docker to manage services at a higher level of abstraction. The service code and all dependencies are packaged within a container that can be run on one or many CoreOS machines. Containers provide benefits similar to full-blown virtual machines, but focus on applications instead of entire virtualized hosts. Since containers use the Linux kernel of the host, they have very little performance overhead, reducing the amount of required compute resources compared to VMs. CoreOS also provides fleet[17], a distributed init system that allows to treat a CoreOS cluster as if it is a single shared init system. We used fleet's notion of service units to dynamically generate according fleet unit files and use fleet for the automated deployment of virtualized gateways.

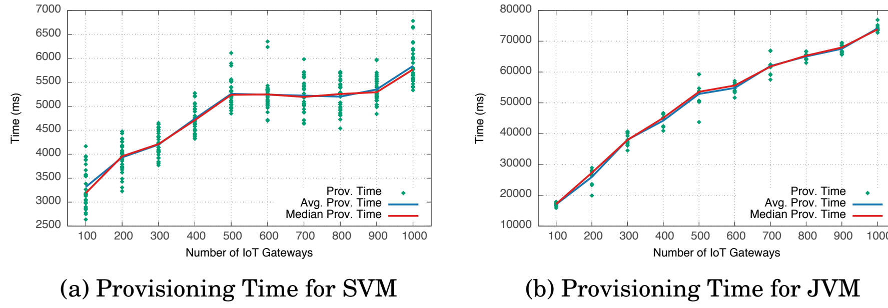
For our experiments we used the following setup: a CoreOS cluster of 8-16 virtual machines (depending on the scenario), where each VM is based on CoreOS 647.0.0 and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB Disk space). Our gateway-specific framework components are pre-installed in the containers. The LEONORE framework is initially distributed over 2 VMs using Ubuntu 14.04. The first VM hosts the balancer and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB Disk space). In order to represent a LEONORE node we created a reusable snapshot of a VM hosting all necessary LEONORE framework components and repositories. For the initial deployment of LEONORE two instances of this snapshot are started at the beginning of the experiment. However, only of of them is initially used by the framework, whereas the other acts as standby node. During the experiments LEONORE, more precisely the balancer, spins up this additional standby node to distribute the load created by the gateways. The VMs hosting the LEONORE nodes use the m2.medium flavor (5760MB Ram, 3 VCPUs and 40GB Disk space).

In the following scenarios we measured the overall execution time needed for provisioning an increasing number of devices. The provisioning time includes analyzing desired gateways, building gateway-specific application packages, transferring the packages to the gateways, installing the packages on the gateway, and executing them.

5.2. Scenario 1: 100 - 1000 IoT Gateways

For the first experiments we picked a scenario with 1000 virtual gateways. The scale of this scenario corresponds to a medium building management system, containing several big buildings (each with more than 10 floors). The 1000 virtual gateways are distributed among a CoreOS cluster consisting of 8 machines, where each machine hosts 125 containers. To demonstrate the scalability of our framework we show how

[14] http://www.openstack.org
[15] https://www.docker.com
[16] https://coreos.com
[17] https://github.com/coreos/fleet



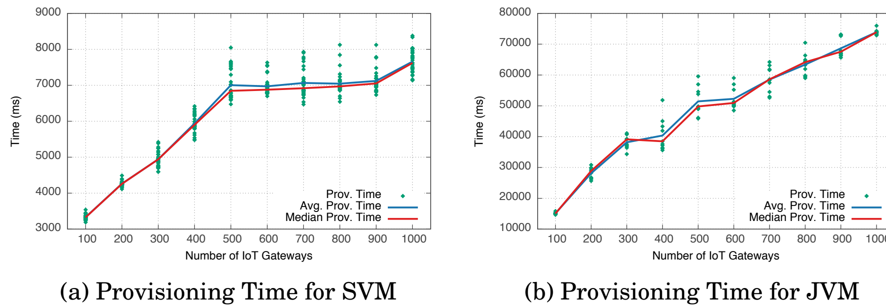
(a) Provisioning Time for SVM

(b) Provisioning Time for JVM

Fig. 4: Provisioning Time – Pull Strategy

our approach behaves with increasing load (number of gateways). For this scenario the balancer uses a scaling strategy that spins up another standby node when reaching 500 IoT gateways.

