SMIoT: A Software Architecture for Maintainable Internet-of-Things Applications

Ilse Bohé, Michiel Willocx, Vincent Naessens

imec-DistriNet
KU Leuven, Technology Campus Ghent
Gebroeders de Smetstraat 1, 9000 Ghent, Belgium
firstname.lastname@kuleuven.be

1 Introduction

Many companies across multiple sectors are currently exploring the opportunities of applying Internet-of-Things (IoT) technologies in their digital transformation process. In the transport sector, smart IoT applications can support tracking and parameter monitoring of sensitive or expensive items. Depending on the specific use case, the temperature, humidity, vibration and location can be parameters of interest. In care environments, IoT ecosystems allow for more accurate and reliable patient monitoring while at the same time being less intrusive. IoT applications connected to sensors and actuators support elderly people to live in their own houses without compromising safety. Examples are fall detection mechanisms and smart emergency buttons. Even though connecting sensors and actuators via digital networks to computer systems brings major advantages, many system integrators are currently struggling to build maintainable and cost-efficient IoT ecosystems.

Today, sensor-centric development is the predominant paradigm when building IoT ecosystems. This implies that sensors and actuators are selected and anchored in a very early design stage. Thereafter, system integrators start to think about the design and realization of attractive software applications. However, the lifetime of advanced software applications often outreaches the lifetime of IoT sensors due to their limited cost or harsh conditions in which they are deployed. Unfortunately, many IoT integrations offer no or very limited flexibility when sensors need to be replaced, as sensors are selected at the very beginning. For instance, a temperature sensor from one manufacturer can often not be replaced by another – more robust or cheaper – one from another manufacturer without major code changes due to lack of flexibility during system design. Vendor lock-in is often mentioned as one of the fundamental problems in current IoT deployments. In addition, implementation cycles are often expensive due to the lack of high-level sensor integration support towards application developers. Many IoT sensors currently on the market only offer a low level API, confronting application programmers with low-level connectivity and data representation problems. On its turn, this increases the development cycle times. Moreover, the large gap between business logic and assets on the one hand, and the actual IoT device sensing and actuating on the other hand complicates application development.

This work proposes a paradigm shift from sensor-centric towards application-centric IoT ecosystem design. The SMIoT architecture facilitates the latter, and supports the development of complex and maintainable IoT applications. The architectural guidance allows for dynamic and reconfigurable IoT sensor and actuator integration, and hides low-level implementation details towards application developers. Hence, the latter can focus on business logic without being expert in IoT sensor technology. The proposed architecture is especially useful for software companies focusing on complex IoT applications in a specific domain or sector. A typical example is a software integrator focusing on innovative health environments, or a company building extensible software ecosystems for fleet management. With complex, we mean that the IoT ecosystems can consist of various IoT applications used by different stakeholders in the domain. For instance, in a care environment, applications can be developed for caregivers, elderly people, family, doctors, nurses, government, insurance... Each stakeholder has a partial view on the overall IoT ecosystem. The software may evolve over time, and its lifetime is typically much longer than the lifetime of the sensor technologies that are plugged in. With complex, we also mean that a prototypical development team typically consists of individuals with complementary skills. Some members focus on implementing business logic; others are experienced in supporting the right software abstractions for IoT sensor and actuator technology. The layering offered by our architecture offers the right approach for development teams with mixed skills.

Contribution. The contribution of this work is threefold. First, this work presents SMIoT, an architecture for application-centric IoT development. This approach enables application developers to build complex though maintainable IoT ecosystems. The proposed design principles allow developers to focus on the business logic by abstracting low level IoT protocols, and communication and security mechanisms. We also show the steps that are required to couple IoT devices to applications for different classes of IoT devices with the architectural principles provided by the SMIoT architecture. Second, the architectural principles are incorporated in an Android framework. The implementation demonstrates the flexibility of the proposed architecture. Replacing, adding or removing devices in the infrastructure layer only results in changes in the application configuration and has no impact on the application or its business logic, hereby tackling the vendor lock-in trap. Last, the SMIoT architecture is validated through the design and development of a care home ecosystem. This use case demonstrates how the SMIoT architecture supports rapid application development. Multiple applications for different stakeholders within the same IoT ecosystem are developed exploiting the high degree of code re-usability and intuitive abstractions that can be provided by our framework.

The remainder of this report is structured as follows. Section 2 points to related work. Section 3 gives an overview of the SMIoT architecture. Subsequently, a description of how IoT devices are bound to applications follows in Section 4. Framework support is described in Section 5 followed by a validation of the architectural concepts in Section 6. Section 7 discusses the proposed architecture. This report ends with conclusions.

2 Related Work

Many communication technologies are often combined in complex IoT ecosystems. Their strengths and constraints are evaluated in many papers Al-Fuqaha et al. (2015), Fadlullah et al. (2011), Sanchez-Iborra & Cano (2016), Magnoni (2015). Standardization efforts are presented to enable interoperability. One example is oneM2M Swetina et al. (2014), which aims at establishing a standardized M2M service layer platform for globally applicable and access-independent M2M services. Other standardization efforts focus on specific application domains such as smart homes Zgheib et al. (2017), smart cities Datta et al. (2016) and Industrial Internet-of-Things (IIoT) Mumtaz et al. (2017). Although these standardization efforts target interoperability between IoT components from different stakeholders, the IoT market is still very fragmented. Hence, the flexibility of IoT ecosystem providers is significantly decreased if they are restricted to IoT components adhering to a specific standard. The SMIoT architecture focuses on internal standardization (i.e. by IoT integrator itself). This enables IoT integrators to tailor the interface to IoT components to meet the specific requirements of their applications. Wrappers for IoT standards can enable interoperability with a wide range of IoT components, while wrappers for proprietary protocols provide the flexibility to integrate non-standardized IoT components.

