Towards IoT Diversity via Automated Fleet Management

Rustem Dautov
SINTEF Digital
Oslo, Norway
Email: rustem.dautov@sintef.no

Hui Song
SINTEF Digital
Oslo, Norway
Email: hui.song@sintef.no

***Abstract*—Large-scale Internet of Things (IoT) systems are characterised by an increased level of heterogeneity, both in terms of hardware and software caused by varying device functionality, capabilities and performance. Moreover, since agile business requirements force IoT vendors to continuously modify the software components deployed at the Edge, even initially uniform devices constituting a common IoT ecosystem might end up running software differing in individual components and/or configurations. The amount of effort required to maintain and operate such an increasingly diverse ecosystem – *i.e.* to perform fleet management – grows proportionally to the size and complexity of the IoT fleet, and is especially important for IoT vendors aiming to achieve economies of scale. To address this challenge, this paper proposes a model-based diversity engineering approach to enable automated fleet management. Based on a model of an IoT system with fine-grained modifications to be applied, the proposed approach is able to diversify and manage large-scale IoT systems at run-time. As a proof of concept, the proposed approach was implemented on top of the Azure IoT Hub fleet management facilities, and validated on a Remote Patient Monitoring use case scenario.**

***Index Terms*—Internet of Things, Edge Computing, Diversity Engineering, Fleet Management, Azure IoT Hub.**

1. Introduction

The Internet of Things (IoT) facilitates creation of smart spaces by converting existing environments into sensor-rich cyber-physical systems. As IoT ecosystems grow in size and complexity, they become increasingly heterogeneous, especially at the very bottom layer, constituted by Edge infrastructures. At this layer, Edge devices, albeit belonging to a common IoT ecosystem, might considerably differ both in terms of their hardware (CPU type, networking interfaces, available sensors/actuators, *etc.*) and software (operating systems, programming languages, libraries, communication protocols, *etc.*) stacks. The former is relatively static, since new devices, once deployed and connected, are typically not expected to update their hardware configuration at run-time.

On contrary, the software configuration of devices constituting the very Edge of an IoT network is expected to continuously evolve during system operation. Examples of such changes may include security patches, user-specific configurations, upgrades to a new version of a software library, or an introduction of a new feature. As a result of this software evolution, initially uniform devices constituting a common IoT system might end up running software stacks, differing in their individual components and/or configurations. Going beyond the traditional notion of a technique for increasing system security and fault-tolerance, we refer to this phenomenon as *software diversity* to describe a wide range of fine-grained software modifications, applied by an IoT vendor and driven by emerging business requirements.

At the same time, it becomes challenging to operate increasingly diverse systems and react to emerging requirements in an agile manner. With the recent advances in hardware, networking, and containerisation technologies, this challenge, known as *fleet management*, is partially addressed by some IoT cloud platforms, which made it possible to remotely access individual Edge devices, deploy containerised applications in a standardised automated manner, and monitor an IoT system at run-time. Such cloud platforms, however, only provide some basic tools and lack an intelligent mechanism that would take into account business requirements and drive the software diversification process in an agile, fine-grained, and secure manner.

To this end, this paper proposes a model-based approach to diversity-oriented fleet management in the context of large-scale IoT deployments. The approach is inspired by and further extends the concept of software diversity, which acts as a reference technique to enable multiple fleet management use cases. Accordingly, the main contribution of the paper is three-fold: *i)* a model-based, diversity-oriented approach to fleet management, *ii)* a diversity-oriented metamodel, and *iii)* a proof-of-concept implementation of the fleet management system in the context of a Smart Healthcare scenario.

The rest of the paper is organised as follows. Section 2 explains the motivation behind the proposed research using a Smart Healthcare scenario. Section 3 provides background information on IoT fleet management and relevant IoT cloud platforms, as well as software diversity. It also briefly describes the current state of the art and existing gaps. Section 4 presents the overall approach with its conceptual architecture, and further elaborates on it with the details

CEUR-WS.org/Vol-2442/paper7.pdf

Copyright ©2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

of model-based diversity engineering for fleet management. Section 5 puts theory into practice by explaining how the proposed approach is applied to the previously introduced Smart Healthcare scenario. Section 6 concludes the paper and outlines some directions for future work.

2. Motivating Example: Remote Patient Monitoring

Healthcare is one of the many domains, continuously improved by the pervasive penetration of IoT technologies, which are used to support core functions of healthcare institutions. Traditional hospitals and private houses are converted into smart digital environments, extensively making use of interconnected sensor systems to continuously collect and transfer biometric data for timely analysis. Such smart telemedicine relies on a powerful infrastructure stack that includes sensor-enabled 'leaf' IoT devices, Edge devices (*e.g.* gateways or other network nodes close to IoT devices), wired/wireless networks, Cloud platforms, *etc*. [1].