Figure 4 shows the overall execution time of the provisioning process for different deployments using the pull-based (gateways poll the framework every second) approach. In Figure 4a we show the execution time for provisioning the SVM application package. We see that the execution time increases almost linear until reaching 300 IoT gateways and then shows a sharper increase up to 500. When reaching 500 gateways, the balancer spins up another standby node and evenly schedules requests to the two active nodes. Therefore, provisioning time slightly decreases and at approximately 600 becomes constant. When reaching 900 IoT gateways, the provisioning time starts to rise again, which means that at this point both LEONORE nodes are fully loaded. In order to investigate possible outliers during the evaluation, we created a scatter plot, which is also depicted in Figure 4a. Since the SVM application is quite small and the polling interval of one second has a strong impact on the overall execution time, we executed each experiment 30 times. In the scatter plot we notice that at 600 IoT gateways we have some executions that finished more slowly, which is caused by the high network load and small polling interval. In general, the deviation of provisioning times is small. This shows that provisioning using the polling strategy is stable and provides reliable results. Figure 4b shows the execution time when provisioning the JVM application package. We clearly see that due to the increased size of the package the provisioning takes noticeably longer than for the SVM package. When the deployment reaches 500 IoT gateways, the balancer kicks in, which leads to a slight increase. In general, we notice that the provisioning of the JVM package scales linearly and produces almost no outliers, as one can see in the scatter plot in Figure 4b. Since this application package is quite big and therefore the provisioning time also increases significantly, the overhead of the polling approach is not noticeable. Figure 5 shows the overall execution time of the provisioning process for different deployments using the push-based (framework pushes provisioning tasks to IoT gateways) approach. In Figure 5a we see the overall execution time for provisioning the SVM application package. We notice a sharp increase up to 500 IoT gateways, which is due to the framework pushing requests to all gateways at once and therefore leads to a high load on both the IoT gateways and the framework. Once the balancer spins up another standby node, the execution time is almost constant, because the load is evenly distributed. When the deployment size reaches 900 IoT gateways, the execution time starts to rise again, which indicates that at this scale both nodes are fully loaded. The corresponding scatter plot is also depicted in Figure 5a, which reveals that there is only a very small deviation among the data points. Figure 5b depicts the provisioning time when using



(a) Provisioning Time for SVM

(b) Provisioning Time for JVM

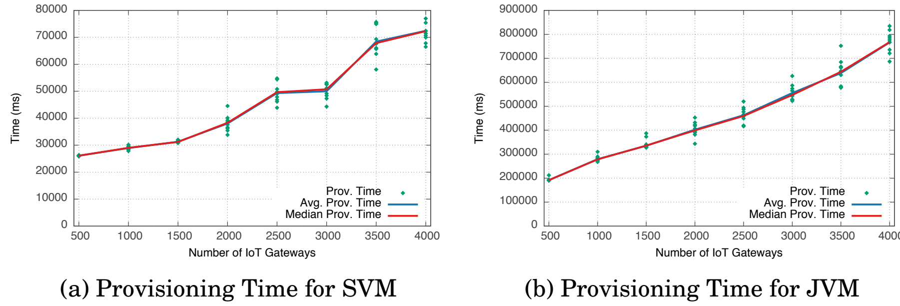
Fig. 5: Provisioning Time – Push Strategy

the JVM application package. Taking the results of the polling approach into account, we notice that the initial execution times are identical. However, at 300 IoT gateways we see that the initial overhead of the pushing approach is compensated and therefore the execution time decreases a little bit. From 400 to 500 IoT gateways, the node reaches maximal load. After the deployment size reaches 500, the balancer schedules the load evenly on two LEONORE nodes. The corresponding scatter plot, depicted in Figure 5b, shows that the deviation of data points is very small and the execution time increases linearly. After comparing both approaches, we see that our framework scales almost linearly and that for smaller application packages the pull-based approach is faster. For bigger packages both approaches put the framework under heavy load, but produced similar results.

