Future Generation Computer Systems 113 (2020) 607–619

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

The Elastic Internet of Things - A platform for self-integrating and self-adaptive IoT-systems with support for embedded adaptive hardware

Alwyn Burger [*], Christopher Cichiwskyj, Stephan Schmeißer, Gregor Schiele

University of Duisburg–Essen, Bismarckstr. 90, Duisburg, Germany

ARTICLE INFO

Article history:
Received 3 February 2020
Received in revised form 29 May 2020
Accepted 13 July 2020
Available online 15 July 2020

MSC:
68N99
68M14
68T05

Keywords:
Internet of Things
Cloud computing
Reconfigurable computing
Self-adaptive systems

ABSTRACT

FPGAs have a huge potential to be used as a type of adaptive hardware in IoT applications, that require self-integration and self-adaptation. While widely accepted and used in cloud and edge applications, they are not commonly used in embedded devices. This is due to the highly complex development process to be able to include adaptive hardware and make it available to all layers of an IoT system. For this reason, we present in this work the *Elastic IoT platform*, a framework to facilitate development and deployment of distributed IoT applications, that take advantage of adaptive embedded devices. It uses a resource-oriented approach to abstract available data and processing capabilities to make them available as-a-Service. We present an application example to illustrate the convenience of using the system services provided by the Elastic IoT platform, simplifying the development of a truly self-integrating and self-adaptive IoT system.

1. Introduction

Adaptive hardware allows computing devices to change their physical structure in such a way that they can instantiate arbitrary digital circuits at runtime. These circuits form an optimised hardware solution that can solve a corresponding application task as efficient as possible, while retaining the capability to change its structure at any given time. Field Programmable Gate Arrays (FPGA) are a well known and established type of adaptive hardware that can provide this feature. They are used both as a standalone computing solution (e.g., to optimise Ethernet layer 2 switching [1]) as well as in combination with a conventional CPU as co-processors (e.g., in data centres [2,3]).

However, despite its huge potential, adaptive hardware has so far not found large-scale adoption in the Internet of Things (IoT). The IoT is a complex and highly dynamic combination of interconnected embedded (sensor/actuator) devices, edge servers and cloud services. Devices regularly join and leave the system, physical events trigger necessary system responses and user needs change continuously. This requires IoT systems to self-adapt themselves continuously to previously unknown and ever-changing situations. Adaptive hardware can provide this type of dynamic self-adaptation, creating systems where the devices can deploy new hardware-accelerated components as required. Unneeded hardware components can be torn down to save energy, or to allow a device to deploy a different – more useful – hardware functionality.

This is especially useful to empower embedded IoT devices to provide improved device-local capabilities to an adaptive IoT System, e.g., allowing to move tasks like neural networks [4,5] dynamically to the actual data sources instead of offloading them to the edge or the cloud. The resulting system can be more efficient, secure and robust. While in the past FPGAs were expensive and power inefficient, this is no longer the case. Newer generation FPGAs, especially embedded FPGAs such as the Xilinx Spartan 7 series, are cheap and power efficient, making them attractive for embedded and battery powered devices. Nevertheless, to the best of our knowledge, there are no existing platforms to develop full-fledged self-adaptive IoT systems with support for embedded adaptive hardware based on FPGAs.

In this paper we present such a platform, which we call the **The Elastic IoT platform**. It builds on and extends our previous work on providing development support for *individual* self-adaptive embedded IoT devices, the so-called Elastic Node Platform [6]. The Elastic Node Platform offers a standard hardware

[*] Corresponding author.
E-mail addresses: alwyn.burger@uni-due.de (A. Burger), christopher.cichiwskyj@uni-due.de (C. Cichiwskyj), stephan.schmeisser@uni-due.de (S. Schmeißer), gregor.schiele@uni-due.de (G. Schiele).

https://doi.org/10.1016/j.future.2020.07.035
0167-739X/© 2020 Elsevier B.V. All rights reserved.

design that combines a microcontroller (MCU) with an embedded, low-power FPGA. This allows a developer to experiment with adaptive devices that combine these fundamentally different processing units, increasing the device's performance through FPGA-based hardware accelerators when needed while also benefiting from the power-efficiency of the MCU. The Elastic Node Platform also provides a lightweight system software that simplifies controlling the embedded FPGA and instantiating hardware components on it. Based on this previous approach, the work presented in this paper provides everything that is needed to (1) facilitate the development *and* deployment of distributed IoT applications spanning embedded devices, the edge and the cloud, as well as to (2) take advantage of adaptive embedded hardware in the form of the Elastic Node to create truly self-adaptive IoT systems.

In the remainder of this paper we first clarify our usage scenario and the requirements that our IoT platform must fulfil in Section 2. We then present the design of our platform in Section 3 before showing in Section 4 how it can be used to implement a use case. We discuss related work in Section 5 before finishing with a short conclusion and outlook in Section 6.

2. Adaptive hardware in self-integrating and self-adapting systems

Integrating embedded devices with adaptive hardware into IoT systems is non-trivial. The improvements it offers in computational power over traditional MCU-based devices create new opportunities, but can complicate the development process of the system. To better understand this, consider an example of such a system and an application that one may want to create for it.

2.1. Scenario

Jessie is a developer who wants to program a swarm of drones to explore a city and create a shared map autonomously. The drones are battery powered, contain adaptive hardware and are able to communicate with each other wirelessly. They can also communicate with a remote (edge) gateway that is connected to the Internet and thus can access cloud services. As the drones move around, their connectivity with each other may be interrupted. They may also run out of battery power and have to land to recharge. Therefore, they should be able to join and leave the swarm whenever necessary, both in terms of communication and computation. A newly joined drone should self-adapt to fulfil an appropriate role in the system.

Jessie's city exploration application includes a variety of complex tasks that need to be executed efficiently. As the first task, the drones should be coordinated to explore the city in an optimised way, e.g., using deep learning. Second, once a drone has reached its dedicated position, it can use an integrated camera and a Light Detection and Ranging (LIDAR) sensor to create a set of images and a point cloud of its environment. Third, using this data, the system can calculate a local map and integrate it into a shared map of the city. Jessie wants to get updates on the drones' progress and retrieve the map on her smart phone.

Providing adaptive hardware in the drones enables this system to self-adapt in new ways. For example, normally, planning the drones exploration would be done by a machine learning agent in the cloud. If the drones lose connectivity, they can use the adaptive hardware on one of them to create a new plan locally. In addition, depending on the quality of the communication to the edge gateway, the drones may switch between sending the raw images and point clouds to the edge or processing them locally.

Developing such a system is complicated because it requires to integrate functionality from very different domains, namely FPGA development, embedded systems and edge/cloud service programming that all use very different tools, frameworks and programming languages. In addition, the developer needs to take care of a large number of possible system deployments and switch between them dynamically whenever a change occurs. This combination of devices joining and leaving, and changing their deployed hardware accelerators represents the *continuous change* and *self-adaptivity* that we aim to address.

2.2. Requirements

To support the development of this type of IoT system, we have identified a set of core requirements.

Support for adaptive hardware. The system needs to provide full support for using embedded adaptive hardware as part of an IoT application. This includes the ability to deploy functionality to the hardware at runtime, switch between functionalities (e.g., reconfigure to change the active hardware accelerator from a neural network to encryption), as well as offering accelerators to other devices.

Support for changing availability. The system needs to be able to detect the current state of the system: which adaptive hardware functionality can be used, which devices and services are available, and how they can be reached.

Support for system updates. The system needs to be able to update and redeploy all parts of the system dynamically, including device functionality and edge/cloud services. This should include system services as well as device functionality.

Multi-level programming abstractions. Since IoT applications are very diverse, developers need different levels of programming abstractions. This includes high-level abstractions that are fully transparent to technical details of the hardware and different communication protocols. In addition, lower level abstractions are needed to access embedded functionality efficiently.

Support for constrained devices. Lastly, all of the above mentioned features have to be implemented light-weight enough so they can be deployed on very resource constrained MCUs (e.g., 8-bit AVR devices or the ARM Cortex M0 from Microchip).

Existing systems do not address these requirements (see our related work discussion in Section 5). Therefore, we have designed a new full-stack approach to adaptive systems for the IoT called the *Elastic IoT platform*.

3. Elastic IoT platform

In this section we introduce our approach, the *Elastic IoT platform*. We first give a brief overview and design rationale of our system before describing its different parts in more detail. Overall, our system can be seen as a *distributed IoT operating system* which provides assistance throughout the life-cycle of adaptive IoT applications:

- **at development time:** by specifying distributed IoT applications as a collection of interdependent software and hardware components that are executed on embedded devices, edge and cloud servers as well as end user clients.
- **at startup time:** by deploying and configuring such applications in an optimised way depending on currently available resources.
- **at runtime:** by detecting relevant changes and allowing the application to adapt to them as needed, e.g., by redeploying software components, switching between hardware components, or starting new software components.

