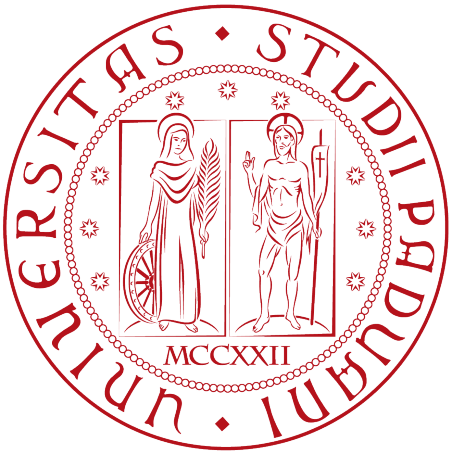
University of Padua

Department of Mathematics "Tullio Levi-Civita"

Master Degree in Computer Science



Integrating IoT devices in 3rd-party smart-home ecosystems: local vs remote middleware solutions

Master Thesis

Supervisor

Prof. Tullio Vardanega

Industrial Advisor

Engr. Giuseppe Ursino

Student

Mariano Sciacco, BSc

Academic Year 2021-2022

I dedicate my master thesis to my entire family, who always inspire me with their valuable advice that I will carry with me in life. A special feeling of gratitude to my loving parents, Patrizia and Riccardo, and my brother, Alessandro, whose good examples have taught me to work hard for the things that I aspire to achieve.

"The thing is, it's very easy to be different, but very difficult to be better."

— Jonathan Ive

Acknowledgements

I would like to show my deep appreciation to my university supervisor, Prof. Tullio Vardanega, who – as a mentor – helped me finalize my master thesis with his invaluable patience and feedback. I would also like to express my deepest gratitude to my internship supervisor, Alberto Pomella, and my industrial advisor, Giuseppe Ursino, who guided me throughout this work. Additionally, I am also thankful to Giorgio Audisio, Electronics R&D Director of Vimar SpA, who allowed me to realize my experiments and researches with the company.

Lastly, I'd like to mention my friends and family who supported me and offered an in-depth insight into the study. Their confidence in me has kept my spirits and motivation high during the preparation of this work.

Padua, September 2022 Mariano Sciacco

Abstract

The Internet of Things is an especially prominent vector of evolution for commercial applications, including smart-home ecosystems. An IoT ecosystem is an ensemble of web-connected devices able to collect, send and act on environmental data. Individual manufacturers employ proprietary data models to endow their devices with ad-hoc properties, functionalities and relationships.

When a 3rd-party ecosystem integration is required in a product, a common language needs to be defined, overarching foreign data models, so that all devices may be interacted with. To address this need, an integrator must use a local or remote middleware, whose deployment affects response times, user functionalities, and maintainability. The former option is less general but incurs less latency; the latter is more versatile, at the cost of higher latency and more complex data exchange. This thesis surveys and compares state-of-the-art integration strategies, and formulates two original solutions.

Contents

List of Figures

List of Tables

Listing of Code Snippets

Chapter 1

Problem statement

1.1 Context overview

In the last twenty years, the raise of the Internet contributed to make significant changes in modern societies. Novel application domains have emerged, some of which are social networks, *smart-connected* networks, and most prominently Internet of Things. In this era of evolution, IoT plays a core role for commercial applications, especially in domotic [6]. As a matter of fact, many modern home devices are also labeled *smart*, meaning that they are capable of connecting to the Internet, exchanging data with other devices and interacting with human beings through software applications. Therefore, *smart* devices are interconnected with the external world and employed for smart-home and consumer electronics. These devices are generally sensors or actuators that are directly accessed by the final user through a simple interface. Such capabilities enable users to address multiple needs from the functional viewpoint. Some of these needs are:

- remote light switches control (e.g. on/off, intensity);
- scheduled tasks – or routines – execution (e.g. open all home window shades in the morning);
- device automations to perform a pre-planned set of actions at once (e.g. turning on a light triggers another light to turn on).

To successfully satisfy these needs, IoT requires a well-designed infrastructure that can manage a growing number of heterogeneous and connected devices. This aggregation of IoT devices compose an **IoT ecosystem**. As reported in a recent survey of IoT networks [16], a big limitation occurs when designing an IoT infrastructure: there is no standard networking protocol for IoT applications, meaning that a vendor can freely choose any wired or wireless technology available (e.g. from [16]: Bluetooth, ZigBee, LTE). Indeed, most manufactures tend to adopt at least a standard protocol when designing a new IoT device, as opposed to inventing a custom one. Moreover, manufacturers are not interested in creating a single device that supports all the existing protocols, because it is unfeasible in terms of cost and maintainability. The role of interoperability shall be delegated to an IoT gateway instead, which should manage different IoT devices as an intermediary entity.

With these considerations in mind, an IoT network normally includes multiple heterogeneous devices from multiple vendors. This leads to a variety of devices, which can work according to a vendor's IoT ecosystem. However, if the ecosystem is closed,

few (if any) foreign devices can inter-operate with it, which is not what the end-user wants. To address this need, an ecosystem must be open to *3rd-party device integration* (i.e. the integration of devices made by a foreign manufacturer with respect to the ecosystem provider). Therefore, a manufacturer should accommodate the interaction between its ecosystem and multiple external ecosystems by design (Figure 1.1). Several solutions have been proposed around the concept of *middleware* [19].

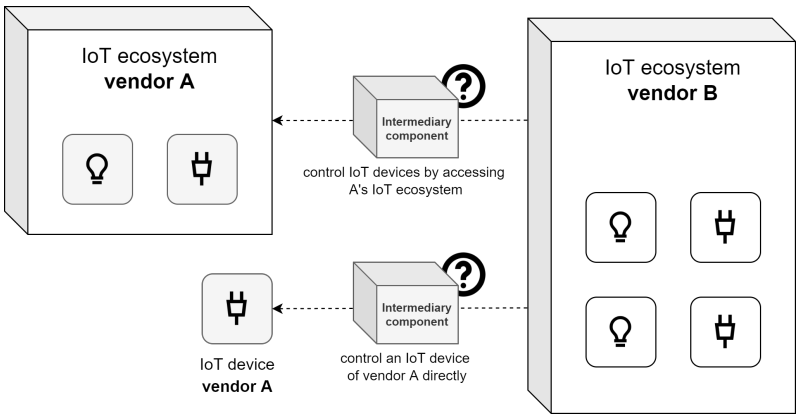


Figure 1.1: IoT ecosystems interaction problem with two intermediary components to control IoT devices of vendor A.

Nevertheless, determining which solution should be most convenient needs to take stock of the current landscape and the emerging trends, through a comprehensive survey. This is also essential to understand data flows of the IoT infrastructure and its interconnections.

1.2 Data-model definition

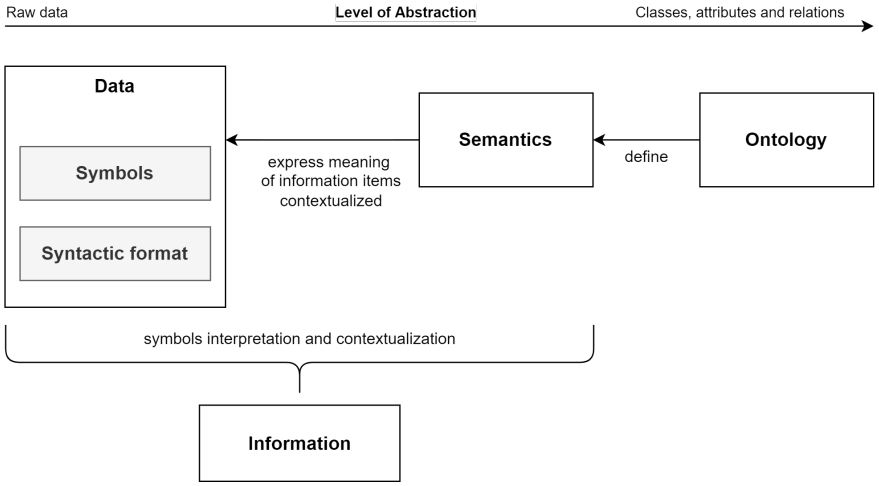


Figure 1.2: Relationships between ontology, semantics, data and information to compose a data-model.

An IoT ecosystem is an ensemble of network-connected devices able to collect, send and act on environmental data. Each manufacturer designs its ecosystem according to its needs, based on a **data-model**. This data-model defines a formalism that uses ontology and semantics to interpret interactions with devices in an ecosystem. The **ontology** defines in a structured manner the semantics conveyed by the entities of interest, while **semantics** captures the intended meaning of information items. **Data** is expressed through symbols and has a syntactic format, thanks to which information is extracted. Therefore, data becomes **information** through symbols interpretation. For example, an IoT device transmits data to report a huge variety of values belonging to a sensor (e.g. temperature, on/off status, light intensity). Figure 1.2 shows the relationships of the terminology just defined.

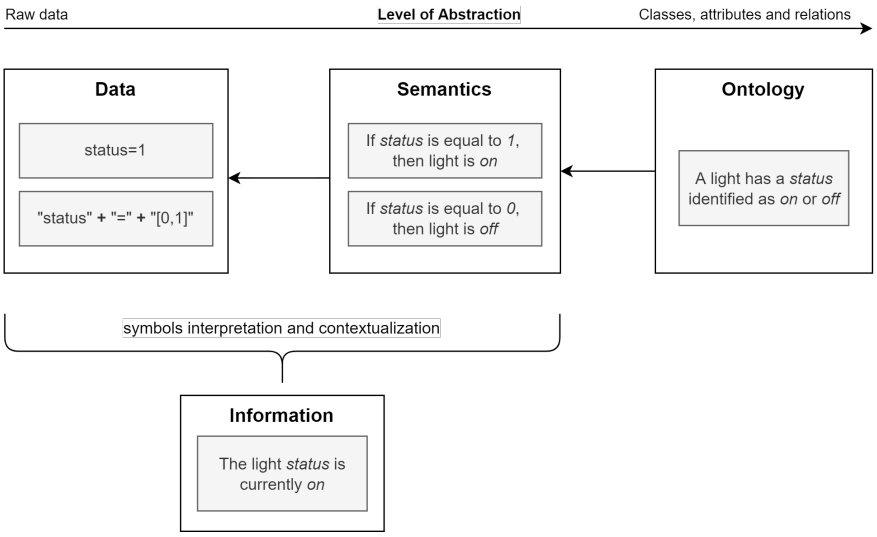


Figure 1.3: Example of a data-model to express the status of a light bulb.

An example may clarify: the data-model of a light bulb. A light has a status that tells whether it is turned on or off. To express this status, a manufacturer can choose either integers (i.e. 0, 1) or boolean values (i.e. `TRUE`, `FALSE`) to use as data symbols. Moreover, a syntactic data format must be employed to represent this status (e.g. a key-value format like `status=1`). Symbols and syntax rules constitutes data representing the status of a light. However, a foreign manufacturer that wants to interpret symbols must know semantics in order to understand the intended meaning of information items in a precise context. Therefore, Figure 1.3 shows that information is extracted from data through symbols interpretation and contextualization.

In this work, these concepts are essentials to understand how data is processed through various components of an IoT ecosystem.