5.3. Scenario 2: 500 - 4000 IoT Gateways

For the second experiment we used a scenario with 4000 virtual gateways, which corresponds to a large building management system containing dozens of big buildings (each with more than 10 floors). The 4000 virtual gateways are distributed among two CoreOS clusters, each consisting of 8 machines, where each machine hosts 250 containers. With this scenario we investigate how our framework scales when dealing with a large-scale deployment by using a scaling strategy that spins up another standby node when reaching 2500 IoT gateways.

Figure 6 shows the overall execution time of the provisioning process for different numbers of gateways using the push-based approach. In Figure 6a we notice that due to the deployment scale the overall execution time for provisioning the SVM application package got slower compared to the first scenario. This is expected since for this scenario we doubled the amount of CoreOS hosts and deployed twice as many containers on each CoreOS machine. This increase, in both the hosts and containers, generates a lot of traffic for the underlying network infrastructure of our cloud, which causes slower response times and therefore the overall provisioning takes longer. Furthermore, for this scenario we configured the balancer to handle 2500 IoT gateways per LEONORE node. We clearly see that up to 2500 IoT gateways, the execution time increases almost linearly. At 2500 the balancer schedules the requests evenly to two nodes, which causes a constant execution time. When reaching 3000 deployments, the execution time rises again, but once more starts to flatten at 4000. When looking at the scatter plot depicted in Figure 6a we see that at the beginning of the experiments the deviation among data points is very small and gets bigger with increasing number of IoT gateways. Figure 6b depicts the provisioning time when using the JVM application package. Compared to the first scenario, we also notice that the overall execution time got slower. However, now we see that the execution time increases linearly throughout



(a) Provisioning Time for SVM (b) Provisioning Time for JVM

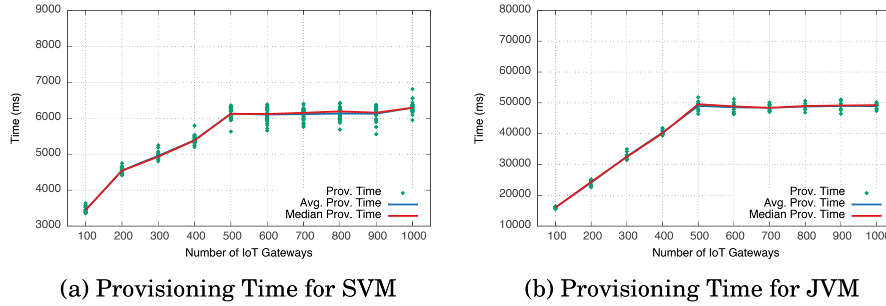
Fig. 6: Provisioning Time – Push Strategy, Large Deployment

the complete experiment, which provides stable provisioning results. This fact is also supported by the scatter plot that shows only small deviations among the results. Taking both experiments into account, we clearly see that our framework deals well with this rather large scenario and provides almost linear scale.

Distributed Provisioning using LEONORE Local Nodes. In order to evaluate the LEONORE local node optimization, we reused the setup discussed in Section 5.1. However, for this evaluation we spawn additional IoT gateways for hosting the LEONORE local nodes. For the first scenario, where we evaluate the provisioning time by using up to 1000 virtual gateways, we start 8 additional gateways. These 8 additional gateways, are distributed across the IoT testbed, so that on each CoreOS-Host one of these additional gateways is running. For the second scenario, where we provision up to 4000 virtual gateways, we use 16 additional gateways (i.e., one on each CoreOS Host). After distributing these additional gateways, we then provisioned them to run the LEONORE local node. Once these gateways got provisioned, their sole purpose is to host the local nodes and they are not changed throughout the evaluation. We decided to pre-provision the local nodes to conduct experiments that follow the same provisioning process as used in the evaluation above. Additionally, we argue that the provisioning and distribution of LEONORE local nodes will only happen sporadically and therefore this additional time should not contribute to the overall provisioning time.