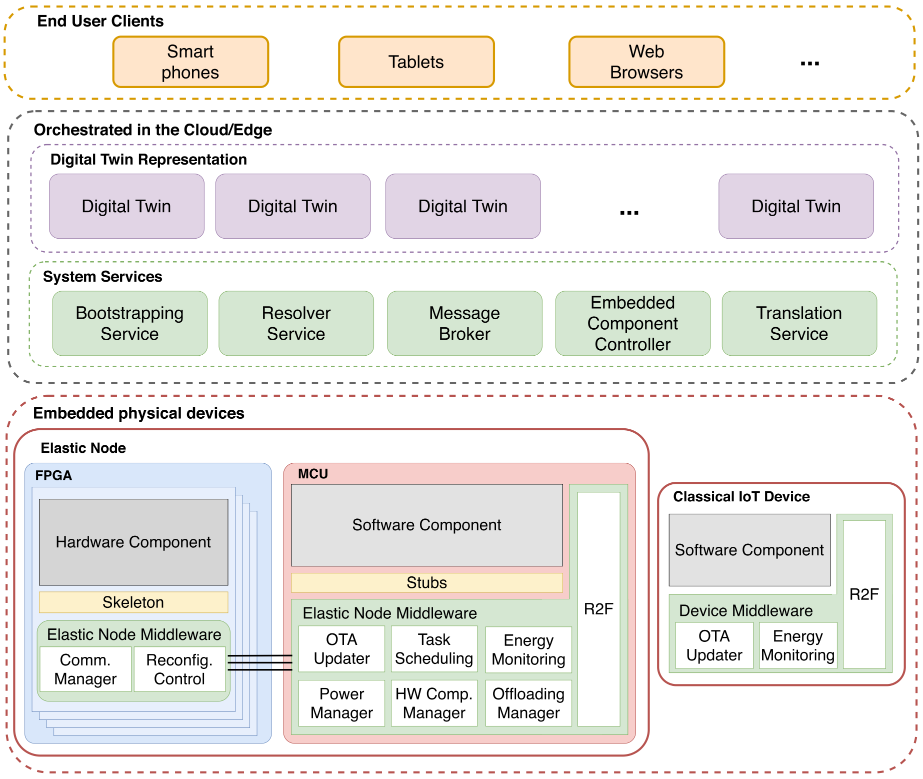


Fig. 1. Overview of the Elastic IoT platform.

The architecture of our system is shown in Fig. 1. As can be seen, we distinguish between three top level subsystems: (1) end user clients, (2) cloud/edge services, and (3) embedded IoT devices.

End user clients are used to access IoT applications that are executed in our system, e.g., through mobile apps or web clients. We decided to not actively manage the software on them to allow end users to have full control over what is installed and executed on their devices. This also is consistent with requirements currently imposed by app stores to ensure the security of users [7].

Software components in the cloud/edge are managed using standard *orchestration techniques*. This provides us with maximum decoupling between the system's various hardware and software, and the application itself. A developer can work on one part of the system without affecting the performance or the uptime of another. We can also handle continuous change by letting Kubernetes deploy new or altered services as they are introduced to the system. Whether a new device or a brand new functionality is added, it is simply handled by spawning the relevant microservice. The ability to use the cloud/edge is clearly important for IoT applications. Nevertheless, enabling this is not our focus since there are a multitude of existing approaches for it. Therefore, instead of developing our own, proprietary solution, we decided to design our system such that we can use existing orchestration technologies for automated deployment and management of software in the cloud/edge. Currently, we are using microservices and Kubernetes [8] for this. Using Kubernetes implicitly provides support for handling *changing availability* (and partially system updates) of any services deployed by it.

We distinguish between two layers of software components that are deployed in the cloud/edge. The first layer is a set of *digital twins*, a digital representation and abstraction of both physical objects and software components of our platform. It offers the primary interface of our system to end user clients and provides the *high-level programming abstractions* mentioned in our requirements. The digital twin abstraction eases the task of application developers, since they only need to interact with abstracted functionality. They can also develop their own, application specific digital twins and deploy them. The second layer is a set of *system services* that provide core functionalities for our system as well as reduce development effort by providing commonly needed features for new, application specific digital twins. One example for a system service is the *embedded component controller*, which is responsible for putting hardware and software components onto embedded devices.

Embedded devices (specifically *constrained devices* with limited available resources) form the third main subsystem of our approach. We support both (1) IoT devices that include an FPGA (so far our own *adaptive hardware* platform the *Elastic Node*) as well as (2) classical IoT devices without adaptive hardware — even tiny basic battery-powered devices with very limited resources. In addition to the system services in the cloud/edge, we also provide a number of services on the embedded devices, e.g., for over-the-air deployment of new software components and for managing the local FPGA and deploying hardware components on it (effectively supporting device-side *system updates*). These services are encompassed into the *Elastic Node Middleware*, which simplifies access to them. Its primary objective is to fulfil our requirement for the lower level part of the *Multi-level programming abstractions*, simplifying the developer's access to each device's capabilities. It allows developers e.g. to register regular updates for a resource representing a sensor value.

Regarding communication, our system is fully resource-oriented. All managed components in our system, i.e. digital twins, system services, as well as embedded devices and their functionalities are modelled as resources and made available to other system parts via uniform resource identifiers (URIs). This includes physical resources such as access to sensor values, interfacing with system software (e.g., accessing the resolver service), and more high-level functionality like requesting additional hardware accelerators to be deployed. As a result, we abstract away how to physically communicate with a specific device or software component. A user or service simply accesses a resource and this is mapped to corresponding messages and protocols by our system services. As an example, all interactions with digital twins and system services are mapped to MQTT message exchanges. Interactions with embedded devices are mapped to protocols supported by the devices.

Our system follows the Observer/Controller Paradigm presented in [9], more specifically the multi-level variant of its architecture which best reflects the multi-faceted approach of the Elastic IoT platform. Each physical device is a system under observation and control (SuOC), where a (distributed) observer is one or more digital twins. These monitor and abstract each device's parameters (e.g., battery level or FPGA state). Groups of digital twins can in return again be a SuOC to other digital twins. Details of this architecture will be further discussed in Section 3.3.1. On the device, the controller mechanism is largely provided through the Elastic Node Middleware. The user can implement device-local control optimisations that use these mechanisms to control device configuration, power states and offloading behaviour.

On a system-wide level, the bootstrapping service can be seen as a centralised observer. It keeps track of active devices and their functionalities. In addition, if needed by an application, a digital twin can be provided to monitor the state of several/all components in the system. Similarly, a digital twin can be added to act as a system controller.

Additionally, one can model the monitoring and deployment services provided by Kubernetes as its observer/controller. The status of all system services and digital twins deployed as containers are managed implicitly, with failing services or even hosting nodes being replaced without user intervention.

In the following sections we further detail the different parts of our system, starting with the Elastic Node hardware platform in Section 3.1. After that follows the Elastic Node middleware as it pertains to local embedded devices in Section 3.2. Lastly, Section 3.3 presents the orchestration of the digital twins and system services.

3.1. Elastic node hardware platform

The objective of the Elastic Node is to be an experimentation tool for developers creating self-* systems. This requires it to fulfil the application logic involved, as well as report functional and experimental results. Ideally most of a user's development effort should be focussed on the application or experiment. Therefore, the intricacies of instantiating hardware and software components should be handled as transparently as possible.

The platform was originally presented in [6], where more information is available on the hardware design and the APIs provided to developers. Here we will briefly discuss the relevant hardware design rationale, and how it makes it possible for us to fulfil our requirements (see Section 2.2) for creating self-adaptive and self-integrating systems.

Specifically, wanting to support adaptive hardware directed our choice of combining an MCU and an FPGA to form the Elastic Node. The MCU is responsible for software components, while the FPGA can deploy hardware components by reconfiguring. Additional hardware such as a wireless communication chip (essential for supporting system updates) and flash storage are available to improve the device's applicability. We developed several versions of the Elastic Node like the Elastic Node v4 (shown in Fig. 2(a)), which combines a very low energy 8-bit AVR AtMega64 MCU with a Spartan 7 S50. More powerful versions use an ARM-based Cortex M4 or M3 and bigger FPGAs — up to a version with the Spartan 7 S100 (shown in Fig. 2(b)).