1.3 IoT ecosystems

An IoT ecosystem aims at empowering the communication with several device types, both in a local network (e.g. in-house devices) and remote network (e.g. Internet-connected devices). Each device can be interpreted as a node, meaning that is capable of intercommunicating with other entities in a network. Each node is equipped with a

wireless (or wired) network technology, which is able to collect data from the physical environment, exchange it with other nodes and send it through the Internet to a remote node. This requires a sustainable way to handle and organize data to avoid incurring network congestion so that the growing network remains stable over time. This means that the network will be not subject to slowdowns or failures during its operation. Moreover, interoperability between devices in the same network is a requirement for manufacturers, while end users demand for transparency. If a node is capable of extracting information from data provided by a different vendor's node, then the awareness of the surrounding environment is enhanced. This means that all nodes in the network can intercommunicate with each other. Specifically, one node is aware that other nodes exist and so it can ask for data. In order to understand the concepts of interoperability and transparency, there are three concerns that need to be analyzed:

- The first concern pertains the structure and relationships of different components to build an **IoT architecture**, that satisfies multiple needs for users and manufacturers (e.g. remote device control).
- The second concern is the **smart-home infrastructure**, which involves the design of a specific IoT architecture along with home-related services and devices.
- The third and last concern is an anticipation of future IoT evolution with emerging **IoT challenges**. These challenges reflect commercial needs for manufacturers and functional needs for end users such as security, privacy and context-awareness.

1.3.1 IoT architectures

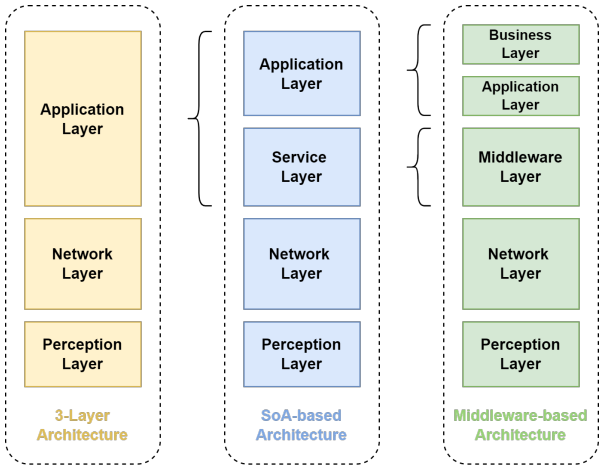


Figure 1.4: IoT architecture views proposed in article [17].
Source: recreated from [17]

In recent literature, the authors of [17] generalize IoT in a 3-layer architecture. At the highest level of the architecture corresponds an abstraction of IoT objects (e.g. capabilities, information), while at the lower level devices are defined (e.g. raw data). The structure is defined as a generic three-level IoT architecture (Figure 1.4) and consists of:

- the **application** layer, giving end-users access to readable information and usable services;
- the **network** layer, which supports data transmission among devices and services;
- the **perception** layer, which handles network routes and IoT devices equipped with sensors or actuators.

More recently, this kind of architecture became unable to capture all features of modern IoT systems. According to [16], [2] and [17], the IoT architecture can be interpreted in a 4 or 5-layer architecture. In this case, the **5-layer architecture** is taken into account as a specialization of simpler architectures.

Starting from the surface, the **business** layer provides analytics, data processing and usage statistics from the acknowledged information of the lower layer. In practice, this is the part that manufacturers use to decide improvements and refinements to act on devices in production. This task is essential from the commercial viewpoint, because not only technology is important, but also how the device functionalities are delivered to customers.

The layer below the business layer is called **application** layer. This layer provides an interface for the end users to acquire information extracted from data received from the underlying layers. Moreover, it is capable of hosting several sets of services, depending on the context: smart-homes [2], providing functionalities to smart appliances and home-related devices; health-care [22], providing ad-hoc devices to measure blood-pressure and heart-rate monitor to constantly tracking health condition of a patient; smart-industry [23], which provides IoT machines (e.g. 3D printers) and monitoring tools to improve productivity of industrial processes; and autonomous driving [11], whose objective is to equip cars with self-driving functionalities regarding object detection, trajectory detection and decision logic. In the end, this layer hides the complexity and the heterogeneity underneath, while providing clear information comprehensible for the end user. As a matter of fact, data is processed in the lower layers, which are closer to IoT devices and data sources. In academic literature [16], the *application* layer is also referred to *service* layer.

The third layer is called **processing** layer, but sometimes it is also referred to *support* layer, *platform* layer or *middleware* layer. The main purpose of this layer is to store and interpret data received from the lower layers. Here, the semantics is employed to understand the intended meaning of information items extracted from data comprehensible for IoT devices; then, data becomes available to the upper layer (the application layer), whose purpose is to serve it in a form of application service. This layer, instead, does not provide any kind of service to the end user but handles the entire logic underneath. This layer is also in support of context-awareness, predictions and cooperation among things. However, the data analysis are a bit different with respect to the *business* layer. In fact, in the *processing* layer data is pure and clean, while in the *business* layer data is enhanced in terms of information items according to user interactions from the *application* layer. For example, if a user interacts with a light switch on a mobile app, the *processing* layer will use the data to send the action to the device, while the business layer will interpret this act as a statistic to learn user behavior.

The **network** layer is the fourth layer of the architecture. It is used to transmit data among devices through the network. Hence, this layer uses routers, gateways and hubs to handle the connection with multiple devices at the lower layer. Here, raw data is received from lower-layer devices and is routed to other devices or to the upper layer.

This layer is also responsible for ruling the network traffic among different devices underneath.

The **perception** layer is the lowest layer of the architecture. In this layer, there are devices capable of communicating raw data perceived from the environment. Then, data is sent towards a gateway or a hub, belonging to the upper layer. The type of devices in this layer may vary from sensors to actuators. Generally, these devices are *smart* enough to autonomously send data to the upper layer using a pre-defined network protocol (e.g. ZigBee, BLE).

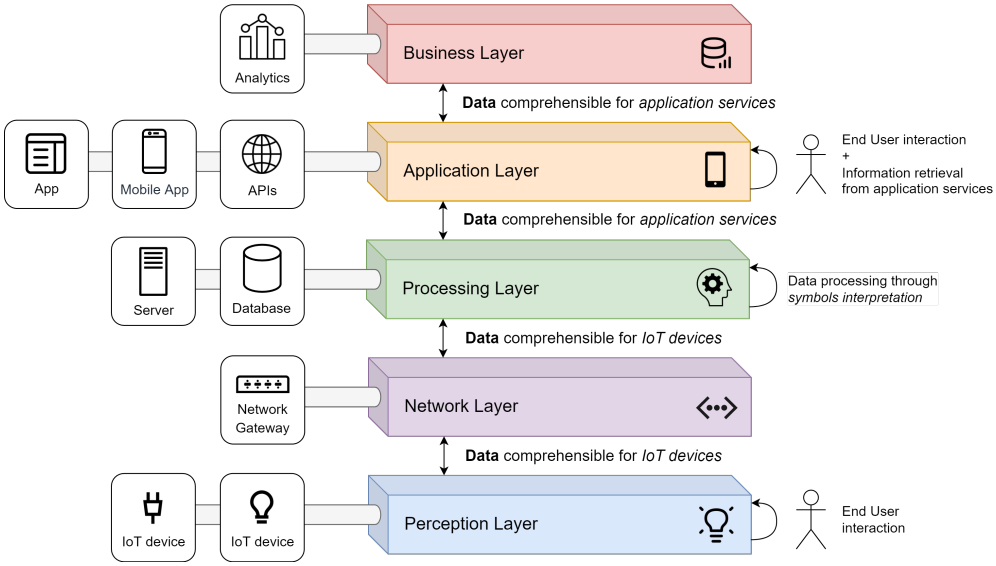


Figure 1.5: IoT five-layer architecture view proposed in this work.

Figure 1.5 shows a vision of an IoT ecosystem that can be seen as a 5-layer stack. Though, in each layer a manufacturer can choose between a variety of standard protocols, network technologies and applications to provide different options to users. This necessity creates a huge heterogeneity in IoT ecosystems that leads to massive decrease of interoperability. Vendors must confront with the IoT market, which continually demands new functionalities. Therefore, as also shown in [20], IoT vendors should consider the inter-operation of heterogeneous devices and systems as a core point of the infrastructure, thus opening the ecosystem to 3rd-party vendors. Such vendors will have the possibility to satisfy a higher number of market needs, which consequently will give in return a higher number of clients. In order to attain interoperability, both the *perception* layer and the *application* layer may have to be involved. The former layer provides interoperability through standard protocols such as ZigBee, Bluetooth and Z-Wave. To achieve this, a 3rd-party vendor must implement in their devices a protocol, which is the same used by a gateway or a hub of the proprietary vendor from the *network* layer. On the other hand, the latter layer – i.e. perception layer – requires apps integration via Cloud through the use of a common interface, at higher level of abstraction. Therefore, the authors of [2] put forward open challenges for the provision of frameworks that can support the production of interoperable smart-home ecosystems. Before exposing these challenges, smart-homes must be defined to properly understand their architecture.

1.3.2 The idea behind smart-homes

The smart-home domain is considered an important research ambit in IoT applications. The main purpose of smart-homes is to provide interconnected devices inside a house that are able to interact with other nodes – and humans – via a local or remote network. Report [2] analyzes the concept of smart-homes in the light of end-user applications and current challenges. In fact, in the last decade, smart-homes achieved huge popularity considering the comfort and quality of life. A user can buy all the needed home-related *smart* appliances, and then everything is controlled through a smartphone app or a software application. In a broader context, smart-homes are considered an extension of building automation, which involves the control and automation of embedded technology. A smart-home ecosystem is made of IoT devices equipped with sensors and actuators connected to a network gateway. The gateway can be connected to the Internet and linked to a Cloud service (e.g. a remote server) to exchange data. Although newer IoT devices are installed, this entire structure should provide an adaptive and progressive ecosystem, which welcomes the constantly-changing needs of home residents. Moreover, manufacturers provide progression and adaptation of IoT ecosystem through firmware and software updates, enabling IoT devices, gateways and server to accommodate new features and new smart-home devices in the future. As reported in [6], the smart-home architecture require at least a few core components to be defined, reflecting the IoT ecosystem structure (Figure 1.5).

- Firstly, **IoT devices** are the source where data and measurements are collected. In fact, devices are generally equipped with sensors, to read data from the environment, and actuactors, to provision and execute commands. A few examples are thermostats, temperature sensors, light switches, energy meters and roller shutter modules. Therefore, IoT devices can be naturally associated to the perception layer.

- Secondly, a **network gateway** and a **server** are required to pass data and process actions, respectively. A network gateway is used in-home to provide network traffic rules and Internet connectivity, while a server can be placed in-home or in-cloud to provide a *compute unit*. In addition, there are also in-home **powerful gateways** in support of a Cloud server. These gateways have enough power to rule network traffic and to interpret data using drivers. These drivers are placed and executed in powerful gateways to interact with a specific IoT device, without demanding computation to the Cloud. Eventually, gateways and server are associated to the network layer and the processing layer.

- Thirdly, a **database** is needed to store collected data and to perform data analysis, as well as data visualization. Databases can be defined in the form of relational and non-relational structures, depending on the needs of the manufacturer. Therefore, this component belongs to the processing layer.

- Furthermore, an **APIs** (Application Programming Interfaces) are required to allow external application to manipulate the existing system, following pre-defined paradigms. For example, an API can serve data to external agents, send push notification and also fulfill user-desired actions, according to what a manufacturer decided to expose. Therefore, APIs can be used to send data to a *Mobile App* or collect statistics that are used for profiling. Accordingly, this part is associated to the application layer.