5.4. Scenario 3: 100 - 1000 IoT Gateways

For the first experiments we picked a scenario with 1000 virtual gateways, which corresponds to a medium building management system. The 1000 virtual gateways are distributed among a CoreOS cluster consisting of 8 machines, where each machine hosts 125 containers and an additional container hosting the LEONORE local node. To demonstrate the scalability of our framework we show how our approach behaves with increasing load (number of gateways). Figure 7 shows the overall execution times of the provisioning process for different deployments by using the push-based approach (framework pushes provisioning tasks to IoT gateways), with LEONORE local nodes in place. In Figure 7a we see the overall execution time for provisioning the SVM application package. Compared to the basic provisioning approach (Figure 5a), we notice an initial steeper increase of the provisioning time for the LEONORE local node approach. This initial overhead is expected, since the local nodes deployed on the gateway are not as powerful as the server-side nodes and therefore need considerably longer for serving the gateways. However, after 500 the initial overhead is compensated and the local node provisioning provides faster results. We see that the provisioning time for the local node stays stable after 500 gateways and results in better overall provisioning times, compared to the original approach. This fact can also be seen in the included



(a) Provisioning Time for SVM (b) Provisioning Time for JVM

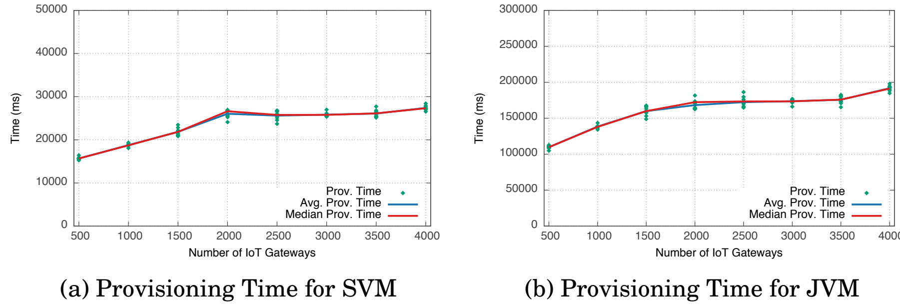
Fig. 7: Provisioning Time – LEONORE local node

scatter plot. The overall improvement can be explained by the fact that the overall load on the framework is more evenly distributed among the local nodes and therefore provides faster provisioning times. We see that the provisioning time improves on average by 10% using the SVM package with the LEONORE local node approach. Figure 7b depicts the provisioning time when using the JVM application package. We notice that initially the execution times of both the original (Figure 5b) and local node provisioning approach are identical. However, after reaching 200 gateways we see that the optimized approach starts to outperform the server-based provisioning. The improvement becomes more significant after reaching 500 gateways, since from that scale onwards the provisioning using LEONORE local nodes provides a constant and good execution time, compared to the ever increasing server-side provisioning. This effect is expected, since each local provisioning node only has to handle a fraction of the total gateway deployment. Additionally, considering the fact that the JVM package is quite big, the local node provisioning approach generates significantly less cloud to edge communication, since the application package gets only sent once to each local node and further provisioning happens only within the edge infrastructure. These facts lead to an average improvement of 17% when provisioning the JVM application package. Furthermore, in general the local node approach creates more stable results, which can be seen in the scatter plot depicted in Figure 7b.