In this context, a simple, yet realistic scenario highlighting the challenges associated with software diversity at the Edge is Remote Patient Monitoring (RPM). As depicted in Fig. 1, an RPM system, to which we will refer throughout this paper, assumes that a single installation in a residential building involves multiple sensor-enabled microcontrollers (*i.e.* IoT devices) for monitoring patients and collecting biometric/environmental data. There is also a Linux-based Edge device that acts as a hub for collecting and preprocessing raw sensor data, and further transferring them to a Cloud-based back-end application. The Edge device is equipped with several networking interfaces to interact with IoT devices and the Cloud. It also provides an interface for end users to configure the installation according to individual requirements. Particular sets of IoT devices differ from one installation to another – *i.e.* some houses might be equipped with fall detection sensors, whereas the others are suited for sleep tracking. Through a Cloud-based management console, the system administrator is able to monitor the status of each installation and push software updates to Edge devices. The RPM software comprises two parts:

- *Front-end application* runs on IoT and Edge devices and is in charge of collecting and transferring data from the surrounding physical environment (including humans), interacting with end users, and acting on the environment via actuators when necessary.
- *Back-end application* runs on the Cloud and is in charge of data analytics and decision making based on the data received from the front-end application.

To remain competitive on the market and offer a user-tailored RPM solution, the IoT vendor has to react to new business requirements in an agile and flexible manner. In these circumstances, it is expected that devices, initially uniform both in terms of hardware and software, will become increasingly diverse in terms of software. This software diversity, introduced by the vendor, goes beyond the traditional notion of a technique for improving system security

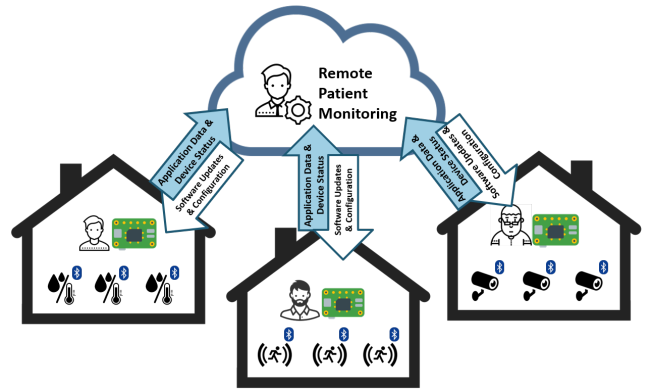

Figure 1. Remote Patient Monitoring system.

and fault-tolerance [2] and is motivated by several other requirements. More specifically, taking the RPM scenario as a reference, the following diversity aspects can be observed:

- *User-tailored configuration*: different users may have different preferences with regard to the functionality, performance, or security/privacy settings. Therefore, the same front-end application, when deployed on different premises, may be accordingly configured/customised by the vendor, the end user, or a third party.
- *Hardware-specific configuration*: large-scale IoT systems are highly heterogeneous in terms of underlying hardware infrastructure (different CPU types, networking interfaces, available sensors/actuators, *etc*.), which requires software to be accordingly configured/customised. Also, in some cases, relatively powerful Edge device may be tasked with local data pre-processing, whereas less powerful ones are configured to only push data to the Cloud, which also results in uneven distribution of software components among Edge devices.
- *Targeted software updates*: as part of a continuous software development process, it is a common practice to maintain more than one running version in production. For example, a widely used technique is the so-called A/B testing, when a sub-set of users is selected to try new features, while the majority of users still keeps using a previous stable release.
- *Asynchronous updates*: Edge devices can unpredictably lose connectivity (*e.g.* due to unstable wireless connection or limited power). During such (unpredictably long) outages, new features or security patches may be applied to the software. As a result, a disconnected device may possibly miss several rounds of updates and thus run an obsolete version of software, thus introducing an accidental complexity in the operation of the system.
- *Synthetic diversity*: vendors may use various techniques to deliberately diversify their software for improving security (*i.e.* implementing a 'moving target' for security attacks) or fault-tolerance (*i.e.* recovering a failed system to a different version not to suffer from the same fault again).

Each of these aspects requires the IoT vendor to diversify their IoT system by deploying and maintaining multiple

software versions for their fleets of IoT/Edge devices. Furthermore, the vendor needs to match software configurations/customisations with target systems, and push software updates to relevant devices in a timely and guaranteed manner. This increases the complexity of IoT operation, challenging the agility, reliability and economy of scale. Admittedly, addressing this challenge in the context of large-scale and highly-distributed IoT deployments goes beyond manual capabilities of the IoT vendor and requires an automated fleet management approach to enable remote, secure, and reliable support for multiple operations, including deployment, upgrade, monitoring, and troubleshooting of software components.