MCU. In order to execute software-based system components, a conventional sequential processor such as an MCU is required. System management tasks such as controlling the FPGA and collecting experiment results are further responsibilities of the MCU. Since it offers low power consumption and easy development through common programming languages such as C, it offers a convenient and power-optimised foundation.

FPGA. Deployed hardware components need to be instantiated in the field, which requires reconfigurable hardware such as an FPGA. By using a bit file that describes the configuration of all the tiny building blocks and interconnects within it, the FPGA can create functionally any hardware architecture that fits the available resources. Effectively this allows the device to change its abilities from a basic low-power embedded device to an application-specific high performance processing node.

RAM. Many applications will require additional volatile memory for processing, such as intermediate results or buffering input images. In this case, external RAM chips (either SRAM or DRAM) can be added to the device. These can be deactivated whenever not in use to reduce energy overhead.

Storage. The configurations that the FPGA uses and other system functionality (such as storing experimental results) may require bulk data storage. Therefore, at least one external flash memory chip is on the device for long-term storage.

Wireless. Connecting and integrating devices requires communication — whether that is directly between devices or with a central server. In most instances, this is an energy-efficient 802.15.4 device capable of very limited communication, but different communications standards can be supported.

Energy monitoring. As one of the features of the platform is to provide detailed energy monitoring, a number of components are included specifically for that purpose, namely multiple current sensors and a dedicated second monitoring MCU. Each sensor can measure the current flowing through two 'channels', allowing for fine-grained power usage monitoring. The advantage of this is that deployed hardware and software components can be individually tracked and their energy costs evaluated.

3.2. Elastic node middleware

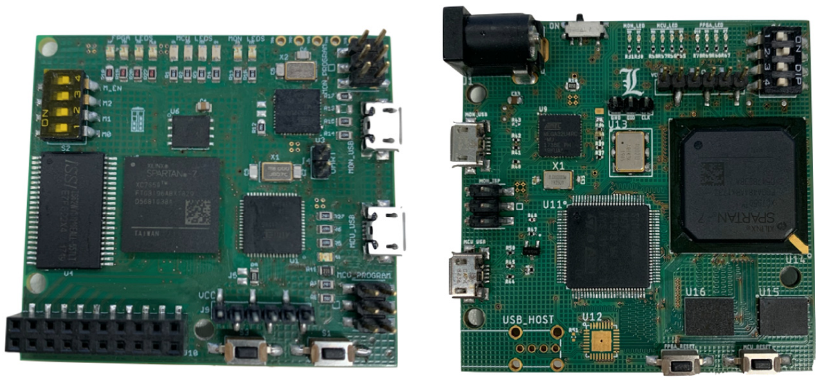
Development of applications including embedded FPGAs has proven to be a very challenging and error-prone process. To facilitate development, an appropriate abstraction for specifics of the hardware is required.

Considering our requirement for *support for constrained devices* and *multi-level programming abstractions* (see Section 2.2), no preexisting embedded operating system or device framework was found, that is able to fit the devices and fulfil our system requirements. For this reason we developed a lightweight set of libraries and system abstractions, called the Elastic Node Middleware, that is specifically targeted at bare-metal environments.

To truly simplify the development for both the hardware component on the FPGA and the software component on the MCU, an abstraction layer has to be provided for both MCU and FPGA. For this reason the middleware is split into parts for the MCU-side and parts for the FPGA. An overview can be seen in Fig. 1 on the lower left side.

3.2.1. MCU services

On the MCU side the middleware provides a set of services that provide control over several aspects of the FPGA and offer its capabilities to the other parts of the Elastic IoT system. We describe these in the following.



(a) (b)

Fig. 2. Two variations of the Elastic Node Hardware Platform: (a) AVR-based and (b) ARM-based.

Remote Resource Framework (R2F). To abstract away from specifics of underlying communication protocols, the Remote Resource Framework (R2F) service incorporates the capabilities to communicate to other devices by providing a resource-oriented interaction scheme. R2F can provide a self-description of the Elastic Node's currently available capabilities, e.g., what hardware components are available on it. Additionally, through the R2F, remote services can access data, control the device and deploy components on it. For this the R2F supports two major application paradigms for developers to use.

In the first paradigm, we assume that a developer is not interested in specifically writing a custom software component for the Elastic Node and instead wants to purely control it remotely on a resource-oriented abstraction. In this case the R2F provides remote access to all services the Elastic Node can offer, e.g., over the power management or what hardware component to execute. This in return means that a programmers does not require to change any of the existing device software.

In the second paradigm, a developer might want to write a custom software component for the MCU-side, e.g., to provide additional services or have more fine-grained control over how to use hardware components. Through the R2F service, the developer can deploy his or her software component onto the Elastic Node. The software component can either use the middleware services directly to gain access to hardware components, or, to further simplify the development, use a stub/skeleton approach. The stubs/skeletons lie between the hardware/software component and the middleware and take care of marshalling data correctly, before exchanging it. Once the software component is deployed, any new services it might be able to also offer to others can be described and provided through R2F.

While the first paradigm provides a transparent, high-level abstraction for programmers, that are not interested in the details of a device's implementation, the second paradigm allows programmers to adapt and change the specific implementation of the device's software component to meet his or her requirements. The R2F framework therefore provides a *multi-level programming abstraction* to accommodate programmers across all levels of the system to control and manage embedded devices.

Power manager. An IoT developer often requires control over an embedded device's current power consumption, to be able to adapt to an application's current needs. The Power Manager service provides the resources for switching both the MCU and the FPGA into low-power states or sleep modes and to completely turn off or power up the FPGA.

Hardware component manager. After the FPGA is powered on, the Hardware Component Manager service allows developers to choose which hardware component should be instantiated. It manages the flash memory in which the hardware components are stored as FPGA bit files and abstracts away the underlying hardware interactions required for the hardware component instantiation. The Hardware Component Manager also handles the communication between the MCU and the FPGA. It is used to provide any necessary input data to a hardware component and receives the results accordingly.

This fulfils our requirement for *support for adaptive hardware*, more specifically for controlling and managing adaptive hardware through software locally on the device as well as remotely through the R2F framework.

Offloading manager. When another part of the IoT system requests that a task be offloaded onto an Elastic Node for execution on a hardware component, the Offloading Manager service handles this request. It detects which hardware component is required for the task, and uses the Power Manager to activate the FPGA and the Hardware Component Manager to instantiate the correct hardware component. To do this, this service accesses the resources for each device's available hardware accelerators.

Task scheduling. When multiple IoT system parts offload tasks to the same Elastic Node, the order in which multiple incoming tasks are executed can lead to frequent reconfigurations, which waste time and energy. However, we assume that certain tasks within the IoT application reuse the same (set of) FPGA bit files but with different input data, e.g. querying a neural network with different sensor data. By reordering the execution sequence, we can queue tasks that rely on the same configuration after each other, thus reducing the overall number of reconfigurations. This can potentially reduce the energy and time overhead for executing hardware components on the FPGA. Additionally the scheduler allows developers to define task priorities themselves, e.g. by scheduling local tasks before offloaded remote tasks.

OTA Updater. During the lifetime of an IoT application the set of available hardware or software components might not meet the application's requirements any more. The Over the Air (OTA) Updater service provides resources that allow developers to change the available hardware components for the FPGA as well as exchange the software component on the MCU. This service supports three types of actions:

- Update: A component may be updated, i.e., the component is replaced with a newer version that has the same purpose.
- Replace: A component may be exchanged for a different component that offers a different functionality.

- Add: A previously not available component is added. If there is not enough memory available to add a new component, this is not possible, and instead an existing component needs to be replaced.

Such an OTA update allows us to fulfil our requirement for *support for system updates* of all application components on embedded devices.

Energy monitoring. To support the energy monitoring described in , we offer the Energy Monitoring service. Internally it consists of a system library that is deployed on the Elastic Node's second MCU, that records the energy consumption, and a service component on the main MCU that allows it to exchange data between the main and the second MCU. Due to the decoupling of the energy measurements, the second MCU can be set up to run energy data collection experiments without interfering with the application execution on the main MCU. The results can be accessed through the Energy Monitoring service on the main MCU either to be used locally by the software component or to be provided as data resources via R2F.

3.2.2. FPGA services

As counterpart to the MCU side, the FPGA side of the Middleware offers a thin set of supporting hardware circuits as services to simplify the hardware component development and deployment. While it would be possible to abstract the inner workings of the FPGA completely to developers, we believe that offering a *multi-level programming abstractions* even to the lower levels of a device offers the greatest flexibility.

Communication managing. The Communication Manager service represents the equivalent to the MCU-side Hardware Component Manager service in terms of communication and data exchange. It accepts the incoming data for a hardware component from the MCU and hands it over either directly to the hardware component or to a skeleton component.