5.5. Scenario 4: 500 - 4000 IoT Gateways

For the second experiment we used a scenario with 4000 virtual gateways, which corresponds to a large building management system containing dozens of big buildings (each with more than 10 floors). The 4000 virtual gateways are distributed among two CoreOS clusters, each consisting of 8 machines, where each machine hosts 250 containers and an additional container hosting the LEONORE local node. With this scenario we want to see how our framework scales when dealing with a large-scale deployment.

Figure 8 shows the overall execution times of the local node provisioning process for different numbers of deployments by using the push-based approach and the SVM respectively the JVM application package. Once again we compare the provisioning times with (Figure 8) and without (Figure 6) local nodes in place. In Figure 8a we notice that the execution time increases almost linearly when using the SVM application package. However, due to the scale of the scenario we see that the server-side nodes need to handle a lot of load, compared to the distributed local node setup, which explains why the local node setup provides better provisioning times from the beginning. When reaching 2000 IoT gateways, the execution time using the local node provisioning stays constant and does not increase anymore. Here, the LEONORE local node approach generates only a fraction of the load on the server-side framework, com-



(a) Provisioning Time for SVM (b) Provisioning Time for JVM

Fig. 8: Provisioning Time – Large Deployment, LEONORE local node

pared to the original approach, which causes the drastic improvement in execution time. On average, the distributed work model and reduced bandwidth consumption of LEONORE local nodes, improves the provisioning time by 49%. In Figure 8b we notice that the local node provisioning approach for the JVM package also provides significantly better results than the server side approach. Since the JVM package is quite big, the distributed work model performs even better and therefore improves the overall provisioning time on average by 64%. Furthermore, by looking at the included scatter plots in both figures we see that the optimized provisioning approach provides stable results, since the deviation among data points is very small. We clearly see that even though our framework already dealt well with this rather large scenario, by introducing LEONORE local nodes we were able to further improve the overall provisioning time.

5.6. Final Remarks

After finishing our experiments and evaluating the results, we see that the introduction of LEONORE local nodes leads to a significant improvement in terms of provisioning time. Additionally, next to the measurement of the provisioning time, we also monitored the amount of data that gets transferred from the cloud to the edge infrastructure during the provisioning of gateway deployments. Our findings show that by using local nodes deployed in the edge, which are managing a cluster of gateways, we reduce the bandwidth usage drastically, since by using local nodes we avoid sending the provisioning package to each gateway, but only send this package once to a local node managing this cluster. Even by taking into account that the LEONORE local nodes need to be provisioned initially, does not diminish the savings. In order to illustrate the bandwidth savings, we will use some numbers from the evaluation scenario above. Let us consider the scenario where we need to provision 1000 gateways with the SVM application package, which has a size of 120KB. With the original provisioning approach, two server-side nodes of LEONORE would transfer the package to each gateway, resulting in 120MB of data that gets sent from the cloud to the edge for each provisioning request. Compared to that, by using LEONORE local nodes we cluster the gateway deployment to 8 clusters and therefore deploy 8 LEONORE local nodes. The local node application package has a size of 14MB and therefore the transferred data sums up to 112MB. Additionally, when provisioning the 1000 gateways we now transfer the SVM application package only to these 8 local nodes and therefore produce in total 112.96MB. For a the relatively small SVM application package, we already save 6% for the initial provisioning cycle. After that, since the LEONORE local nodes are already deployed, we save 99% of the bandwidth for every additional provisioning request.