An alternative approach for managing the heterogeneity in IoT components is relying on a cloud or gateway platform that provides uniform interfaces to the IoT components that are connected to the platform. For instance, Lea et. al. Lea & Blackstock (2014) and Demirkan et al. Demirkan (2013) use a cloud-based hub for developing respectively smart city and healthcare applications. A local gateway setup is used by Yang et. al in their MicroPnP Yang et al. (2015) platform. It is a generic zero-configuration, plug-and-play wireless sensor platform consisting of a gateway that interacts with sensor nodes on which sensors/actuators can be added without requiring any additional configuration. Access to the sensors is provided via a REST interface on the gateway. A setup with a combination of a local gateway and cloud hub is used by Soliman et al. Soliman et al. (2013) and Desai et al. Desai et al. (n.d.) in the smart home application domain, and is also adopted by many commercial organizations. For instance, Google[1], Apple[2] and Samsung[3] provide their own smart home IoT platform. Each platform defines APIs that can be supported by third-party IoT device developers to enable interoperability. Using a cloud or gateway platform to bootstrap access to IoT devices significantly simplifies management and application development. However, typically multiple applications can be developed in the context of an IoT ecosystems. Each of these applications can have different requirements with respect to privacy, real-time constraints, access control etc. To fulfill advanced requirements, often a hybrid distributed setup that combines both direct sensor-application interactions as well as interactions mediated via gateways or a cloud platform is required Roman et al. (2013). The architecture presented in this work facilitates application development in the scope of these complex hybrid and distributed setups.

Security and privacy are major concerns in IoT ecosystems Zhang et al.

(2014), Mahmoud et al. (2015). A lot of existing research focuses on authentication and authorization Hernandez-Ramos et al. (2015), Moosavi et al. (2015) and identity management protocols Bernal Bernabe et al. (2017), Barreto et al. (2015) in which edge devices are involved. Some of these mechanisms are already build-in existing commercial-of-the-shelf IoT devices. These proposed security mechanisms are complementary to the presented work. Our architecture allows for transparent handling of authentication and authorization of sensors and actuators in the IoT ecosystem, and allows for integration of sensors and actuators that are equipped with various security technologies.

The domain of context-aware services Abowd et al. (n.d.), Baldauf et al. (2007) and IoT ecosystems are closely related Perera et al. (2014). Sensor data gathered from IoT devices can be used to establish the context of a user and, subsequently, tailor Xiao et al. (2010) the services and content provided by the application. Several papers Abowd et al. (n.d.), Dey (2001) focus on defining the concept of context and describing an ontology Gu et al. (2005) to share context between devices. Other research Rahmati et al. (2015), Put & De Decker (2016) uses context to control access to remote services. For instance, users may only be granted access to highly sensitive corporate documents if they are at the office. The architecture presented in this work is complementary with the research around context-aware services. Currently context information is often restricted to sensor information obtained from the (mobile) device on which the application is running. Several frameworks Doukas & Antonelli (2013), Carlson & Schrader (2012) have been proposed that enable developers to gather and manage context information using the device's sensors. The SMIoT architecture facilitates access to external IoT sensors, enabling applications to establish the user context beyond the regular sensors available on the device. It also contains a context-aware module to determine which IoT components need to be loaded in the application.

3 Architecture

This section gives an overview of the SMIoT architecture. It supports complex and maintainable IoT integrations, and aims at meeting the advanced flexibility and reconfigurability requirements of both system integrators and customers. The architecture facilitates low-cost modifications of IoT ecosystems over time, as new sensor and actuator technologies can be plugged in while keeping the development cost under control. Hence, IoT ecosystems can evolve over time and vendor lock-in can be avoided which, on their turn, can be competitive differentiators for IoT integrators. An overview of the layered architecture and concepts at each layer is given after the discussion of the requirements.

Intuitive interfaces. Developers want to rely on interfaces that make abstraction of details of the underlying infrastructure. In many cases, they even do not want to be confronted with any IoT sensors or actuators at all. This means that application developers want to invoke methods on assets (such as patients, rooms, cars...) instead of sensors (such as fall detector, light source, gps sensor).

Separation of concerns. The architecture facilitates development teams consisting of members with complementary skills. Some developers focus on IoT device integration while others provide domain-specific intuitive interfaces to application developers. Finally, application developers focus on realising application logic and do not want to be confronted with the underlying IoT infrastructure (like the sensor model or vendor, communication technology...).

Reconfigurability. IoT devices can be replaced by other ones – possibly provided by other sensor manufacturers – without affecting the application. Supporting cost-efficient sensor replacement is essential to tackle vendor lock-in, and enables the use of more accurate or less expensive sensors over time, depending on the specific application needs. For example, replacing a Bluetooth temperature sensor by an alternative one that pushes its data directly to the cloud should be possible with limited implementation effort. The application logic remains unchanged.

Context-awareness. The optimal IoT configuration is loaded and initialized based on contextual parameters, and context may evolve over time. The application can rely on sensor data (like beacon technology and GPS data), user information and external data like time to determine the current context. For instance, while a caregiver can visit multiple patients, the application only needs to access and show the sensor devices of the patient he is visiting.

The SMIoT architecture consists of four abstraction layers: the *IoT Infrastructure Layer* , the *Virtual IoT Device Layer*, the *Component Layer* and the *Application Layer*. The layers are depicted in Figure 1.