Reconfiguration control. While the Hardware Component Manager service allows the MCU to instantiate components, this can also be triggered from within the FPGA with the Reconfiguration Control service. This allows the FPGA to initiate a reconfiguration by itself if required by the hardware component. Through this a hardware component developers can chain the execution of multiple hardware components to execute more complex logic that cannot be instantiated all at the same time.

3.3. System orchestration

The key aspect when designing the orchestration system for the Elastic IoT platform was to provide the maximum amount of flexibility: we strongly believe that only a system that can adapt to changes is sustainable and overcomes the ever-changing requirements of the IoT. To provide the maximum flexibility we took inspiration from microservice architectures and mapped it to our platform: each device (or even just a simple functionality of a device) is seen as a microservice with an accompanying digital representation in the edge or cloud: a *digital twin*. The interaction scheme between all these components internally uses MQTT messages. The message details from creation to delivery is entirely transparent to the user/component as they interact and access resources.

The orchestrated parts of the system are split into two layers, the *digital twins* and the *system services.* Each component in these layers is implemented as a container that is orchestrated by Kubernetes. Using Kubernetes allows for easy deployment as it takes care of the entire lifecycle of each container and its connectivity. Automatic restart of failed containers, auto-scaling and replication for resilience are just a few benefits of using this established technology in cloud and edge computing. In the following we will discuss each layer of the orchestration system in more detail.

3.3.1. Digital twins

The digital twin is the main abstraction of our orchestration system and is reused in various contexts. We can distinguish between two different digital twins: (i) a device twin and (ii) a composite twin. In principle, each twin is a digital abstraction of a part of the platform. It has two distinct interfaces: (i) an internal interface for communication and interaction with the platform, and (ii) an external interface for interacting with users or other software. The internal interface is mandatory and consists of a set of resources (i.e., URIs) that are exposed to the platform and are accessed via MQTT. We provide support for the internal interface though the *twin-interface library*, which integrates your digital twin into our platform. Amongst other things it offers two functions that handle all URI interactions. The function `callURI(EIPMessage)` allows you to access remote URIs, where the *EIPMessage* is a custom datatype that eases accessing URIs. It only requires the URI to be invoked, but can be expanded to contain a multitude of application specific parameters. The second function we provide is the callback `handleURI(EIPMessage)`, which is invoked whenever one of your digital twin's resources is accessed.

The external interface is optional and only necessary for digital twins that should be accessed from outside our platform, e.g. from end user devices. We provide no additional support for developing this interface in our platform. The programmer is free to implement any type of interface that her application requires but we recommend (and use in our own work) RESTful HTTP interfaces since they allow for a descriptive mapping between HTTP URLs and digital twin URIs. They are also easy to implement using existing standard libraries.

Device twin. The device twin is the digital representation of a physical device. It mirrors the state of a device and offers its functionalities by abstracting them as resources. Resources are part of the self-description of a device, and are configured in the twin upon startup (see *Bootstrapping Service* in the next Section).

A digital twin also has the ability to change its own state and therefore the state of the device. Life-cycle management is also incorporated into a device twin, as it is only useful when it has a corresponding physical device. If the device leaves the system, the device twin can terminate itself and be removed by leveraging the dynamic orchestration features of Kubernetes.

As an example, each drone in the scenario from Section 2.1 is represented by its own device twin. Each of the URIs it offers to represent its resources shares a common prefix to identify that drone in a human relatable way (e.g., `EIP://twins/drone1/`). A URI that triggers a repositioning of a drone might be `EIP://twins/drone1/moveToPosition` with parameters denoting a position.

Composite Twin. While having a device twin for each device offers a lot of convenience during application development, the sheer number of devices can render that abstraction overwhelming. In addition, every time a twin becomes available or is removed, the application would need to be notified and adapt itself. To address this we offer another layer of abstraction that we refer to as a *Composite Twin*. The idea behind this twin is to abstract multiple twins and thereby represent a higher level concept instead of a singular device. One such composite twin could combine multiple device twins and abstract them into a digital twin of a room. This twin would offer its own set of resources that are internally mapped to the individual device twins (how that mapping is resolved is explained in the *Resolver* part of the next section). It is even possible to combine multiple composite twins together to create even more abstractions. Composite twins use the same resource-oriented interaction scheme as device twins and the rest of the system and can therefore integrate seamlessly into the platform.

In our drone exploration scenario we can utilise a composite twin that incorporates all active drone device twins, and represents the current exploration strategy (represented by the URI `EIP://cityExploration`). When the optimisation changes and a drone (e.g. `droneX`) has to change its position, this change is initiated by the composite twin by accessing the aforementioned resource `EIP://twins/droneX/moveToPosition` of the corresponding device twin.

Listing 1 shows how this is done, illustrating the support we provide for accessing URIs with the `callURI(EIPMessage)` function available to any digital twin. The *EIPMessage* must be constructed by providing the URI being accessed (in this case `EIP://twins/droneX/moveToPosition`) and the parameters being passed. Parameters are passed as key/value pairs, in this case longitude and latitude. The resulting mapping to an MQTT message is completely transparent to the composite twin.

Listing 1: Accessing droneX URI to trigger repositioning.

```
callURI(new EIPMessage(
    "EIP://twins/droneX/moveToPosition",
    EIPMessage.parameters(
        "long", 6.786714,
        "lat", 51.428735
        )
    )
)
```

3.3.2. System services

The Elastic IoT platform evolves around resources: each component exposes and accesses resources. Different layers of our platform have different requirements and prerequisites for handling resources (i.e. URIs). While the digital twins hosted in the cloud or edge have libraries available for this, the constrained devices often lack the (computational) resources required by these libraries. Therefore, we created a set of system services that implement the features required for our end-to-end resource abstraction. We identified a number of core functionalities and implemented a dedicated system service for each of them (reflecting the microservice oriented design). In the following we will discuss each system service as presented in Fig. 1, providing an example interaction.

Message broker. Internally, the interaction scheme relies on message passing. With potentially thousands of devices and twins we need an intuitive and flexible way to ensure message delivery (preferably in a many-to-many fashion). We have chosen a topic-based publish and subscribe message broker paradigm as means of communication: clients can publish/subscribe to topics and the message broker ensures the delivery of messages to interested clients. This decouples communication partners, which caters well to continuously changing IoT systems that have constant joining and leaving of devices and digital twins. In our current implementation we rely on a centralised MQTT broker, but it can be exchanged for a distributed one. Note that every resource access within the Elastic IoT platform requires multiple message exchanges that are all routed through the message broker. Choosing a future-proof and scalable solution is crucial. As our messages are resource-oriented (i.e., you access a resource from the platform rather than addressing a specific communication partner), we need an efficient mapping from resources to topics. Currently we directly use the URIs as topics, leading to an easy yet efficient mapping due to the hierarchical nature of MQTT topics.

Resolver. Digital twins and resources can represent a plethora of different things, ranging from a single feature to a sophisticated composition of devices. The *Resolver Service* provides a service to resolve resources into either (i) other resources or (ii) specific communication partners. It is automatically queried for every message sent throughout the platform, which means that it requires access to all device descriptions and resource mappings. All this information is stored persistently, e.g., in a linked-data database which the service can query. The resource mapping (resolving one resource into another resource or into a communication partner) is a key component for our resource abstraction. Therefore, we outsourced the responsibility to the resolver service — providing the functionality even for resource constrained devices.

In case of a digital twin communicating with its physical counterpart it is possible that the device is only indirectly capable of communicating with the message broker: the device has to relay its messages through a translation service. Due to the decoupling of hardware and software components (and the reliance on resource-oriented communication), the digital twin is not aware of this, as it only accesses URIs. When accessing the URI of the device, the resolver service would resolve it to be redirected to the translation service. Additionally, it provides a description of how the initial request has to be formatted for the translation service to pass it on to the device.

Continuing our example, to integrate a newly joined drone into the city exploration strategy, the composite twin `EIP://cityExploration` accesses the URI `EIP://twins/droneX/moveToPosition` of the device twin of the drone to move it to the correct position. As shown in Listing 2, the device twin receives this message via the callback function `callURI(EIPMessage)` provided by our twin interface library.