6. RELATED WORK

Since current applications in IoT are receiving a lot of attention, we notice that the scale of these applications can vary from embedded services to enterprise applications. In order to address different ways of designing, developing, deploying and managing applications not only in the cloud, but also in the underlying IoT infrastructure, [Oriwoh et al. 2013] presents initial guiding principles. In addition, [Theodoridis et al. 2013] presents challenges for building a city-scale IoT framework, where among others the important challenge of fine-grained provisioning and management of resources is discussed. To tackle some of the aforementioned challenges, [Bauer et al. 2013] defines an abstract IoT reference architecture for standardizing the Internet of Things. In addition, [Sehgal et al. 2012] addresses general problems of managing resource constrained devices, which are often used for building IoT solutions, by adopting existing network management protocols. Based on general challenges and reference architectures, platforms specifically targeted for deploying and provisioning of IoT applications emerged. INOX [Clayman and Galis 2011] is a robust and adaptable Platform for IoT that provides enhanced application deployment capabilities by creating a resource overlay to virtualize the underlying IoT infrastructure. An additional abstraction layer on top of the IoT infrastructure is frequently used in the literature (e.g., [Li et al. 2013a; Murphy et al. 2013; Li et al. 2013b]), which allows keeping the underlying infrastructure untouched when deploying an IoT solution. In contrast, our approach also considers IoT devices as first-class execution environments, which provides more control and better resource utilization. The Smart-M3 platform [Korzun et al. 2013] aims to create a M3 space by deploying agents on IoT devices that interact based on a space-based communication model. Although the authors mention the provisioning of IoT devices, they solely focus on the actual application design. [Chen et al. 2011] introduce over the air provisioning of IoT devices using Self Certified ECDH authentication technology. Although this approach shares the same general idea, the authors explicitly focus on one specific device and do not provide a general and scalable approach. [Papageorgiou et al. 2014] presents a solution for automatic configuration of IoT devices based on interpretable configurations. Compared to our approach, the authors assume pre-installed application components on IoT devices and only focus on provisioning application-specific configurations. [Li et al. 2013b] presents an approach to deploy applications on IoT devices by facilitating TOSCA. Since changing applications at runtime is not addressed, our approach can be considered an extension to this work.

Configuration management (CM) solutions represent another important area of interest, which in general address a similar problem. The most prominent representatives being Chef[18] and Puppet[19]. However, current tools come with the following limitations that make them unsuitable for the IoT domain. First, they are inherently pull based approaches with clients running on the respective machines, making push based hot fixes (e.g. important security updates) impossible. Second, dependency resolution is usually handed off to a distribution package manager, which is not suitable for the strongly resource-constrained environments we are dealing with.

Finally, since our approach provides an optimization for provisioning edge devices in the domain of building management, we also have to consider relevant work in this research topic. Among others, [Petri et al. 2014] presents an approach that achieves energy related optimizations for buildings by running simulations in the cloud. In addition, [Petri et al. 2015] proposes a service-oriented platform that allows performing (near) real-time energy optimizations for buildings. While these approaches specifi-

[18] http://chef.io
[19] http://puppetlabs.com

cally focus on energy optimizations in buildings at the application level, our framework aims at optimizing deployment topologies on the infrastructure level.

7. CONCLUSION

In this paper we presented LEONORE, a novel infrastructure and toolset to elastically provision application packages on resource-constrained, heterogeneous edge devices in large-scale IoT deployments. LEONORE enables push-based as well as pull-based deployments supporting a vast array of different IoT topology and infrastructure requirements. By introducing the concept of LEONORE local nodes we further enabled efficient distributed deployments in these constrained environments in order to further improve scalability and reduce generated network traffic between cloud and edge infrastructure. For evaluation purposes we utilized a large scale testbed based on a real-world industry scenario. Our evaluation clearly demonstrated that LEONORE is able to elastically provision large numbers of devices in an efficient manner. We further showed that our local node extension significantly improved provisioning time while drastically reducing bandwidth consumption, a factor that is crucial in such constrained environments.