The **IoT Infrastructure Layer** represents the hardware and software that senses and actuates the physical world, and that stores historical values. This layer can consist of IoT devices, gateways and back-end IoT platforms. IoT devices are physical sensors and actuators. They can interact with the application directly (e.g. via Bluetooth), be mediated by a local gateway, or push their data

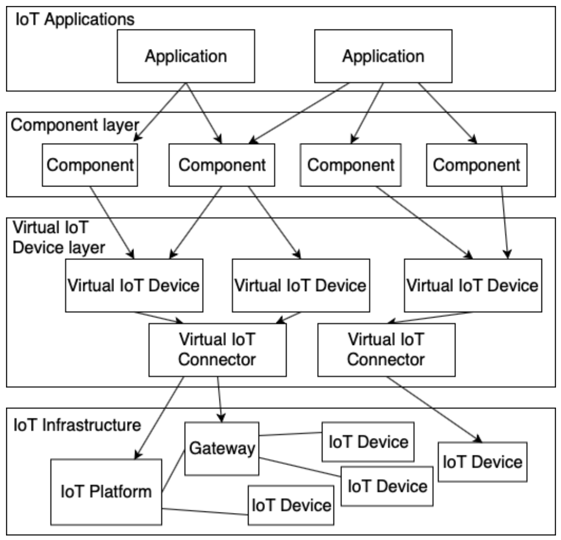


Figure 1: Software layers of the SMIoT architecture.

into an IoT platform. The gateway can either be connected to the application over a local network or push the data to an IoT platform. The devices in this layer can be heterogeneous, and evolve over time. For instance, an IoT ecosystem can contain different types of temperature sensors, potentially relying on different communication technologies.

The **Virtual IoT Device Layer** consists of one or more *virtual IoT devices*. Each *virtual IoT device* defines one sensor or actuator in the IoT ecosystem. A type and technology is tied to each *virtual IoT device*. The type defines a sensor or actuator class. Examples are temperature sensor, lamp, lock... The technology defines the brand and model. Example lamp technologies are Philips Hue lightstrips/bulbs and Osram Lightify models. A uniform interface is assigned to each type. Each technology of the same type must implement the uniform interface. The interface shields the upper layers from technological details. It also defines a uniform representation of the sensor's attributes. For example, a temperature value can be retrieved in °C, °K, or °F. The *virtual IoT device* implementation transforms the values if necessary and passes a uniform representation to the upper layers. Application developers rely on those interfaces to invoke methods on IoT devices. If a sensor is replaced by another one provided by a different manufacturer, a new *virtual IoT device* implementation must be loaded.

Multiple *virtual IoT devices* can point to the same infrastructural element. For instance, multiple sensors can be plugged on a single embedded device. Similarly, access to a set of sensors can be provided via a gateway or a cloud platform. To reduce code redundancy and resource use, *virtual IoT connectors* are defined. *Virtual IoT connectors* are communication handles between the application and each IoT device, gateway or IoT platform. These communication handles implement a part of the communication protocol (e.g. wrap data in a REST request) and the authentication protocol used by the infrastructural element. It provides an API towards the *virtual IoT devices* to access the infrastructural element. If multiple *virtual IoT devices* reside on the same infrastructural element or provided by the same gateway or IoT platform they are linked to the same *virtual IoT connector*. *Virtual IoT connectors* are internal software components and, hence, not exposed to upper layers in the architecture.

The **Component Layer** models the application domain and consists of a set of *components*. Each *component* represents an asset in the application domain. For instance, rooms and patients can be *components* in a care environment. Similarly, trucks and trailers can be *components* in a fleet management ecosystem. Each *component* defines an intuitive interface to interact with the asset. Their implementation relies on the uniform interfaces provided by the *Virtual IoT Device Layer*r. A method in a *component* layer can invoke one or more methods in a *Virtual IoT Device Layer*. For instance, retrieving the heartbeat of a patient *component* will simply result in the invocation of a heartbeat method in a heartbeat sensor. Similarly, to get the location of a truck, the location of a GPS sensor will be requested. In some circumstances, the mapping is less trivial. For example, consider a *component* that contains a heartbeat monitor. Other than a `getHeartbeat()` method, this *component* could also provide a method to developers that pushes a notification when a threshold heartbeat is exceeded. This requires intelligence at *component* level.

The **Application Layer** contains the applications in the IoT ecosystem. These applications invoke the interfaces provided by (a set of) *components* and interact with the physical environment. Hence, application developers do not require knowledge of IoT communication technologies or protocols.

4 Binding an IoT device to an application

In the previous section, *virtual IoT devices* were introduced as a first-class software abstraction in the SMIoT architecture. Each *virtual IoT device* represents one sensor or actuator in the system. This software abstraction allows application developers to access each edge device by means of a uniform and intuitive interface. It means that application programmers are not confronted with complex technology-specific communication handling or data semantics. In fact, the complexity of accessing IoT device functionality is reduced to calling methods with straightforward parameters on local objects. However, although reducing the complexity of sensing and actuating to calling local methods is a noble goal, the latter assumes that the application is already coupled to the right IoT devices. For instance, the application must have the right endpoint information (like name, address...) and credentials to access the IoT device. Setting up an IoT device (i.e. deploying and binding it in a concrete IoT ecosystem) cannot be done fully transparently. The wide variety of IoT platforms and authentication/authorization technologies complicates application binding. This section classifies edge devices according to three categories, and shows how each category can be modelled in the SMIoT architecture. Thereafter, we distinguish three phases that must be foreseen to integrate edge devices in IoT ecosystems and thereafter couple them to applications. The proposed classification covers the wide majority of edge technologies currently on the market:

- *Direct access.* The platform on which the end user application is deployed directly communicates with the IoT device. A prototypical example is a Bluetooth connected smart watch.
- *Gateway mediated device access.* The user communicates with a gateway that can act as a broker for the IoT device. Smart lamps (like Hue en Osram) that are connected to a bridge are examples.
- *IoT platform access.* Finally, the end user application can communicate to a cloud server to retrieve (historical) sensor data, or to actuate sensors.