3. Background and Related Work

3.1. Fleet Management through Cloud Platforms

Fleet management is a cross-cutting multi-faceted concern for IoT vendors, especially when scaling from local installations to global distributed ecosystems. In this paper, we focus on the *control* part, *i.e.* the deployment and upgrading of software components of the front-end applications on the IoT/Edge devices. In this respect, we identify the following key requirements for effective and efficient fleet management:

- *Agility*: Software deployment or update of the entire fleet must be done within an acceptable time frame. In the context of widely adopted DevOps practices, when vendors aim to continuously introduce new features or patch vulnerabilities, such duration is measured with hours or even minutes. Admittedly, achieving such short-term agility implies having an *automated* solution able to *remotely* access the managed system.
- *Reliability*: IoT vendors must be ensured at all times that the designated software is eventually deployed to target end users, even in the presence of unstable network connectivity and varying physical conditions of the Edge environment.
- *Economy of scale*: As the number of end users increases, average operation costs per installation should decrease, so that the vendor can benefit from the growth.

These requirements are partially addressed by IoT cloud platforms – an emerging family of cloud solutions, which, apart from offering the traditional computing and storage resources, also provide an IoT-specific management layer for device monitoring, data flow design, data visualisation, *etc*. These existing platforms [3], [4] enable IoT developers to discover and integrate devices, monitor and diagnose system operation, as well as to collect, process, store, and visualise telemetry data. This typically assumes that Edge devices are deployed and manually configured to push collected data to a centralised back-end Cloud service in a 'vertical', unidirectional manner [5]. Information exchange in the opposite direction (*e.g*. actuation commands, re-configuration, software/firmware updates) is not straight-forward and easily implemented due to network barriers, absence of static IP addresses, constrained hardware/software capabilities of devices, limited connectivity, *etc*. In such circumstances, agile software development for IoT/Edge devices in a truly DevOps-compliant manner is hardly possible, as there is an inevitable element of manual work required to deploy software on a remote device.

This situation is changing with the recent advances in the networking and containerisation technologies. The former enabled accessing and interacting with remote devices by establishing virtual private networks, while the latter allowed deploying and running light-weight and isolated software components in a platform-agnostic manner. These advances have also been underpinned by the continuously increasing computing and networking capabilities of Edge/IoT devices. As a result, existing IoT platforms are now able to extend their functionality with tools for remote deployment of software components (packaged as Docker containers) and management of IoT devices.

3.2. Software Diversity

Traditionally, the scope of software diversity has been limited to the two main fields – namely, cyber-security and fault-tolerance [2]. The former case relies on applying various randomisation techniques at different system levels to make software less vulnerable to generic threats by becoming a 'moving target' for them. In the latter case, software systems with diversified functions and elements are expected to handle various failures either by completely avoiding them, or by recovering an affected system to a different, diversified version, thus lowering the risk of suffering from the same fault again.

Currently, the concept of software diversity appears as a rich and manifold notion with multiple facets, such as the goal of diversity, the diversification techniques, the scale of diversity, the application domain, when it is applied, *etc*. [2]. It is also common to distinguish between *natural* (*emerging*) diversity, which appears spontaneously from the software development process and run-time operation, and results in different software versions, yet with similar functionality, and *artificial* (*synthetic* or *automated*) diversity, which is a result of explicit diversification actions taken by the IoT vendor and applied to the system.

3.3. Existing solutions and Related Work

As of July 2019, there are at least three IoT cloud platforms already offering support for container-based fleet management as part of their portfolio. As a prerequisite, these solutions require Edge devices to be pre-installed with a software agent to communicate with the Cloud counterpart and a container engine to run containerised software components. Among the available alternatives, *Azure IoT Hub*[1] appears as the most advanced option. On the one hand, it offers a rich ecosystem of various tools and services through its marketplace (*e.g*. users can benefit from an existing Docker

[1] https://azure.microsoft.com/services/iot-hub/

image repository, an automatic device provisioning service, or a certificate-based authentication and access control), and, unlike *AWS IoT Greengrass*,[2] supports full-featured containers, not just serverless functions. On the other hand, it outperforms *Balena Cloud*,[3] which also provides similar full-featured support for container management, but is not yet mature and developed enough to offer an extensive collection of of tools and services via a marketplace.

The community-driven Eclipse hawkBit[4] is another relevant framework aiming to automate IoT software updates at scale. Unlike the proprietary Cloud-based solutions, it is agnostic to specific underlying technologies and can be integrated with many third-party components (albeit at the cost of increased manual integration and configuration effort).