Listing 2: DroneX's twin handling the incoming URI access from the city exploration twin to reposition and triggering the reposition of its physical counterpart.

```
@Override
public void handleURI(EIPMessage uri){
    switch(uri.value){
        case REPOSITION:
            updatePosition(uri.parameters);
            callURI(new EIPMessage(
                "EIP://droneX/moveToPosition",
                uri.parameters
            ));
            break;
    }
}
```

After changing its own local state – and possibly checking if the drone can reach this location – the device twin in turn accesses its drone's reposition resource `EIP://droneX/moveToPosition` and forwards the new location to it. However, since the drone does not support MQTT directly, we need to use the Translation service – detailed in the next section – to communicate with it. To do so, we need to rewrite the URI to route the message through Translation service, i.e.

```
EIP://droneX/moveToPosition <mappedTo>
  EIP://PTS/R2F/droneX/moveToPosition
```

where PTS references the Translation service and R2F references the R2F system service on the drone. This is done transparently by the Resolver service such that the device twin itself is independent of the drone's supported communication protocols.

Translation service. The translation service (*PTS*) is an optional service that is only required if physical devices will be included in the Elastic IoT platform that are not connected to the Internet and therefore cannot directly communicate with the rest of the system. This can be the case if the physical device is only equipped for low-energy personal area network communication (e.g., Bluetooth or IEEE 802.15.4) or uses a proprietary long range technology. In these cases the translation service must have access to a compatible hardware interface capable of using the communication standard of the devices. The task of the translation service is to act as a proxy for the devices towards the system: It relays all messages and, if necessary, reformats the messages to make them appropriate for the used communication protocol. Our current implementation provides a translation service for 802.15.4 to support directly connecting our Elastic Nodes to nearby edge servers. In addition, we offer a HTTP translation service for devices that can be connected to the Internet but are not part of our platform. More translation services can be deployed in parallel as needed.

In our example scenario, when repositioning *droneX*, the PTS receives the resolver's message that *droneX* is reachable through R2F. This message does not contain critical protocol specific information such as the IEEE 802.15.4 address or the payload format. This information must be again queried from the resolver similarly to resolving a twin URI. An example resolver response could be

```
EIP://PTS/R2F/droneX/moveToPosition
  <hasAddress> "XXXXXXXX"
  <hasURI> /moveTo
  <hasMsgType> POST
  <expectsAck> true
  <hasParameters>
     "longitude", "latitude"
     ...
```

which includes the R2F URI that has to be invoked, and whether an acknowledgement is expected. The PTS can then assemble the R2F message and send it to the device using the appropriate communication interface.

Bootstrapping. Another integral service, especially for self-integrating systems, is the *Bootstrapping Service*. It acts as the entry point for every new component joining the system, whether it is a device or a digital twin. To be properly integrated into the platform, the resolver service needs access to the descriptions and URI mappings of the newly joined component. Even though this information can be entered manually before joining, the only sustainable option for a large scale system is self-description. Upon detecting a join, the bootstrapping service requests all necessary information from the new component and updates the resolver service to enable the integration on a functional level. Additionally, the bootstrapping service performs minor maintenance on the platform as it keeps track of what devices and digital twins are currently available and updates the resolver service accordingly, including removing mappings if devices leave. At runtime the Bootstrapping Service acts as an observer observing the validity of mappings. It also partially takes on the role of a controller initiating deletion of invalid mappings. It is not, however, responsible for removing/shutting down orphaned components as this role is handled directly by Kubernetes. Special attention has to be paid when a previously unknown device joins the system when there is no digital twin available to represent it. Even though the device was previously unknown it still provides a self-description to the bootstrapping service. Within this self-description it contains information about the required device twin. Usually this information comes in the shape of a container template that can be deployed by Kubernetes. The bootstrapping service will emit a signal to Kubernetes to deploy this new device twin and provide the identifier of the container, taken from the device's description. Once the container is deployed, it registers with the bootstrapping service, and can be integrated into the existing platform. At this stage the correct resource mappings between device and digital twin can be updated for the resolver service to guarantee communication.

In our example scenario the Bootstrapping service is vital when a drone is rejoining the system. If *droneX* joins again after recharging its batteries, it broadcasts a joining message to the bootstrapping service. Since the drone cannot directly communicate with the rest of the platform, the PTS acts as a proxy and creates a preliminary local mapping for reaching *droneX*. The bootstrapping service acknowledges the join request and accesses the by-contract defined URI `EIP://PTS/R2F/droneX/description`. The drone replies with its description which contains its own URI mappings and the container template of its device twin. The bootstrapper hands these mappings over to the resolver to be persisted and informs Kubernetes about the container to be deployed. After successful deployment, the device twin hands its own mappings to the bootstrapping service, so all mappings can be made available by the resolver.

Embedded Component Controller. The Embedded Component Controller service is the facilitator of over-the-air (OTA) updates for embedded devices. It provides the mechanism for updating the embedded system software (either in full or partially), and in the case of the Elastic Node Platform provides bit files for new hardware components. It can also retrieve these software and bit files from remote locations, given the necessary information, e.g., download links. Offering this as a centralised service has the advantage that it hides the implementation details from the digital twins and thus the end user clients. To offer the ability to apply OTA updates, device twins and devices can expose URIs `device(Twin)URI/OTA` and `device(Twin)URI/addAccelerator`. The embedded component controller handles calls to these URIs to avoid code duplication, as these updates usually work the same per device class. Therefore the functionality only needs to be implemented once in the embedded component controller, seamlessly integrating that functionality into the Elastic IoT platform for all devices. Note that actually executing the OTA update (which might be implemented differently for different devices) is the task of the Elastic Node Middleware, which cooperates with the Embedded Component Controller.

4. Use case: Self-aware devices

To investigate how our system can be applied to an application scenario, we consider an example use case. This scenario is based on a device-focussed application. The developer is creating a system of self-aware [10] drones that collaborate, similarly to the system mentioned in Section 2. Each drone may have a camera and a short range Light Detection and Ranging (LIDAR) sensor, and must choose between using these sensors and/or processing. This creates an interesting system that can be used to evaluate a variety of self-* algorithms. This scenario deals particularly with continuous change as not only the active devices change, but each device can change its computational abilities and which type of processing it can perform. Additionally, drones need to be able to leave and rejoin the system freely as they require regular battery recharging.

4.1. System description

Each drone has the sensors mentioned above, as well as a communication module and an FPGA to deploy hardware components to. They are also battery-powered and must therefore finely manage their energy consumption, and balance their sensing and processing with the high energy cost of flying or hovering. Their self-awareness is closely linked to this, as it allows them to not only monitor their current level, but also act based on expected lifetime.

An overview of the system is shown in Fig. 3, which shows a number of drones collaborating in Stage 1 to observe. This involves some drones sensing and others processing, maximising the performance of the hardware accelerator by batching input data. Next the drones report their collected data, updated state, and experimental results such as energy costs of the previous stage.

This is used by a *machine learning component* that controls the decision-making process of the drones. Its objective is to teach the devices how to self-adapt based on their self-awareness. It defines the *behaviour* for each of them, controlling whether they are sensing or processing. Similar to deep reinforcement learning [11–13], this component controls how drones offload to each other. It also controls device state — in this case which sensor a drone should utilise and which hardware accelerator it should instantiate. For example, it may tell one drone to activate its LIDAR sensor to scan a targeted building, while telling another to prepare the hardware accelerator required to process the resulting point cloud.

4.2. Implementation with the elastic IoT platform

In the Elastic IoT platform, each drone is modelled as a device twin, and gets deployed with a software component and a set of hardware components — one for performing feature extraction on an image and one for creating a point cloud from the LIDAR data. This allows them to switch between the different processing options at runtime.

We define the machine learning component as a composite digital twin. This makes it considerably easier to initially develop or later alter such a component. Deploying it as a managed container in the edge or cloud makes it easier to update during the development process or even to compare different versions of it during an experiment.

Each device's current state (battery life, available hardware component, etc.) is sent to its device twin using R2F. The machine learning digital twin creates a controller [9] that would interact with each device twin, and uses that information as input to learn the optimal device behaviour. The *Power Manager* enacts this behaviour, changing which drones should be fully or partially idle. One example of this is a drone focussing on LIDAR sensing being told to completely disable its FPGA to save power. Accessing this information through the device twin is considerably easier than having to query each device individually, especially as devices may join or leave the system dynamically. Instead, a composite device twin that effectively describes the overall system state provides an overview of all the drones.

Our *Hardware Component Manager* simplifies the task of updating the configured behaviour of each drone by ensuring that the required hardware components are present, and that the ones no longer required are torn down to save energy. For example, when a LIDAR sensing drone becomes a processing drone it needs access to a new hardware accelerator. Without our system, this may require the user to manually upload the correct bit files to each drone, and to develop the functionality on the device for controlling the FPGA.

As drones leave and rejoin to recharge their batteries, it creates a continuous need for self-integration. Apart from the functionality provided and required by this drone, it also needs to join the communications network and register its available resources. Using the Elastic IoT Platform, this becomes as simple as defining the required URIs on the device. These are then registered through the Elastic Node Middleware with the *Bootstrapping system service*, making them reachable and resolvable by other devices (and the user).