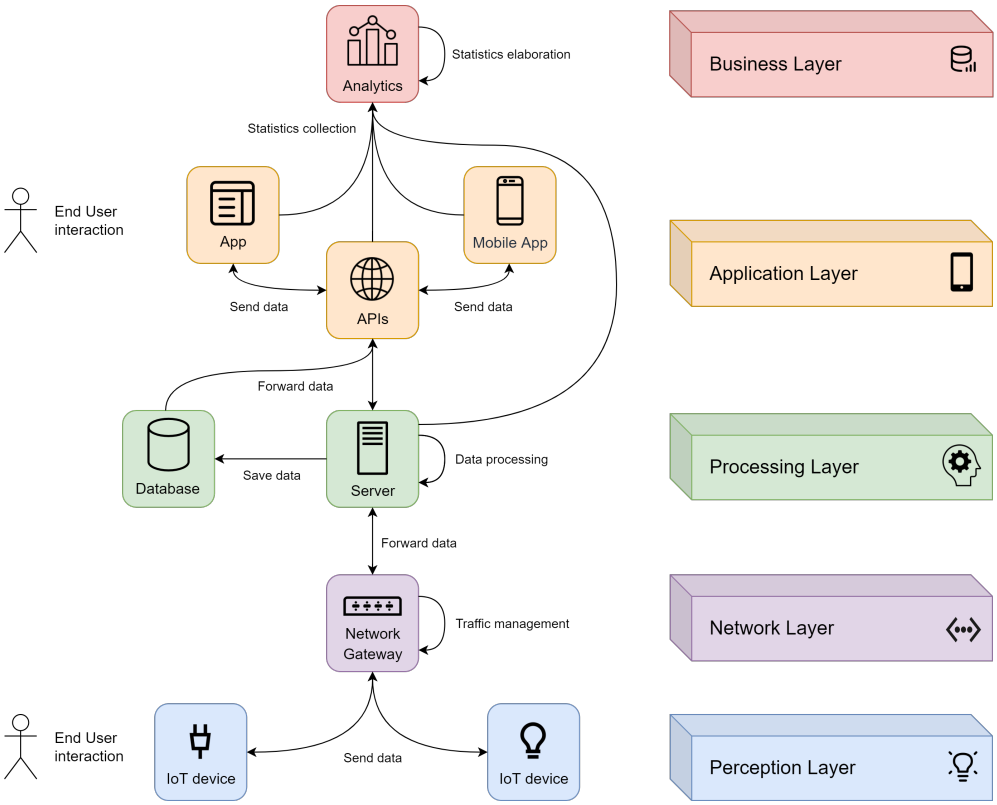


Figure 1.6: Smart-home components in parallel with the proposed five-layer architecture of IoT.

With this structure in mind, Figure 1.6 should represent in a cleaner way these entities. In this picture, the analytics belongs to the business layer, whose function was already discussed in §1.3.1. This architecture addresses several user needs in terms of home automation, but it also raises new issues regarding device interoperability, security and privacy. Now, these issues must be addressed properly around the context of smart-home and IoT in general.

1.3.3 Challenges of modern IoT systems

As previously noted in §1.3.1, interoperability is a huge and central challenge to allow the co-existence of heterogeneous devices in an IoT ecosystem. At the same time, the constant growth of IoT ecosystems opened new challenges regarding security, privacy, context-awareness and energy-awareness of IoT devices. Security and privacy challenges may affect the IoT architecture entirely. Meanwhile, context-awareness and energy-awareness only affect lower levels of the IoT architecture in acknowledging the behavior of other IoT devices. To better understand the problem, below we discuss each challenge in isolation.

Starting from **security**, gateways in the network layer connect to the Internet in order to widen the range of the convenience services availed to end users. However, such services must be secured to mitigate security threats. First of all, many devices allow remote management using an external server as an intermediary. This is a security risk as information becomes controlled and accessible through multiple vectors

(e.g. mobile application, desktop app, chat bots, etc.). Therefore, this requires proper authentication management service. As a matter of fact, an attacker can control devices remotely and use them to hack the entire system. Moreover, data exchanged through multiple nodes – even inside the local network – can be easily eavesdropped and altered. Hence, to ensure integrity and confidentiality of exchanged messages a proper countermeasure must be provided inside the IoT ecosystem. Several security threats have been discovered and reported in [1] and [2]. To overcome these threats, some researches proposed solutions concerning new *security frameworks* to ensure device authentication, availability and data integrity. Therefore, security framework adoption by manufacturers would help prevent malicious code execution, data alteration and information leakage. For example, the solution presented in [9] employs two modules: the *smart appliance module*, to execute basic functions of the device, and the *appliance integrity module*, acting as a monitor that checks the manufacturer's signature across the components of the device. These modules interact with two frameworks checking access control privileges and execution rights. Overall, this solution provides a sophisticated control mechanism that can be implemented in a single IoT device. This would allow the device to prevent security threats for the end users such as data modification, leakage and fabrication.

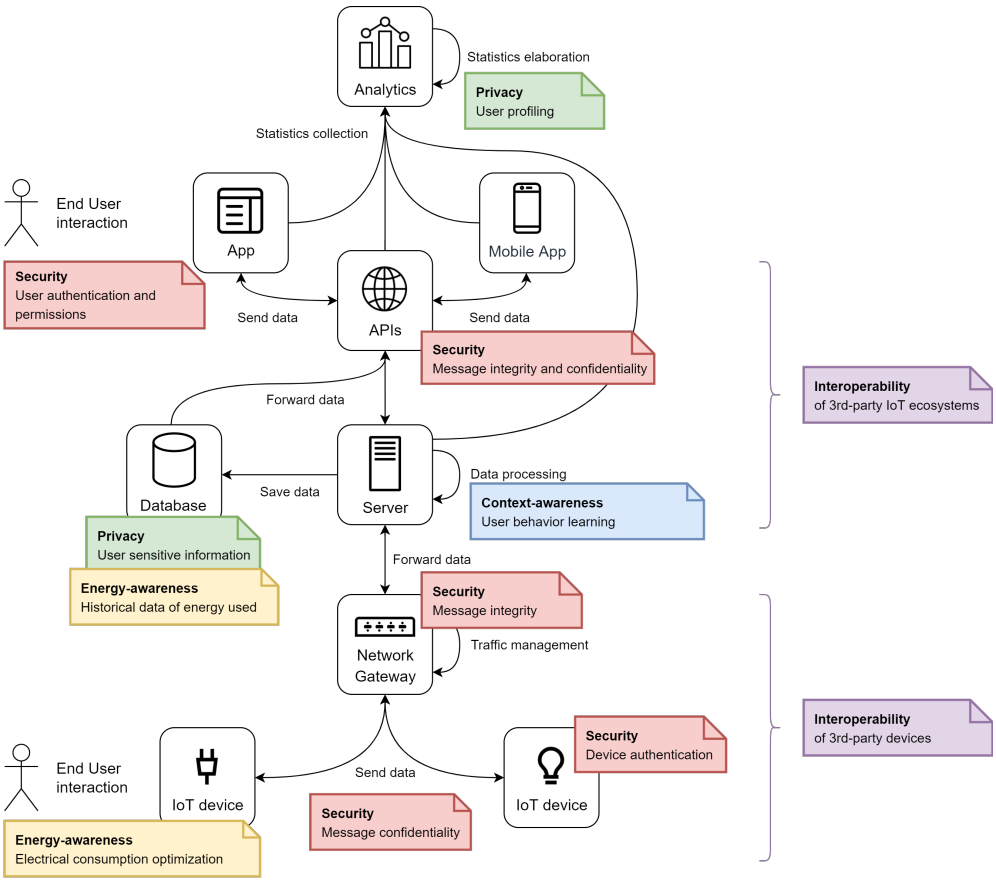


Figure 1.7: IoT challenges affecting the components of a Smart-Home.

As far as **privacy** is concerned, IoT devices are equipped with sensors and actuators

that gather sensitive data from the user through embedded sensors (e.g. wearable devices with heart-rate monitors, smartphones with pedometer apps, etc). This collection of data is processed by service providers, which are capable of profiling a user and re-use those statistics for commercial purposes. Therefore, private data is considerably sensitive for a user and must be addressed correctly to avoid bad impacts from the legal and ethical viewpoints. As an example, a privacy-aware framework has been proposed [2], where each user should be able to define their privacy preferences individually. Hence, this framework provides a complete control to the end user over personal data shared to a service provider.

The **context-awareness** challenge concerns IoT devices placed inside an IoT network and working with different contexts. In particular, context-awareness requires an IoT device to interact with other nodes to adjust behaviors according to changes of the surrounding network, without human intervention. In practice, as an example, a home gateway should be capable of collecting interpretable data from IoT devices. Then, the gateway should learn user behavior in order to be able to take appropriate decisions depending on context. To achieve this, a query mechanism among devices is required to enable any node to interrogate any other node to acquire information. For example, this process can be realized through information exchange that is expressed using the Web Ontology Language discussed in [10]. Moreover, by using a platform with an appropriate reasoning algorithm (e.g. SOCAM), it would be possible to infer high-level contexts from the available low-level context, so that everything is translated to a service (e.g. sleeping, watching TV).

The **energy-awareness** challenge is also actively discussed nowadays. As shown in [21], the energy price increased due to factors such as supply, demand and government regulations. In addition, the consumption of electricity increased as well, affecting environment (e.g. non-renewable electrical sources) and family budgets. As reported in [2], by the end of 2040 the 13% of the world energy consumption will be energy used in homes. Therefore, in the smart-home context, home appliances and IoT devices should minimize energy consumption and waste. To address this problem, several researchers suggested different approaches and solutions. First of all, the authors of [25] proposed an *Energy-Prone Context* (EPC) system to model multiple contexts and their associated energy consumption. This would help acquire historical data of each appliance regarding power consumption with respect to the context. Another work concerns *IntelliHome*, which is discussed in [21]. *IntelliHome* is a framework to reduce energy consumption at home by using data analytics and Machine Learning techniques. The idea is to apply Machine Learning to collect user habits and provide real-time consumption, as well as energy savings recommendations. This approach is more social with respect to the previous solution, and the aim is to actively involve the user in the energy-savings process.

In conclusion of this section, we can say that many new challenges are appearing with the growing number of IoT devices used in different domains. Smart-home is an active application domain in which transparency is the key aspect for consumers and interoperability is the key aspect for manufacturers. To address these aspects, a solution that works for different device vendors must be adopted. As a matter of fact, this solution acts as an intermediary that is able to understand different data formats and semantics to accomplish data transport across foreign entities. Moreover, this intermediary can be associated to a more powerful concept called *middleware*.

1.4 The role of middleware

As discussed in §1.3.3, IoT challenges must be actively investigated to find viable solutions. In this thesis, we introduce the concept of middleware as a support component for the IoT architecture presented in §1.3.1. In general, a middleware is a powerful entity that has multiple definitions. Around the context of software development, a middleware is a component that fits (like *glue*) in the middle of other two components. More specifically, a middleware is used as an *intermediary* between independent software components, whose purpose is to help exchange data through a common interface. The term middleware appeared originally in a report from the NATO Software Engineering conference in Garmisch-Partenkirche (Germany) in the late 60's [24]. It was defined as a solution to provide interoperability of service routines, which were expensive to adapt every time in newer application programs.

Over time, various middleware solutions have been used to link legacy services in old operating systems to newer applications. This helped developers to work on the high-level application disentangling themselves from the complexity of low-level ad-hoc programming. Moreover, low-level components become re-usable for high-level implementations. Let's take for example a microservice architecture, which breaks down an application in a collection of independent services. As shown in [86], a simple microservice architecture has a back-end component with its database, a front-end component and an Application Programming Interface (or API). This API can be embedded with the back-end component and it is in fact a middleware, which operates as an intermediary between front-end and back-end communication (see figure 1.8).