Despite the advanced built-in functionality offered by all these frameworks, they still lack another policy-driven control layer that would enable flexible, fine-grained and diversity-oriented software management and address the previously outlined fleet management requirements. Potential solutions, albeit not primarily tailored to the purposes of IoT fleet management, already exist. More specifically, for some years now, multiple tools have been available on the market to support the deployment and configuration of software systems, *e.g*. Puppet,[5] Chef,[6] and new tools emerged for deployment of cloud-based systems such as CloudMF [6], OpenTOSCA [7], and Brooklyn.[7] In addition, similar tools focus on the management and orchestration of containers, such as Kubernetes.[8] When dealing with large-scale systems, these approaches focus on scaling out, *i.e*. duplicating identical components for load balancing.

As far as IoT fleet management is concerned, a systematic literature review of 17 prominent approaches for orchestration and deployment for the IoT was conducted in [8]. All the surveyed approaches focus on the automatic deployment of one or many IoT systems according to a single deployment specification, without support for managing a diverse fleet of IoT systems according to multiple specifications.

With the proposed approach, we aim to bridge this identified gap by enhancing the existing Cloud-based tools for fleet management with support for intelligent diversity-aware deployment and maintenance of software components. To validate the proposed approach, we will take Azure IoT Hub as the baseline, which currently appears to be the most mature and feature-rich (yet open-source and free for research purposes) offering. It is, nevertheless, expected that the rest of the IoT cloud market will soon catch up with the fleet management trend by offering similar container-based functionality.

[2]https://aws.amazon.com/greengrass/
[3]https://www.balena.io/cloud/
[4]https://www.eclipse.org/hawkbit/
[5]https://puppet.com/
[6]https://www.chef.io/chef/
[7]https://brooklyn.apache.org/
[8]https://kubernetes.io/

4. Proposed Approach

In this paper, we propose a model-based approach to diversity-oriented fleet management of IoT systems, and implement a prototype fleet management tool named DivEnact.[9] We primarily focus on the connection between IoT fleet management and software diversity, regardless of whether it naturally emerges or is synthesised at design- and development-time. Driven by a diversity-oriented model, the proposed DivEnact solution is able to provide automatic run-time management to deploy the diversified software onto individual IoT/Edge devices.

4.1. Conceptual Architecture

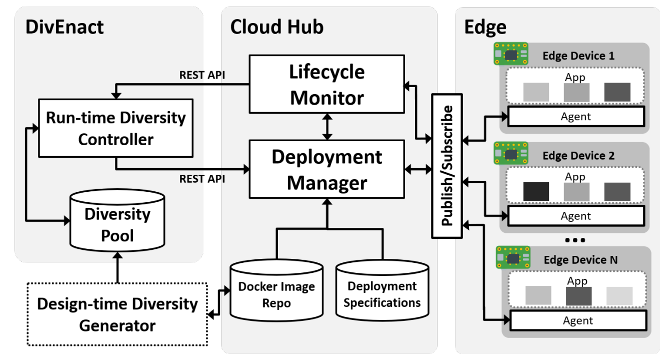

Figure 2. System architecture.

Fig. 2 illustrates the overall architecture of the DivEnact approach, which we conceptually split into three main parts.

1) At the *Edge*, there is a fleet of multiple IoT installations. Each installation serves an end user and has one central Edge device, which uses local wireless channels (*e.g*. WiFi and Bluetooth) to control and communicate with coupled IoT devices belonging to the same installation. For clarity purposes, each installation is represented only by a corresponding Edge device, while the underlying IoT devices are omitted. It is assumed that each Edge device hosts a copy of the front-end application, which consists of a set of software components running as Docker containers (depicted as multicoloured boxes). Under this assumption, fleet management can be simplified to a problem of managing containers running on Edge devices.
2) The edge devices constituting the fleet are registered to a central *Cloud Hub* and, through publish/subscribe messaging, are able to continuously report on their current status and receive management instructions. The initial registration and further interaction with the Hub is undertaken by a device-side software agent, which communicates to the following two services provided by the Cloud Hub:

[9]**Div**ersity **Enact**ment for IoT fleets.

- *Life-cycle Monitor* collects reported information from Edge devices (*e.g.* connection status, deployed components, system up-time, *etc.*) and keeps track of the fleet's state.
- *Deployment Manager* decides what components should be deployed on each Edge device, and enacts this deployment on the device. It maintains a repository of deployment specifications, each of which is defined with a scope of applicable Edge devices. For devices that fall into the scope of a specification, the Deployment Manager will enforce the devices to run the containers as specified, but sending corresponding instructions. Docker container images are assumed to be stored in a publicly available repository, such as Docker Hub.[10]

3) The *DivEnact* component interacts with the Cloud Hub through a REST API for querying and configuring Edge device properties, and for manipulating deployment specifications. Based on this API, the *Diversity Controller* implements diversity-oriented fleet management at run-time. It maintains a *Diversity Pool* – a collection of all potential variants of the front-end application used to generate deployment specifications for the Deployment Manager. The Diversity Controller also maps deployment specifications to specific devices by matching device properties with specification scopes. These potential diversity variants are either manually synthesised by the vendor, or automatically generated at design-time. The design-time diversity generation, at both code and architecture levels, is included in the overall architecture, but goes beyond the scope of this paper. Further details on the model-based underpinnings of the Diversity Controller are discussed below.