Switching between different stages as well as allowing the user to access the system is implemented as a second composite digital twin, which we call the *app twin*. For example, when a certain system state occurs (e.g., all of the devices are dead), the app twin can start the next leg by charging and resetting all the drones through their device twins. This process is greatly simplified compared to requiring the user to manually do it, which may be necessarily if that level of device control is not available. Through the app twin, the user can also access the assembled city map. The twin assembles the collected map data and provides an HTTP interface to render a 3D map on a web page that the user can access with her mobile device.

In addition to this, we can provide yet another composite twin for developers to monitor the system state. To do so, this twin collects the system state over time by regularly querying the device twins. This allows the developer to study how their self-* algorithm is performing on a system-wide level, without interfering with the devices' operation or the application shown to the users.

While the abstractions provided by the Elastic IoT platform make it much more convenient for developers to interact with devices and system functionalities, it is important to evaluate the overhead introduced by them, e.g. for resolving URIs as well as for additional messages required for communicating with different system services. To do so, we conducted an experiment with an End User Client accessing the `EIP://twins/droneX/moveToPosition` URI as described in Section 3.3. We measured the time from the initial URI access to the messages being relayed to the drone by the PTS. Note that this includes creating all necessary 802.15.4 messages but omits the transmission time to the drone using 802.15.4. This later time is independent of using our approach.

The overhead and margin of error involved in our experiment is shown in Fig. 4, where the number of parameters in a message are varied for different queries. This demonstrates that the introduced overhead is very low, remaining under 240 ms for up to 100 parameters in the URI access. The bumps where the overhead suddenly increases can be attributed to the fragmentation performed by the PTS for 802.15.4. Due to the severe message size limitations of IEEE 802.15.4, roughly every 50 additional floating point parameters require an extra message to be created.

Accessing such a functionality without using the Elastic IoT platform would require not only hard-coding the addresses of the gateway and the drone, but also manually encoding and fragmenting these messages. Instead, our system allows the user to interact only with the high-level URI, encoding all the parameters required into the JSON format. The internal details of addressing and message encoding are done transparently by the platform.

5. Related work

There are a number of related approaches, which we discuss briefly.

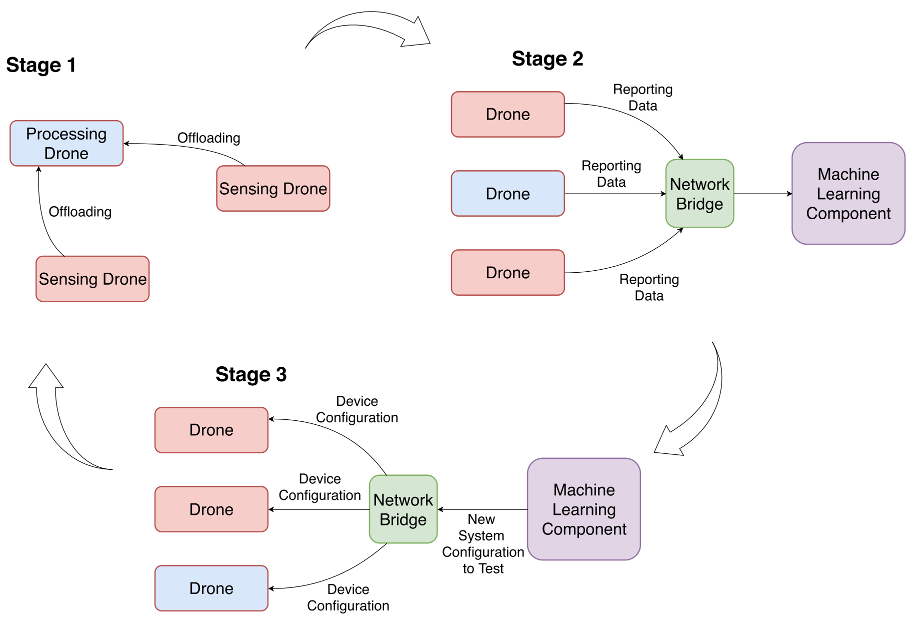


Fig. 3. Self-aware devices use case system.

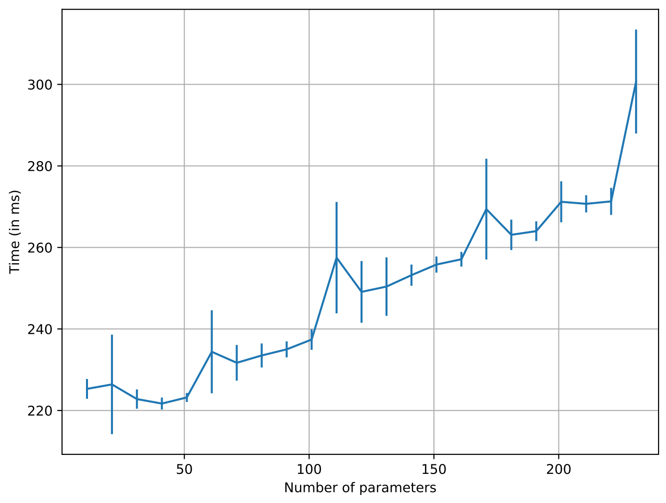


Fig. 4. Overhead introduced by Elastic IoT abstractions.

5.1. Adaptive hardware platforms

Multiple projects have addressed reconfigurable computing platforms for the IoT, but none address all of the requirements we set in Section 2.2. Most of them target a larger class of device, with too high power consumption for our battery-powered deployments and experiments. Among these are some smart camera systems [14] and applications targeted at industrial use cases [15]. Even MCU-based ones such as CaRDIN [16] rely on a full IP stack, while most older systems such as P-HAL [17] and Egret [18] require a full Linux OS. Although this details another valid approach, we aim to also support very power-efficient devices where this is either impossible or simply causes too high overhead.

As the Elastic IoT relies on effectively abstracting the complexities of MCU-FPGA heterogeneity, projects like Cookie [19], HaLoMote [20] and Sentiof [21] too heavily rely on developer effort (i.e., HW/SW co-design). Similarly, multi-core embedded systems projects such as PULP [22] can be very difficult to develop for. Another alternative is putting the MCU onto the FPGA as a softcore (e.g., HiReCookie [23]). This offers multiple advantages such as a smaller physical footprint, but requires larger FPGAs as some of its resources are dedicated to the MCU. The slot-based, modular abstraction scheme used in such projects create larger FPGA resource overhead than our minimal approach, but also creates simpler heterogeneous interaction schemes between the FPGA and primary OS.

5.2. Software support for adaptive hardware platforms

To support developers to use and deploy hardware components on the FPGA a suitable software abstraction has to be offered. This abstraction usually provides an application programming interface to use and control the different aspects of the FPGA, such as data exchange or reconfiguration.

Several frameworks have been developed for server- and PC-grade systems that use FPGAs as co-processors [24–29]. More recent research also includes FPGA-based co-processors for mobile-grade CPUs [30], that are generally used in personal mobile devices, such as smartphones. All of these approaches have in common that a underlying system software or operating system (OS) offers a thread based programming abstraction to interact with the FPGA. OS drivers or library implementations, that control the FPGA are hidden behind this threading abstraction (e.g., ReconOS [31]). Many approaches such as the ARTICo^3 system [32] from the group that developed HiReCookie is aimed at the Zynq 7000 series for System-On-Chip (SoC) devices that feature an ARM-based Linux OS. The underlying architecture for such a framework proves to introduce too large of an overhead in code size, memory consumption and performance to be viable for single core CPUs as used in 8-bit MCUs.

5.3. Software support for adaptive systems

Finally, we compare our approach to existing approaches for system support for adaptive systems in the IoT and related areas. This has been an active research topic, especially in the field of Pervasive Computing, for the better part of 30 years and has addressed many challenges [33]. The result is a set of common denominators, specifically for the supported interaction paradigm. The most common interaction paradigm is based on service oriented architectures (SOA), in which devices and sometimes their data are abstracted as sets of services [33–35]. More recent SOA systems for adaptive systems include: the Hydra Project [36], specifically its LinkSmart Middleware [37], the Global Sensor Network [38], the Middleware from the SOFIA project [39], and SOCRADES [40], etc. All of these approaches rely on the fact that the device supports an OS, either a full fledged Linux or a more stripped-down version like TinyOS. They do not support adaptive hardware, as most of them have been designed before adaptive hardware was widely available and accepted. Other approaches [33] rely on message passing, tuple-space information sharing or publish and subscribe interaction patterns, e.g., [41–44]. They all share the same lack of support for adaptive hardware like the SOA based ones. An interesting approach is presented with the XWARE system [45], which is a framework for integrating different systems. We can use XWARE to integrate our Elastic IoT platform with other system platforms. The *CHARIOT* system [46] proposes a common test environment for self-adaptive systems. It defines a distributed middleware layer that could be deployed as system services in our orchestrated cloud/edge subsystem. Similar to our approach, CHARIOT is designed to provide maximal flexibility and relies on self-description concepts. While in theory, this would enable the system to describe and use adaptive hardware, this is not considered explicitly, yet.