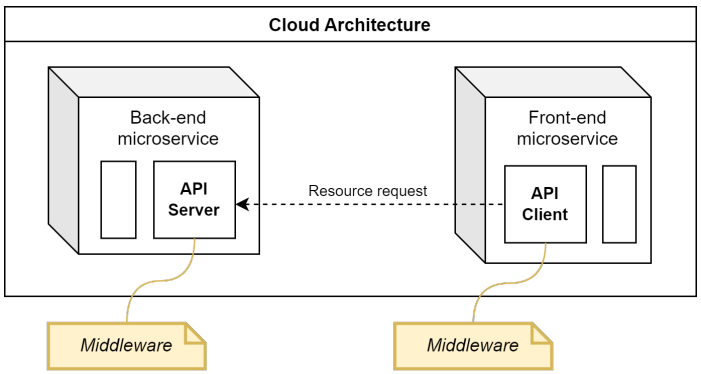


Figure 1.8: Example of a microservice architecture using an API to request a resource. The API Client act as a middleware for the front-end microservice, while the API Server act as a middleware for the back-end microservice.

Nowadays, this term is used in a huge variety of application software, as well as Cloud and IoT architectures, according to respective meanings, as shown in [87].

In **cloud software development**, a middleware is a component that can help developers to gradually transform monolithic applications into Cloud-native applications, through the *Strangler Fig pattern*. As shown in [79] and [80], this pattern is applied by expanding a monolith application with a new microservice application, both using a temporary API middleware. This middleware provides a common interface (i.e. Strangler Façade in Figure 1.9) that keeps using the old monolith and the new microservice cloud-native app. Developers need to gradually shift functionalities from the old monolith to the new microservice by eliminating pieces of software. As soon as

the microservice become self-sustained, the middleware is not used anymore and the old monolith can be removed entirely.

Speaking of **cloud architectures**, a middleware is used to provide integration among services through resource-centric APIs or data-centric APIs, connecting different products. Furthermore, middleware solutions are employed to provide automation for manual decisions in multiple stages of software development. This enhances the overall efficiency and gives a remarkable improvement in resource management.

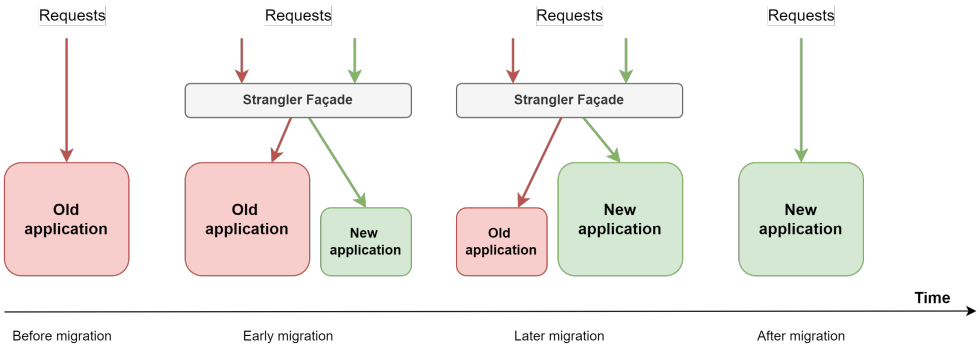


Figure 1.9: The Strangler pattern employed in the migration of a monolith app. The Strangler Façade is an example of a middleware helping the migration.

When it comes to **IoT**, recent literature addresses middleware solutions in different class types of architecture. The class types purpose is to compose IoT applications to perform data collection and analysis without any low-level programming needed, through the use of abstractions. In [19], three architecture class types are defined.

- The first type refers to a *service-based* solution, similar to a Service-Oriented Architecture (SOA), where a single IoT device is identified as a *service*. This architecture is also the one chosen by *OpenIoT* [26], a European Union project that aims at standardize IoT platforms. As shown in Figure 1.10, there are three components: a physical plane (with sensors and actuactors), a virtualized plane (i.e. Cloud infrastructure) and a utility / application plane. The idea is to provide the computational power inside the virtual plane, where data and events are processed, and not on physical devices.

- The second type is called *Cloud-based*, which allows collecting and processing raw data only with the deployment of the middleware on remote nodes (generally in-cloud) or powerful local gateways. Functionalities are exposed via APIs and can be accessed widely by 3rd-party Cloud platforms. However, this architectures needs to adopt specific standard protocols to enable devices inter-operation (e.g. HTTP REST APIs). Moreover, the middleware can stop working whenever the Cloud provider ends the service.

- The third type is called *actor-based* solution. This architecture is divided in three layers: the *Sensory Swarm layer*, made of sensors and actuators; the *Cloud layer*, providing cloud services for the other two layers; the *Mobile Access layer*, providing applications and services for smartphones and laptops. In each layer, a light-weight middleware called *actor host* is used to provide computational power through actors. These actors are pluggable computational units placed in an actor host. Therefore, this solution allows computation to be performed across

the entire architecture where it is most beneficial. Moreover, this architecture embraces heterogeneity of IoT devices by using device abstractions, without the need of a standard technology.

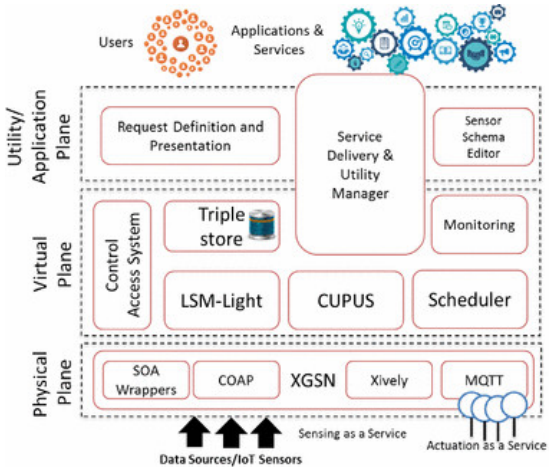


Figure 1.10: OpenIoT view representing an example of a Service-Oriented Architecture. *Source:* [8]

Nevertheless, middelware's definition for *IoT ecosystem* can be characterized differently with respect to previous architecture class types. As a matter of fact, commercial IoT ecosystems do not reflect these architecture models entirely, because each service provider design its own ecosystem based on its needs. In this thesis, an IoT middleware is an essential part acting as a mediator, which understands one interface, applies logic and computation, and then translates the computed result in the other interface's language. Therefore, an IoT middleware must not be addressed just like *glue* between two components, whose main purpose is to pass data from one interface to the other. Consequently, an IoT middleware must be able to interpret data symbols retrieved from one interface and extract information. Then, information is used to recreate data with the other interface's data format. This way, the IoT middleware adapts data from one format to the other, while preserving information.

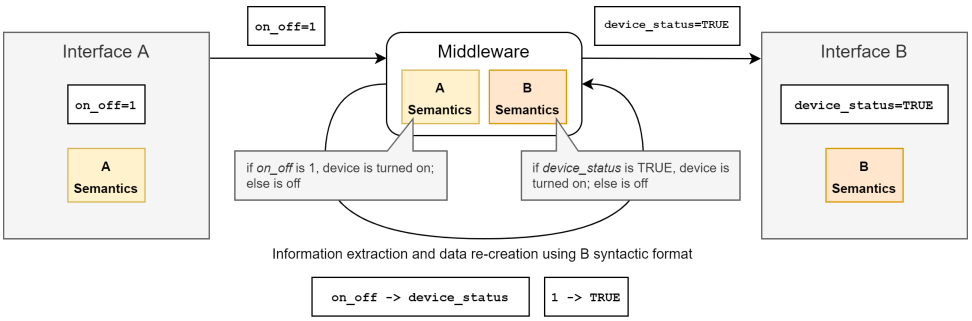


Figure 1.11: Example of data processing in a middleware component handling two interfaces. Interface A sends data to the middleware which is forwarded to interface B after being processed with A and B semantics.

To clarify this concept, Figure 1.11 shows an example with a middleware connected to two interfaces. In this figure, interface A passes data (i.e. `on_off=1`) to the middleware. Then, the middleware extracts information from data by interpreting data symbols (e.g. `1` is the symbol). Semantics from interface A is used to express the meaning of information items. Then, the middleware recreates data using the syntactic format for interface B, according to interface B semantics. Lastly, the middleware passes data (i.e. `device_status=TRUE`) to interface B.

As a result, information is preserved (i.e. the device is on) because the middleware understands both interfaces, interprets symbols, extracts information and then re-creates data for a different data-format and semantics. In conclusion, an IoT middleware can be deployed in different forms, depending on the IoT architecture, and it is capable to interpret and process data in different use cases, thus providing interoperability in foreign components.

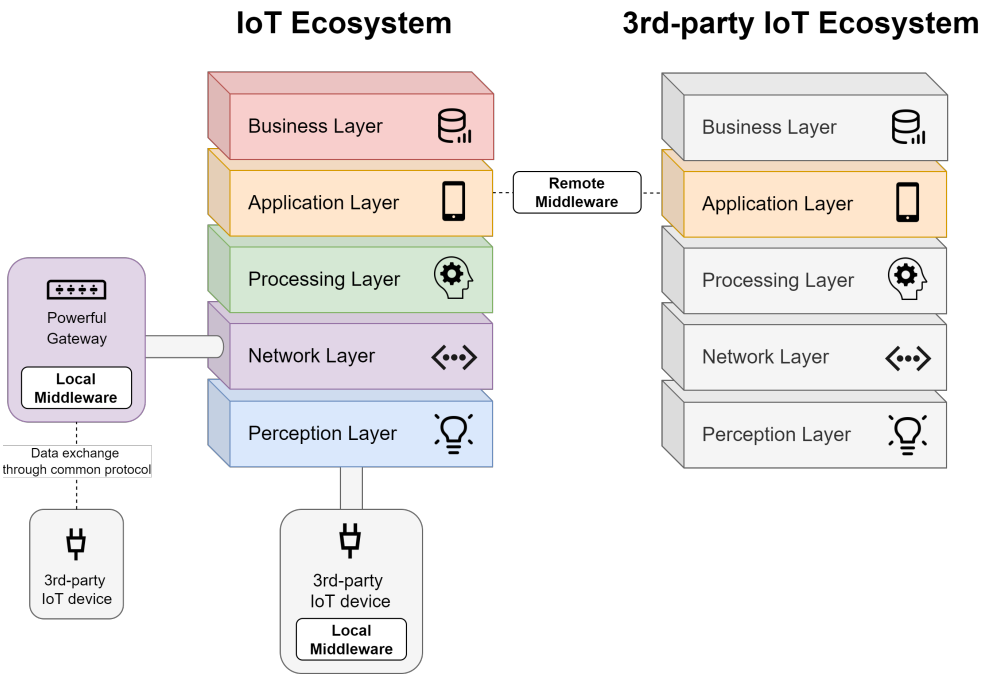


Figure 1.12: Middleware solutions proposed for 3rd-party integrations in an IoT ecosystem.

In order to achieve interoperability, an IoT architecture must be equipped with a component that is able to provide a common interface for a 3rd-party device or a 3rd-party IoT ecosystem. A **3rd-party device** is usually an IoT device from a 3rd-party manufacturer, which is integrated at the lowest level (i.e. perception layer) of a foreign IoT ecosystem. In terms of benefits, this approach helps in satisfying user needs of interoperability and does not require the creation of a Cloud component for the IoT ecosystem by the 3rd-party manufacturer. Conversely, the control over the foreign IoT ecosystem is very limited. For example, the UI for the end users is not always customizable according to the 3rd-party vendor's needs. In the **3rd-party IoT ecosystem** integration, the 3rd-party manufacturer adds the entire set of 3rd-party devices in a foreign ecosystem, at the application level. This can be achieved via

Cloud-2-Cloud integration, which requires the foreign Cloud to exchange data over the 3rd-party Cloud. This helps 3rd-party manufacturers to develop the integration on the highest level of abstraction. Consequently, 3rd-party vendors do not expose the low level implementation of their devices. However, any change from the lowest level must be propagated correctly to keep the device working correctly. In addition, control over foreign IoT ecosystem is still limited for 3rd-party manufacturers. In this case, the business layer is also capable in providing statistics and analytics, by having control over exchanged data. Meanwhile, in 3rd-party device integration these statistics can be requested to the foreign IoT ecosystem's owner.