Direct communication setups are often applied in single user or ad-hoc systems. Access control policies are often simple and static. Every application running on behalf of a user in possession of the credentials can access all device functions. The *virtual IoT device* virtualizes the physical device whereas the *virtual IoT connector* handles the communication with the physical device. The *virtual IoT device* passes the requests to its corresponding *virtual IoT connector.*

In both the gateway and IoT platform setup there is only indirect access between the application and the IoT devices. For instance, applications can only steer Philips Hue lamps via the Hue bridge – which acts as a gateway – when they are on the same local network. If not, the application can eventually access them via a proprietary cloud platform. Each *virtual IoT devices* maps to a physical device. *Virtual IoT connectors* support communication with the gateway or IoT platform instead of interacting with the IoT device directly.

In all setups multiple *virtual IoT devices* can be linked to the same *virtual IoT connector*. This occurs when multiple sensors or actuators are deployed on the same infrastructural element or accessible via the same gateway or IoT platform. In case multiple infrastructural instances of the same technology are rolled out, multiple *virtual IoT connectors* must be constructed. For instance, if multiple Philips Hue bridges are deployed in a physical environment, a separate *virtual IoT connector* is constructed for each bridge.

Three steps often precede calling methods on local objects (i.e. device sensing or actuation). The ultimate goal is to couple an IoT device to an application running on behalf of a principal. During these steps access information and credentials are brought into the scope of the application. The SMIoT architecture foresees intuitive interfaces to handle each step:

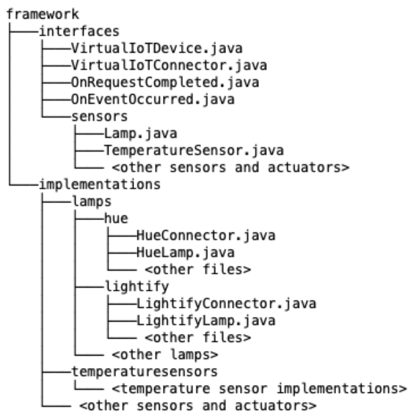
- *Enrollment.* During this phase, physical IoT devices are deployed in the IoT ecosystem. The procedure is technology dependent. Thereafter, access information (both endpoint/address info and credentials) is maintained (either locally or centralized). Finally, each IoT device is linked to one or more components. For instance, a Philips Hue lamp can be coupled to a certain room. Similarly, a heart beat sensor can be coupled to an individual. Enrollment typically occurs once per device, unless some sensors are re-used in other settings after a while.

- *Registration.* During this phase, access information and credentials are brought in the context of an application. The application interacts with a registrar that passes the required access info and credentials after successful authentication of the principal to the registrar. The registrar checks the access rights of the principal before it grants permissions to the application.

- *Device access.* After successful registration, the application can access IoT devices. The application proves to be in possession of the required credentials. The device can optionally further constrain access based on contextual information (such as date, time, location...). After successful access, the devices can be sensed or actuated (i.e. method can be called on *virtual IoT devices*).

5 Android Framework

This section presents a software framework[4] that supports developers with the implementation of the *Virtual IoT Device Layer*. Although the framework presented in this section specifically targets Android, a similar approach can be taken for other platforms.



```
framework
├──interfaces
│   ├──VirtualIoTDevice.java
│   ├──VirtualIoTConnector.java
│   ├──OnRequestCompleted.java
│   ├──OnEventOccurred.java
│   └──sensors
│       ├──Lamp.java
│       ├──TemperatureSensor.java
│       └── <other sensors and actuators>
└──implementations
    ├──lamps
    │   ├──hue
    │   │   ├──HueConnector.java
    │   │   ├──HueLamp.java
    │   │   └── <other files>
    │   ├──lightify
    │   │   ├──LightifyConnector.java
    │   │   ├──LightifyLamp.java
    │   │   └── <other files>
    │   └── <other lamps>
    ├──temperaturesensors
    │   └── <temperature sensor implementations>
    └── <other sensors and actuators>
```

Figure 2: File hierarchy of the virtual IoT device framework.

Figure 2 provides an overview of the file hierarchy of the framework. The framework code is split in two folders: *interfaces* and *implementations*. The former contains generic framework code and defines the uniform interfaces to interact with sensors and actuators. The latter contains implementations of these interfaces for specific IoT technologies. The interfaces are defined as abstract methods in abstract classes.

For each type of sensor/actuator an abstract subclass of `VirtualIoTDevice` is defined that specifies its interfaces. Two types of sensing interfaces can be distinguished: *request-based* and *monitoring*. Request-based calls are used when a sensor value is needed just once (e.g. requesting the temperature of a patient). Monitoring is used when a continuous stream of data is required (e.g. monitoring the heart rate of a patient). With request-based interfaces, the application expects a single response, while a call to a monitoring-type interface typically returns periodic responses. The interfaces are also defined as asynchronous. A callback parameter is added to each method declaration. This shields the component or application developer from the complexity of executing the interaction with IoT devices in a separate thread. This is required since interaction with IoT devices can introduce delays that are undesirable on the main thread. The framework defines two callback interfaces, one for monitoring and one for request-based sensing/actuating (see `OnRequestCompleted` Listing 1). An instance of this interface is passed as a parameter with each call to IoT devices by the application or component developers. These instances handle the response from the IoT device. The framework contains interface definitions for common sensors and actuators such as `Lamp` and `TemperatureSensor`.