4.2. Model-Based Diversity Engineering

At the core of the proposed DivEnact approach is the *Diversity Controller*, which maintains a run-time model of the IoT fleet and a diversity configuration to be applied. Following the *models@runtime* pattern [9], the model is dynamically synchronised with the running system, so that the IoT fleet can be monitored and manipulated by reading and writing the model, either manually or programmatically.

Fig. 3 illustrates key concepts and relationships, which constitute a meta-model for expressing a diversity-oriented IoT fleet management scenario. Bold lines represent relationships directly editable by the IoT vendor, while thin lines indicate the relationships derived at run-time by the Diversity controller. The IoT vendor can control the system by adding/removing model elements, changing their attributes, or editing relationships between them. The DivEnact run-time engine will automatically maintain the derived relationships based on the edits, and apply the changes to the underlying system, *i.e.* the IoT fleet under management.

The concepts that directly represent the IoT fleet are **`Device`** and **`ModuleInstance`**. The former represents an

[10]https://hub.docker.com/

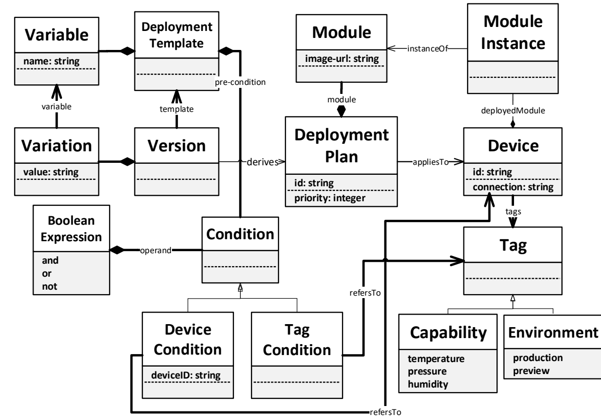

Figure 3. Meta-model for diversity-oriented IoT Fleet Management.

Edge device in an IoT installation, and the latter represents a software component running on the device as a Docker container. A device is identified by its **`id`** and a unique **`connection`** string used for registering to the Cloud Hub. A device may be annotated by a **`Tag`**, which is a user-defined value used to limit the target scope of a deployment specification. For example, two sample **`Tag`** enumerations shown in Fig. 3 are **`Capability`** and **`Environment`**, which reflect hardware capabilities of a device and its application environment, respectively. The relationship between devices and modules is derived – *i.e.* the IoT vendor cannot directly assign modules to an Edge device, but rather they will be specified with an application scope in a deployment specification.

To express deployment specification, the IoT vendor defines a set of **`DeploymentPlans`**, each of which contains a set of **`Modules`**. Briefly, a **`Module`** serves to define how to deploy a piece of software on a device. For example, as depicted in Fig. 3, a **`Module`** may specify a URL of a container image, as well as parameters to instantiate the container from this image. If target conditions of a **`DeploymentPlan`** are satisfied by a device, then each **`Module`** defined in this plan will have a corresponding **`ModuleInstance`** deployed on the device. Accordingly, the **`appliesTo`** relationship between a **`DeploymentPlan`** and a **`Device`** will be derived from the **`Conditions`** of the plan.

A **`Condition`** is defined using either a device **`ID`** (*i.e.* **`DeviceCondition`**) or **`Tags`** (*i.e.* **`TagCondition`**) attached to devices. The former only applies to a device with this specific ID, whereas the latter applies to multiple devices annotated with this specific **`Tag`**. A composite condition can include several expressions, connected by Boolean operators **`and`**, **`or`**, and **`not`**, as illustrated by the code snippet below, limiting the scope of a **`DeploymentPlan`** to devices in a preview environment and either equipped with a temperature sensor or having an empty **`ID`** (used for testing purposes).

```
tags.Environment == preview and
  ( tags.Capability == temperature or
    ID == EmptyForTesting )
```

The IoT vendor can create a **DeploymentPlan** manually, setting up its conditions and device tags to define the scope of the deployment. However, when the size of the fleet and the number of diverse deployment plans increase, such manual control would be time-consuming and error-prone. We introduce the diversity generation part into the meta-model to handle the automated creation of deployment specifications. The vendor can start by defining a **DeploymentTemplate**, which aggregates a number of **DeploymentPlans**, and defines a common part shared by these plans. The varying parts of the plans are defined as **Variables** in the template. From each template, the vendor can define several **Versions**, each of which contains **Variations** that assign concrete values to **Variables**. A template is essentially a text document with a deployment specification, expressed in YAML following the widely adopted 'Infrastructure as Code' trend.