As shown in Figure 1.12, these two integrations can be addressed via two middleware solutions. The 3rd-party device integration happens *locally*, with respect to the network layer. This means that the IoT device must locally interact with a gateway from the network layer, which must be capable – as a middleware – to recognize the foreign device and add it to the ecosystem. Specifically, these last two operations are executed in the processing layer, besides other activities (e.g. device unpairing, etc). Therefore, a **local middleware** component (e.g. driver, plugin) placed under the application layer can be used to provide interoperability of 3rd-party devices. On the other hand, the 3rd-party IoT ecosystem integration happens *remotely*, with respect to the network layer. This means that the Cloud of the 3rd-party IoT ecosystem can create a link with a foreign Cloud to exchange data. This link connects the IoT devices from one ecosystem to the other, thus creating an external conjunction with two architectures. Therefore, this integration can be realized at the application layer using a set of APIs of each respective Cloud. In this case, a **remote middleware** will be a component (e.g. microservice, function) interacting with both APIs to understand and elaborate queries across two IoT ecosystems.

1.5 Objectives of this work

In the context of smart-home IoT ecosystems, several challenges – discussed in §1.3.3 and §1.3.1 – need to be addressed to satisfy user needs and concerns. One essential problem that is transversal to the others in §1.3.3 is **interoperability**. In fact, many smart-home users adopt more than a single ecosystem, because not always their needs are satisfied by a single manufacturer. Therefore, open issues like security and privacy, can be considered vertical issues. This means that the scope of these problems can be limited within a single IoT ecosystem. Conversely, interoperability can be considered an horizontal issue for 3rd-party integration. As a matter of fact, interoperability requires data exchangeability and common protocols adoption by manufacturer to expand IoT ecosystem with 3rd-party devices. Therefore, an IoT architecture must be equipped with a component that is able to provide a common interface for a 3rd-party device or a 3rd-party IoT ecosystem. As discussed in §1.4, **middleware solutions** can be used to achieve interoperability via a *local* or *remote* integration, respectively. These solutions are placed within critical components to provide a way to communicate and interpret data on different levels of the IoT architecture.

Nevertheless, interoperability is not the only concern for manufacturers. When a manufacturer designs its IoT ecosystem, many decision-making factors come into play in support of interoperability.

- At first, a vendor must address which **protocol** should be adopted to seek best interoperability with 3rd-party devices. However, not always the adoption to a standard protocol is the best choice to make. For example, a vendor might

want a specific functionality that cannot be achieved by the standard. On the other hand, disregarding conformance to standards can reduce the number of total installations.

- Secondly, the **performance** of the protocol must be considered with regards to response times, signal coverage and power consumption. For example, some sensors use batteries and so just a few technologies fit the needs of high energy efficiency. Conversely, sensors that do need a direct power source are less portable.

- The third factor is the **development cost**, along with the **maintenance cost**. By choosing a particular type of solution, vendors must plan a set of tests for 3rd-party devices, so that the integration is reliable for a long time. Moreover, there are multiple points of failure in Cloud integrations as compared to direct communicating devices. As a matter of fact, Cloud integration requires a 3rd-party vendor to support its solution the entire time, whose reliability is determined by a high Service-Level Agreement (SLA).

- A fourth factor concerns the different **capabilities** of devices to support. Each type of device might require a common portion of functionalities along with different specific features, provided by 3rd-party manufacturers. This is crucial to handle, hence an inner semantic is opportunistically implemented. This way, multiple capabilities owned by the IoT device are correctly registered and enabled.

- Lastly, an **IoT semantic** (along with a corresponding ontology) is the fifth factor to take in consideration when realizing an IoT ecosystem. Based on a particular semantic, the entire IoT device's profile of can be defined in classes, attributes and relations, which are understandable to 3rd-party integrators.

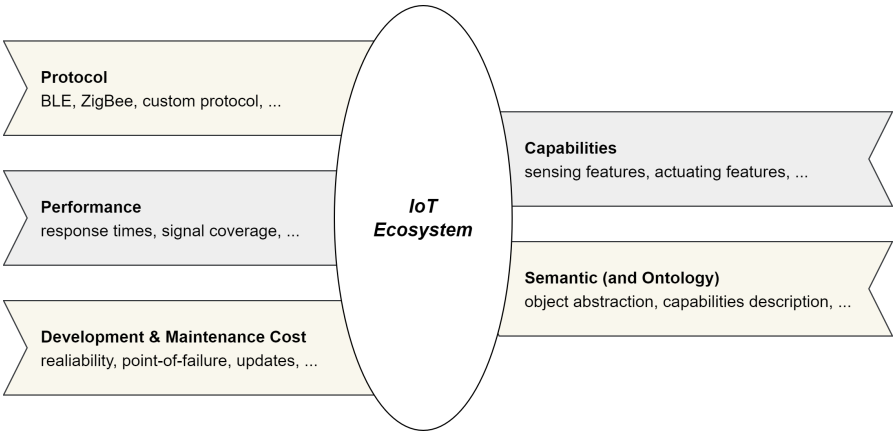


Figure 1.13: Decision-making factors to design an IoT ecosystem.

To conclude, the first objective of this thesis is to explore different real-world 3rd-party integrations and use cases through the concepts of local and remote middleware solutions, as defined in §1.4. Secondly, interoperability is analyzed for each solution by taking into account different design choices used in IoT ecosystems (i.e. hardware and software requirements for 3rd-party manufacturers, protocols, offline control and computational power). Thirdly, this work analyzes the process of a 3rd-party integration

in a commercial IoT ecosystem, exploring a local middleware solution with an industry-standard protocol and a remote middleware solution based on a Cloud interaction. Eventually, decision-making factors of the 3rd-party integration are challenged against recent literature solutions.

Chapter 2

Candidate solutions

2.1 Commercial solutions: an overview

In recent years, smart-home ecosystems have been commercialized by several companies working on home automation and IoT. These companies aim to integrate their devices with value-added services for the end users. This directly involved enterprises to develop IoT applications under realistic conditions. Furthermore, the market demand has led to more vertical than horizontal responses. This means that companies used to focus on improving their ecosystems rather than making their ecosystems interoperable with competitors, due to typical monopolistic trends of large companies. However, in recent years, article [49] explains that collaboration – i.e. an horizontal approach – with external partners has gained success to reach new customers and client needs. Consequently, cooperation led to a greater competitive advantage by realizing stronger IoT ecosystems for a wider audience.

Companies focus their business model in understanding the key players to realize an IoT ecosystem. The authors of [14] look at these players in the context of enterprise IoT, which is a different sector from smart-home IoT. Enterprise IoT focuses on all connected devices used for various business purposes in the enterprise setting (e.g. industries, receptive structures, etc.). Although the cited publication does not refer to smart-homes explicitly, the authors of [18] described similar key actors employed in a non-specific IoT context:

- **Platform providers** (e.g. data platform developers, cloud service providers), whose purpose is to design, develop and maintain the IoT platform software-wide;
- **Device vendors**, i.e. *hardware platform providers* (e.g. board manufacturers, device manufacturers), dealing with the IoT devices for the IoT ecosystem;
- **Application providers** (e.g. app developers, system integrators, data analysts), working on the device capabilities, the end-user apps and analytics for the IoT ecosystem;
- **Infrastructure providers**, i.e. *network technology developers* (e.g. telecom companies, data network developers), providing connectivity and protocols to the IoT platform;
- **Users** and *customers*, who are the ones using the actual IoT solution.

Figure 2.1 shows the key actors enumerated above. These players are essential to create an IoT ecosystem capable of providing an IoT device to the end user from the hardware part to the applications. However, most manufacturers cannot afford internally all of these players by themselves due to high costs. Therefore, some manufacturers can only provide IoT platforms, IoT apps or IoT devices to other manufacturers. This way, proprietary IoT ecosystems adoption grows thanks to company partnerships with device vendors and external players, as shown in [49]. Consequently, an IoT platform with a higher number of installations and services attracts more customers.

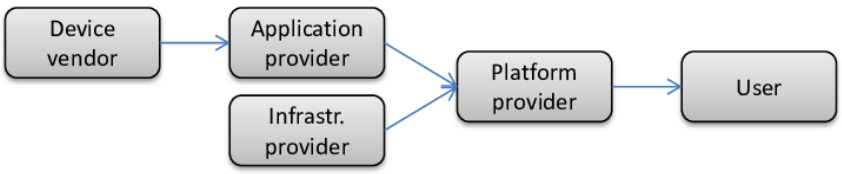


Figure 2.1: Main players working for an IoT ecosystem.
Source: [18]

Nowadays, large companies have all required players to build their IoT ecosystem, while being able to create partnerships with 3rd-party manufacturers. In fact, large enterprises have a big influence in the tech market around the world, being the most profitable companies at present time. As shown in [95], these companies have experienced an enormous growth in terms of innovation and business model. Among them, Google [74], Amazon [72], Apple [73] and Samsung [71] are the ones with an own, fully-vertical, proprietary smart-home IoT ecosystem. Each of these companies provides interoperability solutions for 3rd-party manufacturers in order to expand their IoT ecosystems with 3rd-party devices and 3rd-party IoT ecosystems from minor vendors in the global market.

In the next sections, we report in a detailed analysis the online available solutions of Google, Amazon, Apple and Samsung to capture the different approaches for IoT device integration. These solutions concern Smart-Home integrations for manufacturers, aiming at interoperability of 3rd-party devices. Moreover, these platform providers are interesting to analyze due to their commercial influence in the Smart-Home market. Therefore, we analyze the different ways to realize an integration with the corresponding pros and cons. For each IoT platform we discuss four characteristics:

- *Integration solutions*, i.e. how each vendor decided to open its IoT ecosystem to 3rd-party manufacturers. Depending on the solution, a 3rd-party manufacturer must meet different hardware or software requirements to proceed with the integration.
- *Protocols*, i.e. standard and non-standard network protocols employed in the IoT integration, with their respective pros and cons in terms of interoperability.
- *Offline control*, i.e. whether an IoT device can be controlled when the Internet connection or Cloud connectivity is absent. This helps to understand how the architecture works in terms of services, components independence and computational units.
- *Computational power* of architectural components, i.e. the computational units in which data is received, processed and transmitted (e.g. powerful gateway,

server). We want to identify where and how data is processed across different components of the IoT architecture.

Then, in this chapter we compare and classify each commercial solutions with the above characteristics.

2.1.1 Google Home

Ecosystem overview. Google is a huge company headed by Alphabet. The main commercial ambit of this company regards to internet services such as the famous Web Search Engine (i.e. *google.com*), Cloud solutions (i.e. *Google Cloud*) and smart-home solutions (i.e. *Google Home*). Around smart-home solutions, Google can be considered a manufacturer that has been able to reach the market through entertainment systems, i.e. Chromecast, used for game and video streaming, and smart speakers, i.e. Google Nest, used for voice-controlled virtual assistants. Smart speakers become particularly popular especially for home-related usage. These speakers are able to interpret user questions and provide a response in different contexts like music, weather information and cooking recipes. They also work as vocal user interfaces to control home appliances. By using smart speakers a user can control IoT devices and perform task routines, as shown in [38].