Listing 1: OnRequestCompleted Callback Interface

```
public interface OnRequestCompleted<T> {
  public void onSuccess(T response);
  public void onFailure(Exception exception);
}
```

Since the (type of) parameters required to initialize *virtual IoT devices* and *connectors* is typically different for each type of IoT device, a set of keys is defined by the developer that specify each parameter required to initialize the *virtual IoT device* or *connector*. For the *connectors*, these parameters will typically be related to the used communication and authentication technology (e.g. URL or Bluetooth MAC address). For the *virtual IoT devices*, these parameters will be related to the identification of the specific sensor available on the *connector* endpoint and also a *virtual IoT connector* instance itself. Based on the initialization interfaces, a generic device manager can automatically instantiate and initialize *virtual IoT devices* and *connectors* based on configuration files containing the required parameters.

A configuration file is kept by each application instance. Each configuration file describes a subset of the IoT environment, namely the subset of elements that are relevant for that application in a particular epoch. The latter means that the configuration files may vary over time. Each file consists of a set of *components*, *virtual IoT devices* and *connectors*. Devices are coupled to connectors and components. A sample configuration file is given in Listing 2. The parameters that are assigned to each element are described below.

Connector

- **systemID** A unique identifier chosen by the system designers. This identifier is technology-independent.
- **type** A technology dependent name of the *connector*.
- **settings** The setting needed by the *virtual IoT connector* to access the device, gateway or IoT platform. These settings depend on the technology and contain values such as IP-address and access token.

Device

- **systemID** A unique identifier chosen by the system designers. This identifier is technology-independent.
- **type** The device type defines the sensor or actuator class to which the *virtual IoT device* belongs. Examples are temperature-sensor, lamp, lock etc.
- **technology** A brand and model of the IoT device. It determines the underlying implementation that must be loaded in the application.

settings Settings for identifying the physical IoT Device such as the unique id or mac address.

connector The corresponding *connector* the *virtual IoT device* must be connected to, defined by the systemID of the *connector*.

Component

componentName A unique name identifying the *component*.

childComponents A list of *components* which are a part of this *component*, defined by the componentName of the *component*.

devices A list of *virtual IoT devices* connected to the *component*, defined by the systemID of the *device*.

The location of configuration information depends on the specific ecosystem. A list of all *virtual IoT connectors*, *virtual IoT devices* and *components* in the IoT ecosystem can be kept centrally. However, this may negatively impact security and scalability. Distributing configuration info may be likely in many settings. A subset of *connector*, *component* and *device* settings are passed to the application instances, and stored in a configuration file. A registration phase typically realizes this step. Software objects are constructed in an application instance based on the content of the configuration file. For instance, *connectors* can be initialized based on *connector* info in the file. Also, *virtual IoT devices* can be created and linked to *connector* objects. Finally, *components* can be initialized, and each *virtual IoT device* can be added to the right *component(s)*. Replacing a physical device results in an update of the configuration file. The application must be notified in case an infrastructural modification or update occurs, and a delta configuration file must be stored in the context of the application. Devices that no longer need to be accessible by the application can be removed from the configuration file, and newly added devices must be initialized in the application together with their *connector* – if not yet present in the application. Thus infrastructural changes do not imply a change of the application logic and no additional implementation effort is needed.

Listing 2: Configuration file of a care app instance

```
{
  "connectors": [
    {
      "systemID": "HGW1",
      "type": "PHILIPS_HUE_CONNECTOR",
      "settings": {
        "ip":<ip-address of Hue bridge>,
        "authID":<userToken>
      }
    }
  ],
  "devices": [
    {
      "type": "lamp",
      "model": "PHILIPS_HUE",
      "systemID": "LAMP1",
      "settings": {
        "uniqueID":<philips hue unique device id>
      },
      "connector": "HGW1"
    },
    {
      "type": "lamp",
      "model": "HUE",
      "systemID": "LAMP2",
      "settings": {
        "uniqueID":<philips hue unique device id>
      },
      "connector": "HGW1"
    }
  ],
  "components": [
    {
      "componentName": "CareHouse",
      "childComponents": [Room1,Room2],
      "devices": [ ]
    },
    {
      "componentName": "Room1",
      "childComponents": [ ],
      "devices": [Lamp1]
    },
    {
      "componentName": "Room2",
      "childComponents": [ ],
      "devices": [Lamp2]
    }
  ]
}
```

Developers can add new types of sensors or actuators by creating a new abstract subclass of `VirtualIoTDevice` and defining the interface for that specific sensor/actuator. Developers can add support for specific IoT devices by subclassing the `VirtualIoTConnector` class and the abstract class(es) that define the uniform interfaces of the sensors/actuators available on the IoT device, defining the initialization parameters and implementing the initialization and uniform interface methods. Support for IoT devices containing one type of sensor/actuator is contained in one class. If an IoT device consists of multiple types of sensors/actuators, a subclass for each sensor/actuator is created, sharing the same `virtualIoTConnector`. The framework already contains support for several IoT devices. Examples are *Hue lamps* and *Osram lamps.* Since the framework defines asynchronous interfaces, the implementation needs to delegate the interaction with the IoT devices to a separate thread and trigger the callback when the operation is completed. To realize this, the implementations provided with the framework use the RXAndroid framework, which is an Android port of ReactiveX[5]. This framework gives fine-grained control over the execution of operations on different threads. Since Android only allows UI operations on the main thread, our framework uses RXAndroid to execute the interactions with the IoT device on a worker thread while triggering the callback on the main thread. The obtained sensor values can then be show in the UI by the application developer, without overhead of inter-thread communication.

6 Validation: a care home ecosystem

This section applies the architectural principles described in this work to a care home ecosystem. First, the care home ecosystem is described. Next, we focus on the design of the *Component Layer* and *Virtual IoT device Layer* in the care ecosystem.