4.3. Run-time Model Synchronisation

Following the models@runtime pattern, the IoT vendor is able to manipulate the fleet management model (*e.g.* add new elements, change attributes, and edit relationships), and the *Diversity Controller* will automatically apply these changes to the running devices. Depending on the changes, the synchronisation happens in three different levels.

1) In-model synchronisation, happening after changes on parts of models for deployment templates and version, performed by the diversity controller. The controller collects the user-input changes and directly change other parts of models to maintain the consistency.
2) Model-to-hub synchronisation, happening after changes on the deployment plans and modules, performed also by diversity controller. The controller collects the model changes and transforms them into invocations to the REST API provided by the IoT Hub, in order to update the deployment plans maintained by the hub.
3) Hub-to-device synchronisation, happening after changes on deployment plan and device, performed by the IoT Hub. Such synchronisation is triggered automatically after the API invocations to the IoT Hub, and will eventually result in containers deployed on the devices.

Typically, the synchronisation happens as a sequence in the following three steps, corresponding to the three levels.

1) *Creation*: this step takes place when the vendor creates a deployment plan from one of the templates from the Diversity Pool. Instead of manually defining modules for a deployment plan, the vendor can instead create an empty plan and link it to a version. The synchronisation engine will load the template used by this version, and resolve template variables to generate module definitions. These modules will then be inserted into the deployment plan. If the template contains pre-conditions, the engine will also pass them into the deployment plan.
2) *Instantiation*: when these model elements are created, removed, or edited, the Diversity Controller synchronises the changes with the Deployment Manager by invoking the API provided by the Cloud Hub. The Diversity Controller will first invoke the API to create a deployment plan with the same ID in the Hub, and use a series of subsequent API invocations to create the modules and insert them into the newly created deployment plan. Similarly, when the vendor tags a device, the engine will get its ID call the corresponding API method passing the tag value and the device ID.
3) *Enactment*: we reuse the management features provided by the Deployment Manager to enact the changes on Edge devices. When a deployment plan is created or updated, for each device falling into the scope, the Deployment Manager will evaluate if the desired modules (the ones defined in the deployment plan) are already running on the device by querying the Life-cycle Monitor. If not, it will instruct the device to download and deploy the desired module. In case a device is not currently connected to the Hub, the pending deployment action will be triggered immediately after the device appears online again.

5. Proof of Concept[11]

The proof of concept is currently implemented on top of Azure IoT Hub, which offers wide integration opportunities and rich built-in functionality to be re-used. Azure IoT Hub already provides its built-in Life-cycle Manager, Deployment Manager, and a software agent to be installed on Edge devices, which exempts us from 're-inventing the wheel' and focus on implementing the required DivEnact functionality.

5.1. Experimental Setup

We now re-visit the reference RPM scenario to demonstrate the viability of the proposed approach. This simplified demonstration uses three Raspberry Pi boards acting as Edge devices and three SenseHat[12] shields acting as sensor-enabled IoT devices. Each SenseHat shield is equipped with a LED matrix and three sensors (humidity, temperature, and pressure) that can be used to collect information about the surrounding environmental conditions. Table 5.1 summarises the available capabilities of each Raspberry Pi board. The front-end application deployed on Raspberry Pi boards can interact with the SenseHat shields to light up the LED matrix, collect sensor data and transfer them to the back-end application, which is assumed to evaluate whether current conditions in a patient's house are acceptable.

The fleet of these three Edge devices and a sample deployment plan are modelled in Fig. 4. Each Edge device is running in a production environment and is tagged with its capabilities (as summarised in Table 5.1). The model also contains a deployment plan with two modules (*i.e.* Docker

[11] Source code related to this implementation can be found in https://github.com/SINTEF-9012/divenact

[12] https://www.raspberrypi.org/products/sense-hat/

TABLE 1. TESTBED SETUP.