In our ongoing and future work, we plan to further extend LEONORE to address additional challenges. We see the necessity to improve our IoT gateway representation to better utilize the underlying device-specific capabilities and develop techniques to allow the creation and deployment of more flexible IoT applications. Additionally, we want to introduce update priorities, in order to allow for clear distinctions between ordinary and important (e.g., security patches) updates, since delays in these important updates can expose the infrastructure to severe security risks. Next, to allow LEONORE to scale across privately managed infrastructure boundaries, we aim to address emerging security aspects such as authentication and authorization. In order to provide more efficient provisioning, we intend to evaluate a hybrid push/pull provisioning strategy. Furthermore, we plan to integrate and align LEONORE with our work on software-defined IoT systems, where it will serve as an integral part of IoT infrastructure management [Schleicher et al. 2015; Vögler et al. 2015b]. Finally, we will explore how different application development methodologies (e.g., [Inzinger et al. 2014]) can be extended in order to efficiently support application design, deployment, and composition considering large numbers of IoT gateways to perform certain parts of application business logic.

REFERENCES

Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. 2010. Big data and cloud computing: new wine or just new bottles? *Proceedings of the VLDB Endowment* 3, 1-2 (Sept. 2010), 1647–1648. DOI:http://dx.doi.org/10.14778/1920841.1921063

Martin Bauer, Mathieu Boussard, Nicola Bui, Jourik De Loof, Carsten Magerkurth, Stefan Meissner, Andreas Nettsträter, Julinda Stefa, Matthias Thoma, and JoachimW Walewski. 2013. IoT Reference Architecture. In *Enabling Things to Talk*. Springer Berlin Heidelberg, 163–211–211. DOI:http://dx.doi.org/10.1007/978-3-642-40403-0_8

Deji Chen, Mark Nixon, Thomas Lin, Song Han, Xiuming Zhu, Aloysius Mok, Roger Xu, Julia Deng, and An Liu. 2011. Over the air provisioning of industrial wireless devices using elliptic curve cryptography. In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*. 594–600. DOI:http://dx.doi.org/10.1109/CSAE.2011.5952541

Stuart Clayman and Alex Galis. 2011. INOX: a managed service platform for inter-connected smart objects. In *IoTSP '11: Proceedings of the workshop on Internet of Things and Service Platforms*. ACM Request Permissions. DOI:http://dx.doi.org/10.1145/2079353.2079355

Li Da Xu, Wu He, and Shancang Li. 2014. Internet of Things in Industries: A Survey. *IEEE Transactions on Industrial Informatics* 10, 4 (2014), 2233–2243. DOI:http://dx.doi.org/10.1109/TII.2014.2300753

Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. 2014. MADCAT - A Methodology for Architecture and Deployment of Cloud Application Topologies. In *Pro-*

ceedings of the 8th International Symposium on Service-Oriented System Engineering. IEEE, 13–22. DOI:http://dx.doi.org/10.1109/SOSE.2014.9

Dmitry G Korzun, Sergey I Balandin, and Andrei V Gurtov. 2013. Deployment of Smart Spaces in Internet of Things: Overview of the Design Challenges. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 48–59–59. DOI:http://dx.doi.org/10.1007/978-3-642-40316-3_5

Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. 2013a. Efficient and Scalable IoT Service Delivery on Cloud. In *IEEE 6th International Conference on Cloud Computing*. 740–747. DOI:http://dx.doi.org/10.1109/CLOUD.2013.64

Fei Li, Michael Vögler, Markus Claeßens, and Schahram Dustdar. 2013b. Towards Automated IoT Application Deployment by a Cloud-Based Approach. In *IEEE 6th International Conference on Service-Oriented Computing and Applications*. 61–68. DOI:http://dx.doi.org/10.1109/SOCA.2013.12

Shancang Li, Li Da Xu, and Shanshan Zhao. 2014. The internet of things: a survey. *Information Systems Frontiers* (April 2014), 1–17. DOI:http://dx.doi.org/10.1007/s10796-014-9492-7

Sean Murphy, Abdelhamid Nafaa, and Jacek Serafinski. 2013. Advanced service delivery to the Connected Car. In *IEEE 9th International Conference on Wireless and Mobile Computing, Networking and Communications*. 147–153. DOI:http://dx.doi.org/10.1109/WiMOB.2013.6673354

Sam Newman. 2015. *Building Microservices*. O'Reilly Media, Inc.