6.1 The care home ecosystem.

This validation presents a scalable care environment that consists of numerous care home units and their residents. Each care home unit consists of multiple rooms (living room, bathroom, bedroom) and each room is equipped with sensors (e.g. for sensing environmental parameters such as temperature, pressure, humidity) and actuators (e.g. lamps and thermostat). Moreover, also the resident – also called patient – can wear body sensors such as a heartbeat monitor or a fall detection sensor. Figure 3 demonstrates the setup. It is important to note that the set-up of each individual care home unit can be different. A heterogeneous set of devices can be deployed and rooms do not necessarily contain the same set of sensors and actuators. Moreover, each home unit can rely on different types of sensors/actuators to achieve goals.

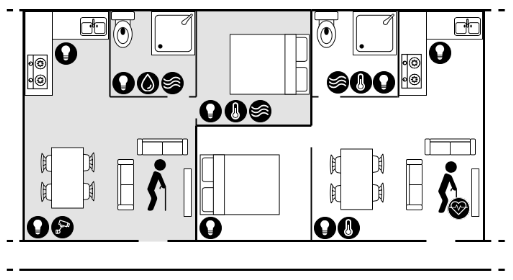


Figure 3: A representation of two care home units.

Several types of end-users are active in this ecosystem: the resident or patient, caregivers, the management and the maintenance crew. Each end-user can use a different application that supports her/his needs. Hence, each type of end-user has a different view on the IoT ecosystem. For example, the IoT environment view of the *resident app* includes his own care home unit, and allows her/him to retrieve sensor values and control actuators in that care home. The view on the environment (and thus the application configuration) in this application will not change frequently. The view only changes when devices are added, removed or replaced. The *caregiver app* needs to control the sensors and actuators of a subset of care home units, namely the ones that correspond to the patients he needs to visit. The view on the environment – along with its configurations – can change according to work schedule of the caregiver. Each time new patients are added/removed to/from the schedule, the app configurations can be adapted. The *maintenance app* needs a different view on the

IoT environment. Maintenance crew need access to particular devices based on reparations and checks that need to be done.

6.2 Application design

6.2.1 The Virtual IoT Device Layer

For each device type used in the care home ecosystem, a uniform interface is defined. For actuators, this interface includes all actions that can be performed by that type of device. For example, this listing provides the uniform interface of a Lamp.

```
public abstract class Lamp extends VirtualIoTDevice {
    abstract void turnOn(<CB>);
    abstract void turnOff(<CB>);
    abstract void changeColor(String clr, <CB>);
    abstract void changeBrightness(int br, <CB>);
}
```

As described in Section 5 each method has a callback (`<CB>`) that returns the result of the call asynchronously.

In the case of a sensor, the uniform interface usually consists of two methods: one for requesting the current value, and one for monitoring purposes over a certain time interval. Below, the uniform interface for a heartbeat sensor is provided. Analogously a uniform interface is provided for the other sensors and actuators.

```
public abstract class HeartbeatSensor extends VirtualIoTDevice {
    abstract void requestHeartbeat(<CB>);
    abstract void monitorHeartbeat(<CB>);
}
```

For each device technology type (i.e. model and vendor), a one-time implementation effort is required to create an implementation that is compliant with the uniform interface. If a specific sensor/actuator technology does not support a method of the interface, an error is thrown.

6.2.2 The Component Layer

Using Figure 3, two parent components can be defined: the `CareHome` and the `Patient`. The `CareHome` consists of three child components: the `Bathroom`, the `Bedroom`, and the `LivingArea`. On its turn, the latter has a `Kitchen` and `LivingRoom` as child components. Each component includes the complete set of all IoT devices that it can contain, and defines the API that is provided to the application layer. Note that care home units are not required to be equipped with all devices defined in the design. The component diagram is presented in Figure 4.

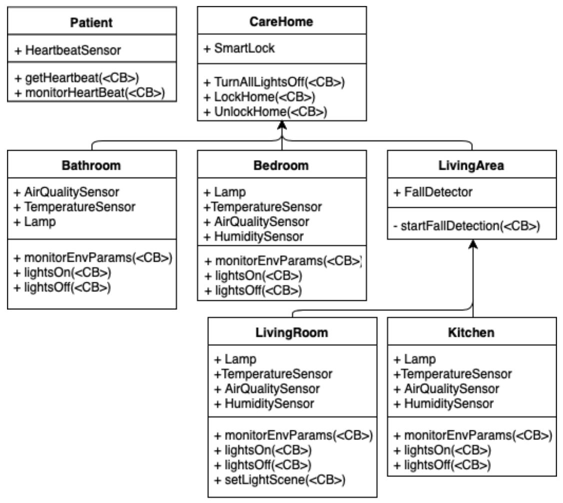


Figure 4: Representation of the Component Layer.

6.2.3 The Application Layer

The *Application Layer* relies on the APIs defined in the *Component Layer*. Depending on the application instance (and thus the type of user), a different subset of *component* APIs and *virtual IoT devices* is used. For example, the *patient app* or the *caregiver app* receives configurations that allows the application to load all *virtual IoT devices* related to one care home unit and one patient, and provides functionality to sense and control all devices located in that room. The *maintenance app* requires access to devices located in several care home units. However, the set of device types they can access is limited to their current assignment. For instance, a particular team can only repair lamps.

7 Discussion

This section discusses the impact of more complex setups on the architecture and deliberates several aspects that need to be taken into account when using the architecture.