6. Conclusion & outlook

We have presented a platform for developing and deploying IoT systems that utilise adaptive hardware for handling continuous change. It provides flexibility and convenience both on the embedded hardware layer as well as in the cloud and edge. Through a resource-oriented interaction scheme, either data (e.g., a sensor) or computational power (e.g., an accelerator deployed to a local FPGA) can be provided as-a-service.

This is done using a number of abstraction layers that can be used for developers on different system levels. This includes system services that provide access to universally required system functionality and a set of digital twins that bridge the gap between the system and end user clients. They can be used to abstract either the resources provided by a specific device or more complex aggregated concepts such as a set of spatially related devices. In addition, we provide low level programming access on individual embedded devices through our Elastic Node Middleware. With respect to self-integration and self-adaptation, new devices and services can be added dynamically to the system and are integrated automatically. For cloud and edge resources, Kubernetes ensures that failing servers are detected and new software versions can be deployed at runtime. For embedded devices, the Elastic IoT Platform provides its own services to accomplish this. We also provide support for dynamically adapting the hardware configurations of devices, e.g., by deploying new hardware accelerators on adaptive embedded devices.

In the future we plan to interconnect multiple instances of our system together. Organisations will likely opt to isolate their own systems from others, but may choose to expose certain resources. This requires support for resource access management [47] and routing communication to allow systems across the world to collaborate.

Our other primary focus is to improve support for a larger variety of applications and devices. This includes additional communication protocols — both local and WAN radio standards such as Narrowband IoT or LoRaWAN. This requires new variants of the Translation Service that integrates the communication with these devices.

As our hardware and embedded software are already available as open source, we plan to release the cloud/edge parts as well. We hope this will allow other research groups to use our approach to develop and deploy their own experiments and applications. This will likely introduce new system services, extending the applicability of our system even further.

CRediT authorship contribution statement

Alwyn Burger: Conceptualization, Methodology, Software, Investigation, Writing - original draft, Writing - review & editing. **Christopher Cichiwskyj:** Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing. **Stephan Schmeißer:** Conceptualization, Methodology, Software, Formal analysis, Investigation, Writing - original draft, Writing - review & editing. **Gregor Schiele:** Supervision, Conceptualization, Methodology, Writing - original draft, Writing - review & editing, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the KI-Sprung LUTNet project (project number 16ES1125) as well as the KI-LiveS project, Germany (project number 01IS19068A).

References

[1] M. Rezaee, T.H. Szymanski, Demonstration of an FPGA controller for guaranteed-rate optical packet switching, in: 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM, 2015, pp. 1139–1140, http://dx.doi.org/10.1109/INM.2015.7140448.

[2] J. Weerasinghe, F. Abel, C. Hagleitner, A. Herkersdorf, Enabling FPGAs in hyperscale data centers, in: 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops, UIC-ATC-ScalCom, 2015, pp. 1078–1086, http://dx.doi.org/10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.199.

[3] N. Tarafdar, T. Lin, E. Fukuda, H. Bannazadeh, A. Leon-Garcia, P. Chow, Enabling flexible network FPGA clusters in a heterogeneous cloud data center, in: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 237–246, http://dx.doi.org/10.1145/3020078.3021742.

[4] A. Burger, C. Qian, G. Schiele, D. Helms, An embedded CNN implementation for on-device ECG analysis, in: IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), 2020.

[5] A. Burger, G. Schiele, Demo abstract: Deep learning on an elastic node for the internet of things, in: 2018 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops, 2018, pp. 424–426, http://dx.doi.org/10.1109/PERCOMW.2018.8480160.

[6] G. Schiele, A. Burger, C. Cichiwskyj, The elastic node: An experimentation platform for hardware accelerator research in the internet of things, in: 2019 IEEE International Conference on Autonomic Computing, ICAC 2019, IEEE, 2019, pp. 84–94, http://dx.doi.org/10.1109/ICAC.2019.00020.

[7] Apple Inc., App store review guidelines, 2020, URL https://developer.apple.com/app-store/review/guidelines/. (Accessed 3 February 2020).

[8] The Kubernetes Authors, Production-grade container orchestration, 2020, URL https://kubernetes.io/. (Accessed 3 February 2020),

[9] S. Tomforde, H. Prothmann, J. Branke, J. Hähner, M. Mnif, C. Müller-Schloer, U. Richter, H. Schmeck, Observation and control of organic systems, in: Organic Computing — a Paradigm Shift for Complex Systems, 2011, http://dx.doi.org/10.1007/978-3-0348-0130-0_21.

[10] P.R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, X. Yao, A survey of self-awareness and its application in computing systems, in: International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW, 2011, pp. 102–107, http://dx.doi.org/10.1109/SASOW.2011.25.

[11] Z. Xu, Y. Wang, J. Tang, J. Wang, M.C. Gursoy, A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs, in: IEEE International Conference on Communications, IEEE, 2017, pp. 1–6, http://dx.doi.org/10.1109/ICC.2017.7997286.

[12] L. Huang, S. Bi, Y.-J.A. Zhang, Deep reinforcement learning for online offloading in wireless powered mobile-edge computing networks 518060, 2018, pp. 1–24, URL arXiv:1808.01977.

[13] J. Li, H. Gao, T. Lv, Y. Lu, Deep reinforcement learning based computation offloading and resource allocation for MEC, in: IEEE Wireless Communications and Networking Conference, WCNC 2018-April, 2018, pp. 1–6, http://dx.doi.org/10.1109/WCNC.2018.8377343.

[14] Z. Guettatfi, P. Hubner, M. Platzner, B. Rinner, Computational self-awareness as design approach for visual sensor nodes, in: 12th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip, ReCoSoC 2017 - Proceedings, 2017, http://dx.doi.org/10.1109/ReCoSoC.2017.8016147.

[15] S. Lee, C. Son, H. Jang, Distributed and parallel real-time control system equipped FPGA-Zynq and EPICS middleware, in: 2016 IEEE-NPSS Real Time Conference, RT 2016, 2016, http://dx.doi.org/10.1109/RTC.2016.7543117.

[16] X.S. Le, J.-C. Le Lann, L. Lagadec, L. Fabresse, N. Bouraqadi, J. Laval, CaRDIN: An agile environment for edge computing on reconfigurable sensor networks, in: 2016 International Conference on Computational Science and Computational Intelligence, No. 2, CSCI'16, 2016, pp. 168–173, http://dx.doi.org/10.1109/CSCI.2016.38.

[17] X. Revés, V. Marojevic, R. Ferrús, A. Gelonch, FPGA'S middleware for software defined radio applications, in: Proceedings - 2005 International Conference on Field Programmable Logic and Applications, FPL 2005, IEEE, 2005, pp. 598–601, http://dx.doi.org/10.1109/FPL.2005.1515794.

[18] J.A. Williams, N.W. Bergmann, Embedded linux as a platform for dynamically self-reconfiguring systems-on-chip, in: Ersa '04: the 2004 International Conference On Engineering of Reconfigurable Systems and Algorithms, CSREA Press, 2004, pp. 163–169, URL http://espace.library.uq.edu.au/view/UQ:10330.

[19] Y.E. Krasteva, J. Portilla, J.M. Carnicer, E. De La Torre, T. Riesgo, Remote HW-SW reconfigurable wireless sensor nodes, in: Proceedings - 34th Annual Conference of the IEEE Industrial Electronics Society, IECON 2008, 2008, pp. 2483–2488, http://dx.doi.org/10.1109/IECON.2008.4758346.

[20] A. Engel, A. Koch, T. Siebel, A heterogeneous system architecture for low-power wireless sensor nodes in compute-intensive distributed applications, in: Proceedings - Conference on Local Computer Networks, LCN 2015-Decem, 2015, pp. 636–644, http://dx.doi.org/10.1109/LCNW.2015.7365908.

[21] K. Shahzad, P. Cheng, B. Oelmann, SENTIOF : An FPGA based high-performance and low-power wireless embedded platform, in: Proceedings of the Federal Conference on Computer Science and Information Systems, 2013, pp. 901–906.

[22] F. Conti, D. Pajossit, A. Marongiu, D. Rossi, L. Benini, Enabling the heterogeneous accelerator model on ultra-low power microcontroller platforms, Date (2016) 1201–1206.