Edewede Oriwoh, Paul Sant, and Gregory Epiphaniou. 2013. Guidelines for Internet of Things Deployment Approaches – The Thing Commandments. *Procedia Computer Science* 21 (2013), 122–131. DOI:http://dx.doi.org/10.1016/j.procs.2013.09.018

Apostolos Papageorgiou, Manuel Zahn, and Ernö Kovacs. 2014. Auto-configuration System and Algorithms for Big Data-Enabled Internet-of-Things Platforms. *IEEE International Congress on Big Data* (2014), 490–497. DOI:http://dx.doi.org/10.1109/BigData.Congress.2014.78

Ioan Petri, Haijiang Li, Yacine Rezgui, Yang Chunfeng, Baris Yuce, and Bejay Jayan. 2014. A modular optimisation model for reducing energy consumption in large scale building facilities. *Renewable and Sustainable Energy Reviews* 38 (2014), 990–1002. DOI:http://dx.doi.org/10.1016/j.rser.2014.07.044

Ioan Petri, Yacine Rezgui, Tom Beach, Haijiang Li, Marco Arnesano, and Gian Marco Revel. 2015. A semantic service-oriented platform for energy efficient buildings. *Clean Technologies and Environmental Policy* 17, 3 (2015), 721–734. DOI:http://dx.doi.org/10.1007/s10098-014-0828-2

Johannes M Schleicher, Michael Vögler, Christian Inzinger, and Schahram Dustdar. 2015. Smart Fabric An Infrastructure-Agnostic Artifact Topology Deployment Framework. In *Mobile Services (MS), 2015 IEEE International Conference on*. IEEE, 320–327. DOI:http://dx.doi.org/10.1109/MobServ.2015.52

Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla, and Jürgen Schonwalder. 2012. Management of resource constrained devices in the internet of things. *Communications Magazine, IEEE* 50, 12 (2012), 144–149. DOI:http://dx.doi.org/10.1109/MCOM.2012.6384464

Ganesh Shrestha and Jürgen Jasperneite. 2012. Performance Evaluation of Cellular Communication Systems for M2M Communication in Smart Grid Applications. In *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 352–359–359. DOI:http://dx.doi.org/10.1007/978-3-642-31217-5_37

Evangelos Theodoridis, Georgios Mylonas, and Ioannis Chatzigiannakis. 2013. Developing an IoT Smart City framework. In *4th International Conference on Information, Intelligence, Systems and Applications*. 1–6. DOI:http://dx.doi.org/10.1109/IISA.2013.6623710

Michael Vögler, Fei Li, Markus Claeßens, Johannes M Schleicher, Sanjin Sehic, Stefan Nastic, and Schahram Dustdar. 2015a. COLT Collaborative Delivery of Lightweight IoT Applications. In *Internet of Things. User-Centric IoT*. Lecture Notes of the Institute for Computer Sciences, Vol. 150. Springer International Publishing, 265–272. DOI:http://dx.doi.org/10.1007/978-3-319-19656-5_38

Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. 2015b. DIANE - Dynamic IoT Application Deployment. In *Mobile Services (MS), 2015 IEEE International Conference on*. IEEE, 298–305. DOI:http://dx.doi.org/10.1109/MobServ.2015.49

Michael Vögler, Johannes M. Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. 2015c. LEONORE – Large-Scale Provisioning of Resource-Constrained IoT Deployments. In *9th International Symposium on Service-Oriented System Engineering (SOSE'15)*. 78–87. DOI:http://dx.doi.org/10.1109/SOSE.2015.23

Qian Zhu, Ruicong Wang, Qi Chen, Yan Liu, and Weijun Qin. 2010. IOT Gateway: BridgingWireless Sensor Networks into Internet of Things. In *IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing*. 347–352. DOI:http://dx.doi.org/10.1109/EUC.2010.58

Received July 2015; revised October 2015; accepted November 2015