In many circumstances, multiple applications possibly on behalf of different stakeholders require access to the same physical device. For many commercial devices, it is not trivial to maintain connections to multiple applications. Examples are many Bluetooth devices or other devices that wipe access tokens each time another device authenticates. In those situations, it is often advantageous that an IoT device remains connected to one application for a longer time which can act as a proxy for other apps. Hence, other applications are able to retrieve sensor data or send actuation commands via the proxy application. For example, a heart rate sensor can be connected to a health app running on a patient device. Caregivers can request or monitor those values via that proxy app. Similarly, a tablet that is mounted in a fixed location in the living room can be persistently connected to lamps in a care home. The patient and the caregiver each have a separate app. Both apps can control the lamps via that tablet. In addition, the proxy approach can also improve the security level. First, credentials to access edge devices can be stored in the context of the proxy app. Hence, loading IoT credentials on multiple end-user apps is no longer required which improves security management. Moreover, this approach can tailor access control to the needs of the IoT ecosystem and their applications. We are no longer constrained to the often primitive access control mechanisms that are built in IoT devices. Device access can be granted based on a set of contextual parameters such as the time and the role and identity of the principal. Also, access policies can specify the (sub)set of API calls are allowed in certain circumstances.

Currently, commercial smartphone platforms support permissions handling based on user consent. Examples are permissions to access user data (like contacts) and system features (like camera). Similarly, users controlling an IoT application would benefit from a permission-based mechanism that regulates access towards the IoT devices in the system. However, our IoT applications present a high level of dynamics and heterogeneity. Different application instances installed form the same build can lead to totally different types and technologies that need to be coupled. The static permission handling currently provided by Android and iOS has shortcomings and cannot be copied.

Several non-functional aspects related to IoT device selection (i.e. aspects other than which IoT device interfaces are supported) can have a significant impact on the behavior of the application. Hence, it is not always possible to replace an IoT device with any other device providing the same interfaces. Applications may require a specific Quality of Service (QoS) from IoT devices to ensure the desired behavior. This can be specified via QoS parameters such as measurement resolution, latency, sampling frequency, security, form factor and communication range.

For example, IoT devices using LPWAN technologies typically have a longer latency compared to devices using more low-range communication technologies such as Bluetooth. Depending on the use case a device with a specific measurement resolution or latency can be selected. For instance, for temperature monitoring in smart city applications an IoT device with long-range communication, low measurement resolution and sampling frequency may be selected while these sensors would not be adequate for climate control applications. Further, some applications rely on advanced features which are not present in all devices of a certain type. For instance, applications aiming at creating various atmospheres in a room, rely on color and brightness. Not all lamps support those properties.

The *virtual IoT device* framework makes abstraction of the monitoring frequency. The system administrator can specify the frequency at which sensor values are monitored. For IoT devices that support monitoring via, for instance, publish-subscribe messaging protocols such as MQTT this is typically defined as a parameter on the IoT device. For devices that do not support these types of protocols, the monitoring interface can be implemented via software-based polling by the *virtual IoT device.* Although this enables more flexibility in the specification of the monitoring frequency (i.e. the monitoring frequency can be provided as a parameter to the *virtual IoT device*), it can negatively impact the performance of the application (e.g. computational load, battery life). This is especially the case if sensor values should not be monitored at a fixed frequency but only if they pass a certain threshold (e.g. trigger an alarm if the temperature is too high). In these case, polling also significantly reduces the battery life of the sensor.

The *Virtual IoT Device Layer* and *Component Layer* provide a high-level abstraction of the underlying IoT infrastructure to application developers. In some cases, even significant changes in the underlying infrastructure have no impact on the *Application Layer.* For instance, presence of people can be detected either via infrared sensors or via cameras. In some care homes, cameras may be used for presence detection because they are also used for security services. In other care homes, infrared sensor may be selected for cost reasons. The application developer can use the same presence interface on the *component* (e.g. *Room*), regardless of the underlying infrastructure. However, some changes in the underlying infrastructure will trigger changes in the *component* APIs provided to application developers. For instance, fall detection systems working with accelerometers will typically be used via the *component Patient*, while systems working with cameras will be used via, for instance, *Room components.*

Most applications contain both IoT and non-IoT related functionality. This is also reflected in the interfaces provided by the *components* to application developers.

For example, care home applications typically require access to medical records of the *Patient.* Hence, the IoT-related interfaces of *components* are often complemented with interfaces supporting more traditional application-level operations.

For systems that are retrofitted to include IoT applications, often *components* can wrap existing object representations of assets and add the IoT-related interfaces to the already existing functionality.

8 Conclusion

This work presented SMIoT, an architecture supporting IoT application developers building maintainable IoT applications. Our architecture is especially beneficial for application-centric development . The architecture proposes two abstraction layers that allow to decouple the application from the underlying IoT infrastructure. The *Virtual IoT Device Layer* provides a uniform and intuitive interface for sensor and actuator access and shields complex low-level and non-functional aspects like communication mechanisms and security protocols from the application developer. The intuitive APIs in the *Component Layer* imply that the developer can focus on implementing business logic and intuitive user interaction. We also presented guidelines to integrate and couple different types of IoT devices, and switch between them. The architectural insights are incorporated in an Android framework and validated through the design and development of a care home ecosystem consisting of various IoT components.

References

Abowd, G. D. et al. (n.d.), Towards a better understanding of context and context-awareness, *in* 'Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing', HUC '99, Springer-Verlag, London, UK, UK, pp. 304–307.

Al-Fuqaha, A. et al. (2015), 'Internet of things: A survey on enabling technologies, protocols, and applications', *IEEE Communications Surveys Tutorials* **17**(4), 2347–2376.

Baldauf, M., Dustdar, S. & Rosenberg, F. (2007), 'A survey on context-aware systems', *Int. J. Ad Hoc Ubiquitous Comput.* **2**(4), 263–277.

Barreto, L., Celesti, A., Villari, M., Fazio, M. & Puliafito, A. (2015), Identity management in iot clouds: A fiware case of study, *in* '2015 IEEE Conference on Communications and Network Security (CNS)', pp. 680–684.

Bernal Bernabe, J., Hernandez-Ramos, J. L. & Skarmeta Gomez, A. F. (2017), 'Holistic privacy-preserving identity management system for the internet of things', *Mobile Information Systems* **2017**.