Device integrations. In order to integrate 3rd-party devices in Google Home ecosystem, Google provides a certification program called *Works with Google Home*. In this program there are three different ways to integrate new devices:

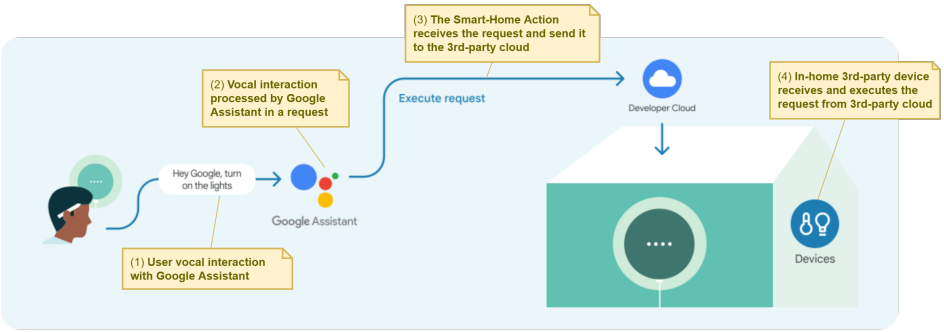


Figure 2.2: Cloud-2-Cloud integration in Google Home.
Source: modified with labels from [34]

- In **Cloud-to-Cloud** integration [34], Google provides its APIs for 3rd-party cloud integration. This type of integration requires 3rd-party developers to use their own platform that should work independently from Google. As shown in Figure 2.2, a user perform a vocal interaction with Google Assistant (1). This interaction is translated in a *text* request by Google Assistant (2). Then, an intermediary component – called *Smart Home Action* (3) – interprets the text request and send it to the 3rd-party Cloud. Lastly, the targeted 3rd-party IoT device executes the request received from the 3rd-party Cloud through the Internet (4). This process requires the user to authorize external operations (e.g. device installation, device control) on its 3rd-party devices from a foreign Cloud. Therefore, the user must be registered in both IoT platforms (i.e. Google smart-home platform

and the 3rd-party platform) and both accounts must be linked. In order to achieve authentication, OAuth 2.0 is employed, which is an industry-standard protocol to provide authentication flows and permissions for web applications, machine-to-machine interactions and also IoT devices (see [82], [64]). In details, machine-to-machine interaction provides autonomous operations to exchange data (e.g. in OAuth, security credentials) between two or more computing entities (e.g. web server, web applications) without human intervention.

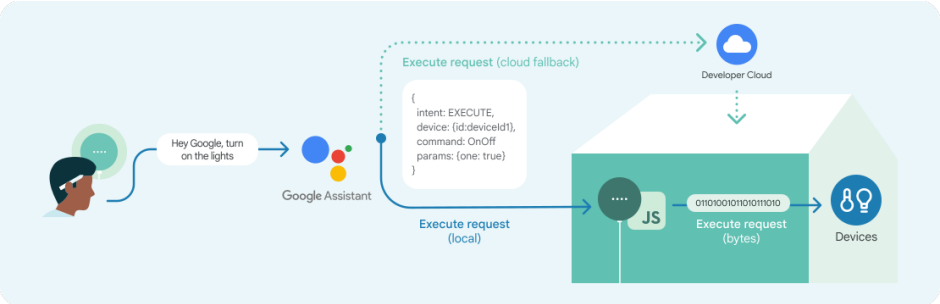


Figure 2.3: Local Home SDK integration in Google Home.
Source: [57]

- In **Local Home SDK** integration [57], Google provides an SDK, i.e. a Software Development Kit, to add 3rd-party devices with local control capabilities, without demanding the entire action to the Cloud. This way 3rd-party devices can interact directly with Google Home devices via intents. An *intent* is a simple messaging object – i.e. data – that describes what action to perform like turning off a light or streaming audio to a speaker. As illustrated in Figure 2.3, a user sends a request to Google Assistant, which is then translated to an intent. The execution is demanded to Google Home devices sending the request to 3rd-party devices nearby in a local network. In case of local execution failure, the request can be processed via the 3rd-party Cloud using a fallback path. This approach helps avoid Single-point of Failure (SPoF) whenever a Google Home device is unreachable or cannot process the request. In addition, offline execution is not supported in this integration. This means that if the internet connection is temporarily absent, the request will not be satisfied by the device.

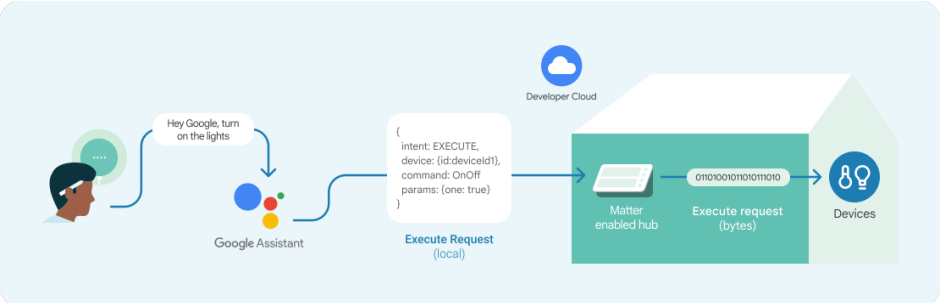


Figure 2.4: Matter integration workflow in Google Home.
Source: [59]

- In **Matter**[1] integration [59], Google provides full compatibility with Matter, which is an emerging industry-standard provided by the Connectivity Alliance Standards (CSA). The official website [33] declares that Matter aims at resolving security and interoperability issues with heterogeneous IoT devices. Nonetheless, at the time of this writing Matter is not available to the market. As shown in Figure 2.4, using Matter, an intent from a user is produced and executed through a Matter-compatible Hub connected to nearby 3rd-party devices. This Hub receives the request from the 3rd-party Cloud, that must be linked with the Google Cloud sending the request. The Hub is entirely responsible for the execution request, thus lacking a different execution path to use as a fallback in case of failure. Moreover, this approach does not provide any offline control of an IoT device, as an Internet connection is required.

Discussion. These integrations provide multiple approaches for 3rd-party integration in the Google Home ecosystem. Speaking of **integration solutions**, the Cloud-to-Cloud solution takes advantage of 3rd-party Cloud to handle execution request, while the Local Home SDK exploits Google Home devices to send execution requests to nearby 3rd-party devices. On the other hand, Matter's integration requires a 3rd-party Hub that must be Matter-enabled. In addition, the 3rd-party device shall be compatible with Matter and the Hub. Lastly, using Matter the entire interaction can be performed through the 3rd-party Cloud. In all integrations, *offline control* of IoT devices is not supported, which leads to a huge dependency to Cloud-related services for *computational units*. As a matter of fact, Google Home takes advantage of the Cloud computing power for the Cloud-to-Cloud integration, while Local Home SDK and Matter integrations delegate parts of the computation via an intermediary. This intermediary works as a *powerful gateway*, as previously described in §1.3.2, which can be a Google Home device or a Matter-enabled Hub. As a result, Local SDK integration requires the use of an extra component – similar to a *driver* – that is installed inside Google devices, while Matter integration requires a protocol-compliant IoT device without extra components.

2.1.2 Amazon Alexa

Ecosystem overview. Amazon is a huge enterprise covering several commercial sectors. It started as an e-commerce company in 1994, and today has expanded its business to cloud computing, digital streaming, and Artificial Intelligence applications. Among these sectors, smart-homes and smart-speakers reached a high market share in the US and worldwide, as shown in [28] and [63]. The first Amazon solution with smart-home was introduced with *Alexa* which is a virtual assistant embedded inside Amazon smart-speakers. Similarly to Google Home assistant, Alexa enables 3rd-party IoT devices to be controlled in the Amazon smart-home ecosystem via voice control and mobile app. The core processing component of Alexa are skills. An *Alexa skill* is a set of tasks or actions that are executed by various executors all in the Amazon ecosystem. In other words, skills may help users to perform everyday tasks naturally through voice-recognition. From a developer viewpoint, a skill is a set of serverless lambda functions. In programming languages, a *lambda function* (also called *anonymous function*) is a function definition that is not bound to an identifier. Moreover, a function is called *serverless* because it is dynamically executed by the cloud provider through on-demand resource allocation. Consequently, in the *Function-as-a-Service* (FaaS) context, an

[1] Matter is formerly known as *project CHIP*, Connected Home over IP

application can be realized through the composition of multiple serverless lambda functions, whose purpose is to handle events passed as arguments during function invocations. Furthermore, FaaS is also provided by Amazon Web Services (AWS) – i.e. Amazon Cloud platform – via *AWS Lambda*, that hosts Alexa Skills.

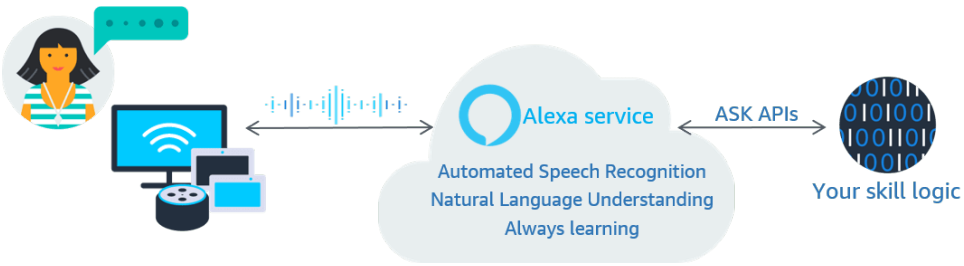


Figure 2.5: Amazon Alexa skill overview interacting with the Alexa service.
Source: amazon.com

Device integrations. In order to integrate 3rd-party devices in Amazon smart-home ecosystem, Alexa provides a certification program called *Works with Alexa* (WWA). As shown in [67], this program allows developers to integrate their IoT devices in four different approaches. However, one approach is not considered in this analysis because it explains how to create an Alexa built-in 3rd-party client, which concerns only a subset of IoT devices integration – i.e. smart-speakers. Therefore, three integration approaches are analyzed.

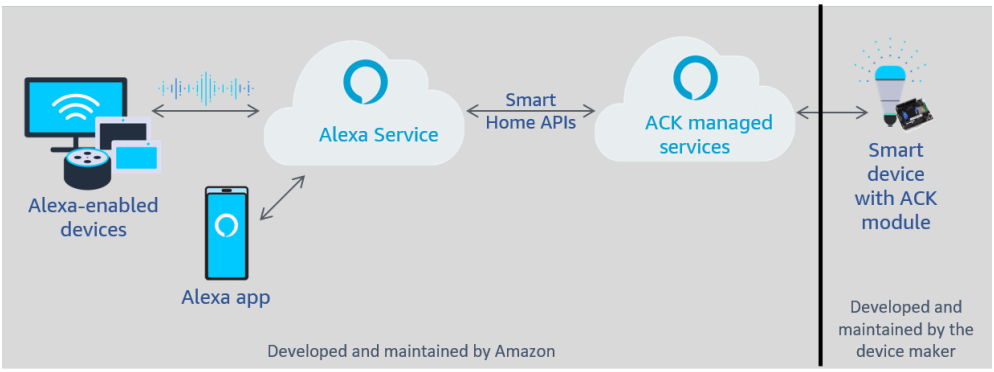


Figure 2.6: Amazon Alexa integration via Hardware / SDK module.
Source: [67]

- In **hardware or SDK module** integration, a 3rd-party manufacturer shall use an ACK – i.e. Alexa Connect Kit –, which includes a required SDK library and an optional hardware module to directly integrate 3rd-party IoT devices in the Alexa ecosystem. The hardware solution is suitable for newer devices, while existing devices requires a firmware update to add the required SDK, if compatible. As shown in Figure 2.6, the IoT device is developed and maintained by the 3rd-party manufacturer, who is responsible for the entire device life-cycle. On the other hand, the entire IoT platform is developed and maintained by Amazon. This does not allow 3rd-party manufacturers to have control on their

device with their own IoT infrastructure, hence firmware update management, metrics and logs are operated by Amazon.