[23] J. Valverde, A. Otero, M. Lopez, J. Portilla, E. de la Torre, T. Riesgo, Using SRAM based FPGAs for power-aware high performance wireless sensor networks, Sensors 12 (3) (2012) 2667–2692, http://dx.doi.org/10.3390/s120302667.

[24] M. Jacobsen, D. Richmond, M. Hogains, R. Kastner, RIFFA 2.1: A reusable integration framework for FPGA accelerators, ACM Trans. Reconfig. Technol. Syst. 8 (4) (2015) http://dx.doi.org/10.1145/2815631.

[25] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, D.P. Singh, From openCL to high-performance hardware on FPGAs, in: Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012, IEEE, 2012, pp. 531–534, http://dx.doi.org/10.1109/FPL.2012.6339272.

[26] K. Nam, B. Fort, S. Brown, FISH: Linux system calls for FPGA accelerators, in: 2017 27th International Conference on Field Programmable Logic and Applications, FPL 2017, 2017, http://dx.doi.org/10.23919/FPL.2017.8056785.

[27] L. Sommer, J. Korinth, A. Koch, OpenMP device offloading to FPGA accelerators, in: Proceedings of the International Conference on Application-Specific Systems, Architectures and Processors, IEEE, 2017, pp. 201–205, http://dx.doi.org/10.1109/ASAP.2017.7995280.

[28] R. Kirchgessner, A.D. George, G. Stitt, Low-overhead FPGA middleware for application portability and productivity, ACM Trans. Reconfigurable Technol. Syst. 8 (4) (2015) 21:1–21:22, http://dx.doi.org/10.1145/2746404.

[29] A. Parashar, M. Adler, K.E. Fleming, M. Pellauer, J.S. Emer, LEAP: A virtual platform architecture for FPGAs, in: Workshop on the Intersections of Computer Architecture and Reconfigurable Logic, CARL 2010, 2010.

[30] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y.T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, J. Anderson, Automating the design of processor/accelerator embedded systems with legup high-level synthesis, in: Proceedings - 2014 International Conference on Embedded and Ubiquitous Computing, EUC 2014, IEEE, 2014, pp. 120–129, http://dx.doi.org/10.1109/EUC.2014.26.

[31] A. Agne, M. Happe, A. Keller, E. Lubbers, B. Plattner, M. Platzner, C. Plessl, ReconOS: An operating system approach for reconfigurable computing, IEEE Micro 34 (1) (2014) 60–71, http://dx.doi.org/10.1109/MM.2013.110.

[32] A. Rodríguez, J. Valverde, J. Portilla, A. Otero, T. Riesgo, E. De La Torre, FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: The ARTICo3 framework, Sensors (Switzerland) 18 (6) (2018) http://dx.doi.org/10.3390/s18061877.

[33] C. Becker, C. Julien, P. Lalanda, F. Zambonelli, Pervasive computing middleware: current trends and emerging challenges, CCF Trans. Pervasive Comput. Interact. 1 (1) (2019) 10–23, http://dx.doi.org/10.1007/s42486-019-00005-2.

[34] A.H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, Q.Z. Sheng, IoT middleware: A survey on issues and enabling technologies, IEEE Internet Things J. 4 (1) (2017) 1–20, http://dx.doi.org/10.1109/JIOT.2016.2615180.

[35] G. Fersi, Middleware for internet of things: A study, in: Proceedings - IEEE International Conference on Distributed Computing in Sensor Systems, DCOSS 2015, IEEE, 2015, pp. 230–235, http://dx.doi.org/10.1109/DCOSS.2015.43.

[36] M. Eisenhauer, P. Rosengren, P. Antolin, HYDRA: A development platform for integrating wireless devices and sensors into ambient intelligence systems, in: D. Giusto, A. Iera, G. Morabito, L. Atzori (Eds.), The Internet of Things, Springer New York, New York, NY, 2010, pp. 367–373, http://dx.doi.org/10.1007/978-1-4419-1674-7_36.

[37] P. Kostelník, M. Sarnovský, K. Furdík, The semantic middleware for networked embedded systems applied in the internet of things and services domain, Scalable Comput. 12 (3) (2011) 307–315, http://dx.doi.org/10.12694/scpe.v12i3.726.

[38] K. Aberer, M. Hauswirth, A. Salehi, The Global Sensor networks middleware for efficient and flexible deployment and interconnection of sensor networks, Report, Ecole Polytechnique Federale de Lausanne (EPFL) (5005), (LSIR-2006-006) 2006, pp. 1–21.

[39] J. Honkola, H. Laine, R. Brown, O. Tyrkkö, Smart-M3 information sharing platform, in: Proceedings - IEEE Symposium on Computers and Communications, IEEE, 2010, pp. 1041–1046, http://dx.doi.org/10.1109/ISCC.2010.5546642.

[40] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, D. Savio, Interacting with the SOA-based internet of things: Discovery, query, selection, and on-demand provisioning of web services, IEEE Trans. Serv. Comput. 3 (3) (2010) 223–235, http://dx.doi.org/10.1109/TSC.2010.3.

[41] B. Johanson, A. Fox, T. Winograd, The interactive workspaces project: Experiences with ubiquitous computing rooms, IEEE Pervasive Comput. 1 (2) (2002) 67–74, http://dx.doi.org/10.1109/MPRV.2002.1012339.

[42] R. Grimm, One.world: experiences with a pervasive computing architecture, IEEE Pervasive Comput. 3 (3) (2004) 22–30, http://dx.doi.org/10.1109/MPRV.2004.1321024.

[43] A.L. Murphy, G.P. Picco, G.-C. Roman, LIME: a middleware for physical and logical mobility, in: Proceedings 21st International Conference on Distributed Computing Systems, 2001, pp. 524–533, http://dx.doi.org/10.1109/ICDSC.2001.918983.

[44] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications with the tota middleware, in: Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. Proceedings of the, 2004, pp. 263–273, http://dx.doi.org/10.1109/PERCOM.2004.1276864.

[45] F.M. Roth, C. Becker, G. Vega, P. Lalanda, XWARE—A Customizable interoperability framework for pervasive computing systems, Pervasive Mob. Comput. 47 (2018) 13–30, http://dx.doi.org/10.1016/j.pmcj.2018.03.005.

[46] C.M. Barnes, K. Bellman, J. Botev, A. Diaconescu, L. Esterle, C. Gruhl, C. Landauer, P.R. Lewis, P.R. Nelson, A. Stein, C. Stewart, S. Tomforde, CHARIOT - Towards a continuous high-level adaptive runtime integration testbed, in: Proceedings - 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems, FAS*W 2019, 2019, pp. 52–55, http://dx.doi.org/10.1109/FAS-W.2019.00026.

[47] C. Schmitt, B. Stiller, Secure and efficient wireless sensor networks, ERCIM News 2015 (101) (2015) 18–19, http://dx.doi.org/10.5167/uzh-116554, Special theme: The Internet of Things and the Web of Things.

Alwyn Johannes Burger, M.Eng. has been employed as a researcher at Prof. Schiele's Embedded Systems group at the University of Duisburg–Essen since April 2015. He received his bachelor's degree in electrical and electronic engineering at the University of Stellenbosch in South Africa in 2012 with a focus on Robotics. In March 2015 he received his master's degree in engineering from the University of Stellenbosch for his work on autonomous 3D mapping using stereo vision.

Christopher Cichiwskyj, M.Sc. has been working as a researcher and Ph.D. student at the "Embedded Systems" group of the University Duisburg–Essen. He received his master's degree in "Angewandte Informatik" with a focus on distributed, reliable systems at the University Duisburg–Essen. He received his bachelor's degree in 2013 with a special focus on pervasive computing.

Stephan Schmeißer, M.Sc., has been working as a researcher since April 2015 at the Embedded Systems group of Prof. Schiele at the University of Duisburg–Essen. He received his bachelor's degree in 2013 with a focus on pervasive computing. In 2015 he graduated his master's degree in applied computer science with an emphasis on distributed and reliable systems at the University of Duisburg–Essen.

Dr. Gregor Schiele is professor for embedded systems and leads since November 2014 the Embedded Systems group at the University Duisburg–Essen at the campus Duisburg. Before that he was working from 2012 to 2014 at the Insight Centre for Data Analytics and the Digital Enterprise Research Institute (DERI) as well as at the National University of Ireland, Galway. From 2006 to 2012 he was working at the department of Prof. Dr. Christian Becker at the University Mannheim. He wrote his doctorate 2007 at the University Stuttgart at the department of Prof. Dr. Kurt Rothermel.