Carlson, D. & Schrader, A. (2012), Dynamix: An open plug-and-play context framework for android, *in* '2012 3rd IEEE International Conference on the Internet of Things', pp. 151–158.

Datta, S. K., da Costa, R. P. F., Bonnet, C. & Härri, J. (2016), onem2m architecture based iot framework for mobile crowd sensing in smart cities, *in* '2016 European Conference on Networks and Communications (EuCNC)', pp. 168–173.

Demirkan, H. (2013), 'A smart healthcare systems framework', *IT Professional* **15**(5), 38–45.

Desai, P., Sheth, A. & Anantharam, P. (n.d.), Semantic gateway as a service architecture for iot interoperability, *in* '2015 IEEE International Conference on Mobile Services', pp. 313–319.

Dey, A. K. (2001), 'Understanding and using context', *Personal Ubiquitous Comput.* **5**(1), 4–7.

Doukas, C. & Antonelli, F. (2013), Compose: Building smart amp; context-aware mobile applications utilizing iot technologies, *in* 'Global Information Infrastructure Symposium - GIIS 2013', pp. 1–6.

Fadlullah, Z. M., Fouda, M. M., Kato, N., Takeuchi, A., Iwasaki, N. & Nozaki, Y. (2011), 'Toward intelligent machine-to-machine communications in smart grid', *IEEE Communications Magazine* **49**(4), 60–65.

Gu, T., Pung, H. K. & Zhang, D. Q. (2005), 'A service-oriented middleware for building context-aware services', *Journal of Network and Computer Applications* **28**(1), 1 – 18.

Hernandez-Ramos, J. L., Pawlowski, M. P., Jara, A. J., Skarmeta, A. F. & Ladid, L. (2015), 'Toward a lightweight authentication and authorization framework for smart objects', *IEEE Journal on Selected Areas in Communications* **33**(4), 690–702.

Lea, R. & Blackstock, M. (2014), City hub: A cloud-based iot platform for smart cities, *in* '2014 IEEE 6th International Conference on Cloud Computing Technology and Science', pp. 799–804.

Magnoni, L. (2015), Modern messaging for distributed sytems, *in* 'Journal of Physics: Conference Series', Vol. 608, IOP Publishing, p. 012038.

Mahmoud, R., Yousuf, T., Aloul, F. & Zualkernan, I. (2015), Internet of things (iot) security: Current status, challenges and prospective measures, *in* '2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)', pp. 336–341.

Moosavi, S. R., Gia, T. N., Rahmani, A.-M., Nigussie, E., Virtanen, S., Isoaho, J. & Tenhunen, H. (2015), 'Sea: a secure and efficient authentication and authorization architecture for iot-based healthcare using smart gateways', *Procedia Computer Science* **52**, 452–459.

Mumtaz, S. et al. (2017), 'Massive internet of things for industrial applications: Addressing wireless iiot connectivity challenges and ecosystem fragmentation', *IEEE Industrial Electronics Magazine* **11**(1), 28–33.

Perera, C., Zaslavsky, A., Christen, P. & Georgakopoulos, D. (2014), 'Context aware computing for the internet of things: A survey', *IEEE Communications Surveys Tutorials* **16**(1), 414–454.

Put, A. & De Decker, B. (2016), Pacco: Privacy-friendly access control with context, *in* 'Proceedings of the 13th International Joint Conference on e-Business and Telecommunications', ICETE 2016, SCITEPRESS - Science and Technology Publications, Lda, Portugal, pp. 159–170.

Rahmati, A. et al. (2015), Context-specific access control: Conforming permissions with user expectations, *in* 'Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices', SPSM '15, ACM, New York, NY, USA, pp. 75–80.

Roman, R., Zhou, J. & Lopez, J. (2013), 'On the features and challenges of security and privacy in distributed internet of things', *Computer Networks* **57**(10), 2266 – 2279. Towards a Science of Cyber Security Security and Identity Architecture for the Future Internet.

Sanchez-Iborra, R. & Cano, M.-D. (2016), 'State of the art in lp-wan solutions for industrial iot services', *Sensors* **16**(5).

Soliman, M., Abiodun, T., Hamouda, T., Zhou, J. & Lung, C. H. (2013), Smart home: Integrating internet of things with web services and cloud computing, *in* '2013 IEEE 5th International Conference on Cloud Computing Technology and Science', pp. 317–320.

Swetina, J. et al. (2014), 'Toward a standardized common m2m service layer platform: Introduction to onem2m', *IEEE Wireless Communications* **21**(3), 20–26.

Xiao, H. et al. (2010), An approach for context-aware service discovery and recommendation, *in* '2010 IEEE International Conference on Web Services', pp. 163–170.

Yang, F. et al. (2015), Micropnp: Plug and play peripherals for the internet of things, *in* 'Proceedings of the Tenth European Conference on Computer Systems', EuroSys '15, ACM, New York, NY, USA, pp. 25:1–25:14.

Zgheib, R., Conchon, E. & Bastide, R. (2017), *Engineering IoT Healthcare Applications: Towards a Semantic Data Driven Sustainable Architecture*, Springer International Publishing, Cham, pp. 407–418.
URL: *https://doi.org/10.1007/978-3-319-49655-9*$_4$9

Zhang, Z.-K., Cho, M. C. Y., Wang, C.-W., Hsu, C.-W., Chen, C.-K. & Shieh, S. (2014), Iot security: ongoing challenges and research opportunities, *in* 'Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on', IEEE, pp. 230–234.

Notes

[1]https://developers.nest.com
[2]https://www.apple.com/us/shop/accessories/all-accessories/homekit
[3]https://www.smartthings.com/
[4]The source code of the Android framework can be found here:
`https://github.com/msec-kul/SMIoT_v1.git`
[5]`http://reactivex.io`