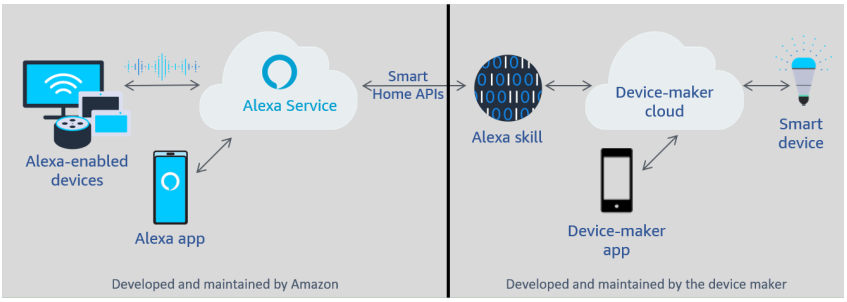


Figure 2.7: Amazon Alexa integration via Cloud-to-Cloud.
Source: [67]

- In **Cloud-to-Cloud** integration (called *Smart Home Skills*), a 3rd-party manufacturer uses its own IoT infrastructure connected to the Alexa IoT platform. This integration requires an Alexa Skill interacting with Alexa Smart Home APIs and a 3rd-party Cloud. Therefore, the 3rd-party manufacturer must use its Cloud to receive actions and send proactive device updates. As shown in Figure 2.7, the 3rd-party IoT infrastructure is completely developed and maintained by the device maker. Consequently, the manufacturer has more control on its devices, in spite of maintaining its own architecture, which requires the Alexa Skill to be always up-to-date with the 3rd-party IoT ecosystem modifications.

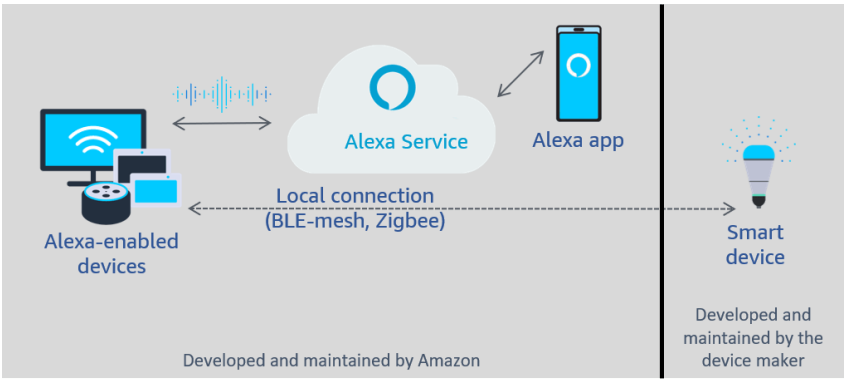


Figure 2.8: Amazon Alexa integration via Local Connection.
Source: [67]

- In **Local** integration, a 3rd-party device is integrated by using a local connection via a standard wireless protocol. In this case, Alexa supports Bluetooth Low-Energy (BLE), ZigBee and Matter to provide a direct connection with an Alexa smart-speaker. However, this integration requires an Alexa smart-speaker enabled to act as a powerful gateway, thus controlling 3rd-party IoT devices via local connections. As illustrated in Figure 2.8, this approach is more Alexa-independent

than the *hardware or SDK module* integration. In fact, the IoT device must use an industry-standard protocol (e.g. ZigBee, BLE) to work with an Alexa-enabled device (i.e. powerful gateways). In this case, we note that the IoT device can potentially work with other IoT ecosystems supporting that protocol. Moreover, in *Local* integration, 3rd-party manufacturers are only responsible for their IoT devices, which work without requiring an Alexa Skill. Lastly, Alexa smart-home provides offline control of IoT devices, meaning that in case of unreachable Cloud services or absence of Internet connection, 3rd-party IoT devices keep working under Alexa control.

Discussion. Regarding *integration solutions*, Cloud-to-Cloud integration leverages the 3rd-party Cloud, while local connection and hardware / SDK solutions can work directly with the Alexa IoT ecosystem without a 3rd-party Cloud. However, the hardware / SDK solution requires a strong joint with the Alexa ecosystem, while local connection requires a standard Alexa-supported protocol to be adopted. Since local connection requires an industry-standard, with this solution a manufacturer can be independent from the Alexa ecosystem. As a result, if the manufacturer intention is to create a 3rd-party device that is interoperable across multiple IoT ecosystems, local connection and Cloud-to-Cloud solutions can be appropriate. Speaking of *offline control*, local connection exploits Alexa-enabled devices to work in absence of Internet connection or cloud services. This is important to provide end users control capabilities in such situations. Moreover, the *computing power* is demanded to powerful gateways working as independent intermediaries for 3rd-party IoT devices. Conversely, Cloud-to-Cloud solution does not support offline control with the Alexa ecosystem and the execution request are processed via an Alexa Skill, acting as a *middleware* for Alexa Smart Home APIs and a 3rd-party Cloud.

2.1.3 Apple

Ecosystem overview. Apple is a global company specialized in consumer electronics, software and online services. A report of July 2022 [95] shows that Apple is the largest technology company by market value, as opposed to Alphabet and Amazon. Apple provides many consumer electronics solutions, especially in the mobile market. In the smart-home sector, Apple owns an emerging IoT ecosystem, made of 3rd-party accessories and devices. As discussed in [30], the entire power of Apple devices resides in the Apple ecosystem, which allows customers to inter-operate easily across Apple devices. Consequently, Apple is highly selective when it comes to 3rd-party manufacturers. Smart-homes are managed through their Apple Home application (see [73]) where users can control IoT devices from different vendors.

Device integration. Apple uses a framework called *Homekit* to provide 3rd-party device integration. For commercial purposes, 3rd-party manufacturers must join the *MFi* program to develop and test their integration. This program gives the manufacturer access to closed documentation regarding the HomeKit Accessory Protocol (HAP) specification. This documentation explains the interaction with Apple ecosystem and 3rd-party integrators. Since Apple policy is not permissive (see [84]), in this thesis device integration will be partially discussed from the available online resources. As shown in [37], Apple supports two main approaches:

- In the **IP / BLE** integration, a 3rd-party manufacturer must use an IP-based family of protocols (e.g. WI-FI) or Bluetooth Low-Energy. 3rd-party devices

shall use HAP to interact with Homekit framework. As discussed in [40] and [46], Apple Homekit exploits an optional HAP-enabled bridge (or Home Hub) that works as a powerful gateway to control multiple HAP-enabled 3rd-party devices, even remotely through the Internet. However, this bridge is not strictly required, because HAP can also work directly with local Apple devices. These Apple devices can be smartphones, tablets and computers providing a user interface (e.g. Apple Home app) for the IoT ecosystem control. Therefore, as shown in Figure 2.9, HAP 3rd-party accessories are able to interact directly with Apple devices (i.e. blue connection lines in the figure) or through a home hub route that connects through the Internet to iCloud, i.e. the Apple Cloud (i.e. green connection lines in the figure).

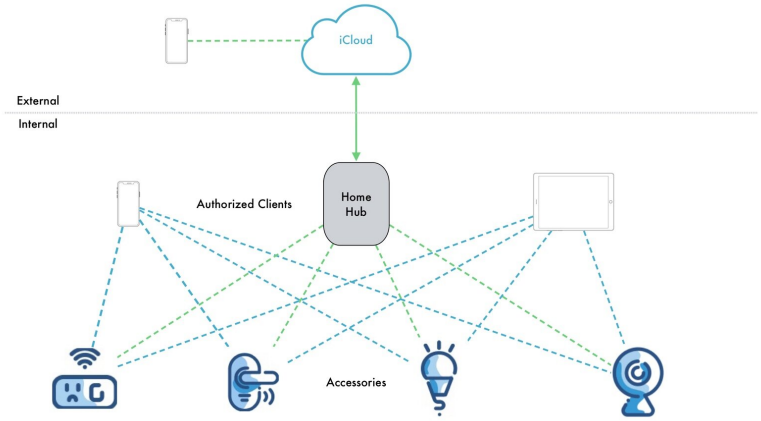


Figure 2.9: High level interaction of Apple Homekit connecting multiple devices.
Source: [40]

- At the time of this writing, **Matter** integration is not fully supported by Apple but it is present inside Apple developers documentation. As shown in [37], [27] and Figure 2.10, this approach uses Matter along with HAP in the Homekit framework. Therefore, a 3rd-party manufacturer using Matter as a standard protocol is eligible to add its IoT device to the Apple ecosystem. As a result, end users shall use the same interfaces of Apple Home used in the IP / BLE approach (i.e. Apple Home app with Apple devices) to interact with 3rd-party IoT devices.

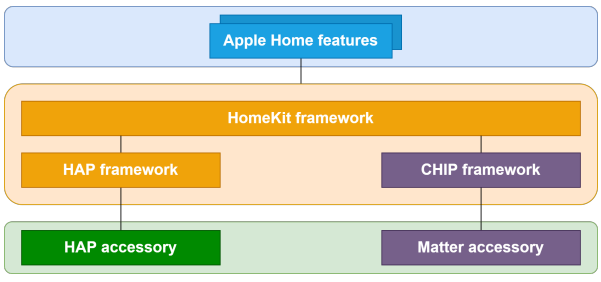


Figure 2.10: Apple Homekit architecture view for Matter support.
Source: recreated from [27]

Discussion. Apple *integration solutions* do not involve 3rd-party Cloud integrations. This requires the 3rd-party manufacturers to completely integrate their IoT device through HAP, which is a custom protocol that works on top of IP / BLE. Consequently, this approach does not require 3rd-party vendors to develop or maintain their own IoT platform in order to manage their IoT devices. Conversely, HAP requires a deep integration in terms of 3rd-party device compatibility. The use of Matter as an alternative approach gives 3rd-party manufacturers more independence from the Apple ecosystem, even though at present times its support is not completely available. Speaking of *offline control*, Apple IP / BLE approach gives end-users the option to control their devices even during absence of Internet connection. On the other hand, Matter approach is not currently documented by Apple. However, Apple integration solutions show a strong independence with Cloud-related services, because no *Cloud-to-Cloud* approach is available for 3rd-party vendors. Regarding *computational power*, 3rd-party IoT devices are handled with a direct connection through smartphones, tablets or powerful gateways. Internet connectivity is only used when a gateway is employed to provide remote control.

2.1.4 SmartThings

Ecosystem overview. Samsung is a multinational conglomerate specialized in manufacturing electronics component such as semiconductors, camera modules and displays. It also provides consumer electronics and services in different fields (e.g. smartphones, tablets and TVs). In this conglomerate of companies, SmartThings Inc. is a subsidiary company working on home automation and IoT technologies. This company has built an entire IoT ecosystem which integrates with Samsung products and 3rd-party devices. An article from [94] shows that SmartThings IoT ecosystem has over 60 million active users in 2020, whose increase was equal to 70% from 2019. As a result, this company concentrated on 3rd-party device integration, gaining success in terms of interoperability. As opposed to Amazon, Apple and Google, SmartThings does not use its own virtual assistant in smart-homes. Instead, SmartThings direction is to make virtual assistants interoperable with their own IoT ecosystem through future Matter integration. Therefore, the IoT ecosystem has several devices from different manufacturers that are controlled via a mobile application for end users.