Edge Device	Temperature	Pressure	Humidity	LED Matrix
RPi_1	Yes	Yes	-	Yes
RPi_2	Yes	-	Yes	Yes
RPi_3	-	Yes	Yes	Yes

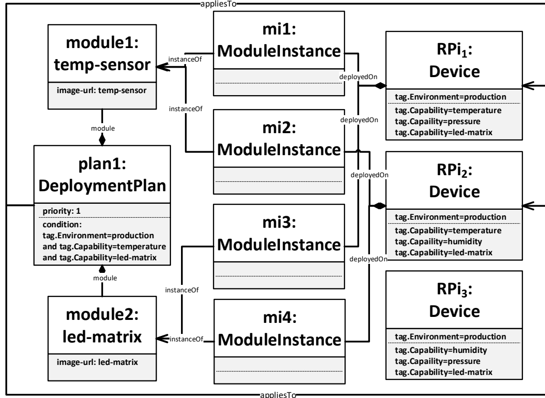

Figure 4. Sample deployment plan.

images) **`led-matrix`** and **`temp-sensor`** that contain corresponding application logic, while the latter is also able to set up the sampling frequency of a temperature sensor. The plan is generated from the following template:

```
templates:
  divenact-template:
    temp-sensor:
      settings:
        image: songhui/temp-sensor:{{FREQUENCY}}
        createOptions:
        "{"HostConfig":
          {"Privileged": true }}"
      type: docker
    led-matrix:
      settings:
        image: songhui/led-matrix
        createOptions:
        "{"HostConfig":
          {"Privileged": true }}"
      type: docker
    pre-condition: tags.Capability=temperature
        and tags.Capability=led-matrix
        and tags.Environment=production
versions:
  frequency-1:
    template: divenact-template
    parameter:
      FREQUENCY: 1
  frequency-10:
    template: divenact-template
    parameter:
      FREQUENCY: 10
```

The code snippet above shows a sample deployment template with two modules named **`temp-sensor`** and **`led-matrix`**, based on corresponding Docker images.[13] The **`temp-sensor`** image is registered with two different versions **`frequency-1`** and **`frequency-10`** with frequencies of 1 and 10 readings per second respectively, which can be retrieved by specifying the variable **`FREQUENCY`**. The template also has a composite pre-condition **`tags.Capability=temperature and tags.Capability=led-matrix and tags.Environment=production`**. As a result, the generated deployment plan only applies to RPi_1 and RPi_2, but not RPi_3 (which is not equipped with a temperature sensor). As a result, the module defined in the deployment plan will only be instantiated on RPi_1 and RPi_2.

[13]Please note the configuration parameter **`Privileged`** set to **`true`**, which is required to enable communication with a SenseHat shield through low-level GPIO pins.

5.2. Sample Diversity Use Cases

We now demonstrate how the Diversity Controller is able to support some of the diversity use cases outlined in Section 2. These diversity use cases are intended to demonstrate how the model-based approach simplifies automated IoT fleet management. Each use case is implemented as a simple, yet efficient script, which manipulates the model accordingly.

5.2.1. Pushing a new version to production. The IoT vendor has released a new version **`frequency-100`**, and wants it to be the only production version for all Edge devices equipped with temperature sensors. DivEnact can implement this as the following script:

```
for d in Device:
    d.tags.Environment = production
p = DeploymentPlan("production")
p.version = frequency-100
p.condition = "tags.Environment==production"
  and p.version.template.precondition
```

Briefly, the script sets all the devices into production environment, and then create a deployment plan **`p`** from the version **`frequency-100`**. The condition of **`p`** is a two-fold: *i)* the plan works for all devices tagged with the production environment, and *ii)* the plan inherits a pre-condition from the template, which limits its scope to devices equipped with a temperature sensor.

5.2.2. Previewing a staging version. The IoT vendor has developed a new version **`frequency-100`**, but before releasing it to production wants to test it on a limited number of selected users, while the rest of the users will still run the current production version. Extending the previous use case's model, the following script will accomplish this:

```
frequency-100 = Version('frequency-100')
temp-devices = [d for d in Devices
    if eval(frequency-100.template.precondition, d)]
for d in shuffle(temp-devices)[0:2]:
    d.tags.Environment = preview
p = DeploymentPlan('preview')
p.version = frequency-100
p.condition = 'tags.Environment==preview'
  and p.version.template.precondition
```

The script first finds all devices that satisfy the pre-condition of the version **`frequency-100`**, *i.e.* the devices that are tagged with **`Capability=temperature`**. After that, it randomly picks 2 devices and tags them with the **`preview`** environment. Finally, it creates a new deployment plan from this version, similar to the previous use case.

5.2.3. Shuffling versions. Given the three available versions for the temperature sensor module (**`frequency-1`**, **`frequency-10`**, and **`frequency-100`**) and one version for the LED matrix (**`led-matrix`**), the IoT vendor wants to artificially synthesise diversity for increased security and fault-tolerance by deploying several diversified, but functionally equivalent versions onto the fleet.

```
group1 = [frequency-1, frequency-10, frequency-100]
group2 = [led-matrix]
v2c = {}
for group in [group1, group2]:
  devices = [d for d in Devices if
    eval(group[0].template.precondition, d)]
  index = 0
  for d in shuffle(devices):
    v = group[index % len(group)]
    v2c[v] = "id = " + d.id
for v in group1 + group2:
  p = DeploymentPlan(v.id)
  p.version = v
  p.condition = "or". join(v2c[v])
```

The script divides the available versions into two groups according to their relation to the temperature sensor or the LED matrix, respectively. For each group, it finds all devices that satisfy the common pre-condition of this group. Next, for each device it assigns a version from this group in a circular order, and generates a device-based condition. Finally, it creates a deployment plan for each version, and the condition of this plan is a conjunction of all device-based conditions that we created for the corresponding version.

In all the experiments in the current implementation, the time frame between the moment when a deployment plan is first generated and the moment when an Edge device starts running new software ranges from 30 seconds to two minutes. This primarily depends on the size of a Docker image to be downloaded and launched, which remained relatively small given the simplified setup of the experiments. In practise, downloadable software components are expected to be somewhat larger. Nevertheless, we still expect the operation of the proposed system fulfil the agile requirements of DevOps practices.

6. Conclusion and Future Work

With this paper, we addressed the automated, diversity-oriented fleet management in large-scale IoT deployments through a model-based approach. Based on a software diversity model, the proposed approach allows dynamically managing containerised software components on Edge devices in IoT systems ranging from local deployments to large-scale, geographically distributed ecosystems. Characterised by an increasing degree of software diversity, such systems are required to be managed in a reliable and automated manner so as to achieve economies of scale. As a first step towards the validation of the approach, the paper described a simple, yet realistic smart IoT scenario, where software components were pushed to Edge devices based on a diversity-oriented model continuously synchronised at run-time.

The proposed approach assumes that target Edge devices are sufficiently capable to run a container engine and communicate with the Cloud. However, another common source of diversity comes from 'leaf' IoT devices, which are typically equipped with micro-controllers, run simple firmware written in C, and only communicate with a central Edge hub using a wireless channel. Admittedly, there are many situations, when IoT vendors might also require remotely updating such device firmware in an agile manner. In this light, a primary direction for future work to extend the current approach with our IoT deployment tool named GeneSIS [10], which supports the 'last-mile deployment' from Edge devices to IoT devices. The idea is to deploy a GeneSIS engine as a container on an Edge device; the engine will then interact with the coupled IoT devices and push required updates using available communication channels. By extending the proposed meta-model with relevant concepts, it will be possible to model IoT devices and diversification and apply these modifications to the running system. As a result, we expect to provide a holistic fleet management solution covering both Edge and IoT devices.

Acknowledgments

This work is supported by the H2020 programme under the grant agreement #780351 (ENACT).

References

[1] R. Dautov, S. Distefano, and R. Buyya, "Hierarchical data fusion for Smart Healthcare," *Journal of Big Data*, vol. 6, no. 1, p. 19, 2019.

[2] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, p. 16, 2015.

[3] P. P. Ray, "A survey of IoT cloud platforms," *Future Computing and Informatics Journal*, vol. 1, no. 1-2, pp. 35–46, 2016.

[4] T. Pflanzner and A. Kertész, "A survey of IoT cloud providers," in *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 2016, pp. 730–735.

[5] R. Dautov, S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "Pushing Intelligence to the Edge with a Stream Processing Architecture," in *2017 IEEE International Conference on Internet of Things (iThings)*. IEEE, 2017, pp. 792–799.

[6] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, and A. Solberg, "CloudMF: Model-Driven Management of Multi-Cloud Applications," *ACM Transactions on Internet Technology*, vol. 18, no. 2, p. 16, 2018.

[7] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, and F. Leymann, "OpenTOSCA for IoT: Automating the Deployment of IoT Applications Based on the Mosquitto Message Broker," in *Proceedings of the 6th International Conference on the Internet of Things*. ACM, 2016, pp. 181–182.

[8] P. H. Nguyen, N. Ferry, G. Erdogan, H. Song, S. Lavirotte, J.-Y. Tigli, and A. Solberg, "Advances in deployment and orchestration approaches for iot - a systematic review," in *IEEE International Congress On Internet of Things (ICIOT)*. IEEE, 2019.

[9] N. Bencomo, R. B. France, B. H. Cheng, and U. Aßmann, *Models@run.time: Foundations, Applications, and Roadmaps*. Springer, 2014, vol. 8378.

[10] N. Ferry, P. Nguyen, H. Song, P.-E. Novac, S. Lavirotte, J.-Y. Tigli, and A. Solberg, "GeneSIS: Continuous Orchestration and Deployment of Smart IoT Systems," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 870–875.