Device integration. In order to integrate 3rd-party devices in SmartThings IoT ecosystem, the company provides a certification program for 3rd-party manufacturers called *Works with SmartThings* (WWST). In this program there are four ways to integrate new devices, that are illustrated in Figure 2.11.

- In **Cloud-connected device** integration [42], 3rd-party manufacturers use their own Cloud to handle tasks and actions from the SmartThings Cloud. This approach requires 3rd-party devices to autonomously connect to the Internet, receive commands and update their status with their Cloud. As shown in Figure 2.11, the SmartThings Cloud interacts with the 3rd-party Cloud through OAuth 2.0, whose protocol has been discussed in §2.1.1. Then, the 3rd-party Cloud exchange data with the Cloud Connected Device. Consequently, if the Internet connection is absent, the 3rd-party device cannot be controlled. In order to realize this solution, a 3rd-party manufacturer shall provide an intermediary (e.g. a serverless function) that is capable in exchanging data between the two Cloud infrastructures.

- In **Direct-connected device** integration [43], 3rd-party manufacturers use SmartThings Cloud directly without using a 3rd-party IoT platform. This solution enables 3rd-party devices to directly connect through the Internet and dialogue with the SmartThings Cloud, but it requires a custom firmware that uses a compatible SmartThings protocol, such as MQTT (i.e. MQ Telemetry Transport). MQTT is a machine-to-machine network protocol through which messages are exchanged through the one sending the message to a queue (called *publisher*) and the one reading the message (called *subscriber*) from the queue. Messages are organized in queues called *topics* to which multiple publishers and subscribers can hook to exchange data.

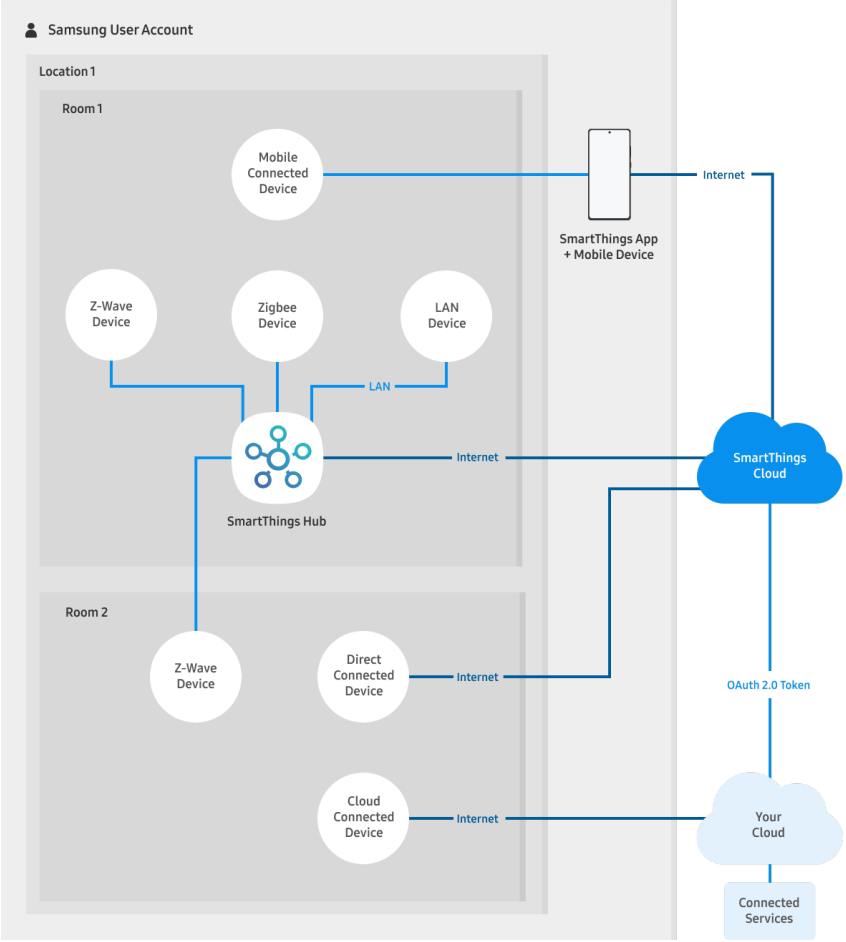


Figure 2.11: SmartThings solutions overview for Cloud-connected, Device-connected, Mobile-connected and Hub-connected integrations.
Source: [31]

- In **Mobile-connected device** integration [45], a 3rd-party device requires an intermediary that connects to the SmartThings platform through the Internet. This intermediary is a mobile device – such as a smartphone – with the SmartThings app installed that connects directly to the IoT device. This solution uses

the Bluetooth protocol to accomplish 3rd-party integration. Some examples of targeted products for this solution can be wearables or headphones that can also be used outside the smart-home context.

- In **Hub-connected device** integration [44], a 3rd-party manufacturer can add its IoT device to the SmartThings platform by using an industry-standard network protocol such as ZigBee and Z-Wave. Moreover, this solution also allows LAN connection to be used through non-standard low-level socket connection provided via TCP / UDP. In addition, even Matter will be supported in the future, as reported in [81]. In order to provide such compatibility, 3rd-party IoT devices are controlled through a SmartThings Hub that acts as a powerful gateway. This Hub uses *drivers* as intermediaries to understand tasks and actions from data received from the network protocol. Then, data is transmitted to the SmartThings Cloud through the Internet. The entire interaction is illustrated in Figure 2.11.

Discussion. Starting with *integration solutions*, SmartThings provides several different ways to integrate 3rd-party devices in its IoT ecosystem. A 3rd-party manufacturer can choose between using its Cloud infrastructure (Cloud connected solution) or not, but it can also decide to integrate a Mobile-dependent device (Mobile connected solution) or a Mobile-independent device (Direct connected solution). Lastly, the 3rd-party vendor can also choose an industry standard protocol (e.g. ZigBee, Z-wave) that works with a powerful gateway owned by SmartThings, along with other home devices. Speaking of *offline control*, each solution requires the 3rd-party device to deal with an Internet connection in order to provide these services. However, the use of drivers in the powerful gateway allows the devices to be partially controlled offline only for routines and automated tasks. This means that an IoT device can be configured to trigger a pre-defined action with other IoT devices belonging to the SmartThings ecosystem (e.g. if a user turn on a light, another light turns on), but the user cannot use an interface that is directly connected to the powerful gateway to control devices offline. Regarding *computational power*, all solutions – besides Hub-connected integration – use the SmartThings Cloud to process requests. In the Mobile-connected solution, the mobile device acts as a powerful gateway demanding part of the operations to the Cloud, as also happens in the Hub-connected solution.

2.2 Drawing a taxonomy of solutions

As discussed in §2.1, Google, Amazon, Apple and Samsung brought several possibilities in terms of 3rd-party IoT device integration. We identified 3 macro-categories in which each integration solution can be classified.

- **Cloud-to-Cloud integration**. This solution requires a machine-to-machine interaction between the IoT ecosystem of the platform provider and the IoT ecosystem of the device vendor. End users can interact with the applications provided by the platform provider or the device vendor. The intermediary between the IoT ecosystems shall be a *software component* (e.g. a serverless function) that is able to extract information from different data formats and semantics. Offline control is not supported in this integration.

- **Gateway-connected integration**. This solution uses a gateway as an intermediary for IoT devices communication. This gateway is also a powerful gateway that can process part of the requests through the use of *drivers*. These drivers

are software or firmware components that are capable in interpreting commands from a network protocol and translate them into another protocol format. In addition, gateway-based solutions can support offline control.

- **Direct device integration**. This solution provides a direct connection with the 3rd-party IoT device to be controlled via a mobile device or the Cloud platform. The IoT device can be connected directly to a mobile device that end users can interface to using a mobile app. Otherwise the IoT device can establish a direct connection with the IoT Cloud platform through a network gateway with no data-processing capability (e.g. WI-FI router). The IoT device integration can be accomplished by using an SDK and/or an hardware component supplied by the infrastructure provider. Offline control can be supported in this integration.

Each commercial solution has been classified in Table 2.1. Samsung and Amazon offers the higher support among the three solutions. Moreover, Gateway-connected integration is supported across all IoT platforms. Another interesting fact is the company approach to support cloud-based solution. Apple focuses on gateway-connected and direct device integration, while Google focuses on the Cloud-powered integrations.

	Google	Amazon	Apple	Samsung
Cloud-to-Cloud integration	✓	✓	-	✓
Gateway-connected integration	✓	✓	✓	✓
Direct device integration	-	✓[2]	✓	✓

Table 2.1: Commercial IoT ecosystems supporting the proposed classification of integration solutions.

2.3 Comparison of IoT integration solutions

In §2.2, we classify commercial IoT solutions in three categories. In the following section, each category is discussed across manufacturers to understand integration solutions.

2.3.1 Cloud-to-Cloud integration

Table 2.1 shows that Cloud-to-Cloud integration is supported by Google, Amazon and Samsung. In this integration, a set of APIs owned by platform providers and/or 3rd-party manufacturers is exposed to provide authentication and services. In details, authentication is provided with the standard-industry security protocol called OAuth 2.0 – already mentioned in §2.1.1 –, while services are accessible through resource-centric APIs. These APIs focus on serving a specific resource, that can be manipulated by a client through basic CRUD operations (i.e. Create, Read, Update, Delete). As a result, in all three alternatives OAuth 2.0 must be employed to provide what it's called *account linking*. This operation is accomplished by end-users to link the account from an IoT platform to a 3rd-party IoT platform. This process requires the user to have an account on both IoT ecosystems. Moreover, it is required to perform machine-to-machine communication at application level. However, in order to make the APIs dialogue with each other, an intermediary is required in each solution.

[2]In August 2022 direct device integration solution from Amazon is restricted to the US market only. See [68] for more information.

- Google employs a *Smart Home Action* as an intermediary, which is a component written in Node.JS or Java. This component is responsible to fulfill user intents. In details, a user perform an action (e.g. vocally, through a smart-speaker) that is represented through data. Then, the Smart Home Action receives data and adapts the data format for the 3rd-party IoT ecosystem. Lastly, the Smart Home Action sends data to the 3rd-party IoT ecosystem using APIs. This process can also work in reverse (that is, 3rd-party IoT platform - Smart Home Action - IoT platform), whenever the interaction is triggered from the IoT device (e.g. a user turns on a 3rd-party light by pressing the physical switch).

- Amazon uses an *Alexa skill* as an intermediary, which is a component written as a Lambda function (i.e. a serverless function) in different programming languages (e.g. Node, Python, etc.). When the user invokes an action (e.g. though an Alexa smart-speaker), the Lambda function is responsible in adapting data with the 3rd-party data format; then, data is transmitted to the 3rd-party Cloud using 3rd-party APIs, similarly to Google Smart Home Action.

- SmartThings exploits a *Cloud Connector* as an intermediary, which works as an HTTP webhook or a Lambda function, written in different programming languages. An HTTP webhook is a web application designed with custom callback APIs, that is employed by the 3rd-party manufacturer to receive data from the IoT platform. This component handles the callback invoked. Then, the business-logic inside the webhook adapts data in the 3rd-party data format. Lastly, the webhook passes data to the 3rd-party Cloud, either via 3rd-party APIs or directly – in this last case, the webhook shall be embedded in the 3rd-party APIs.

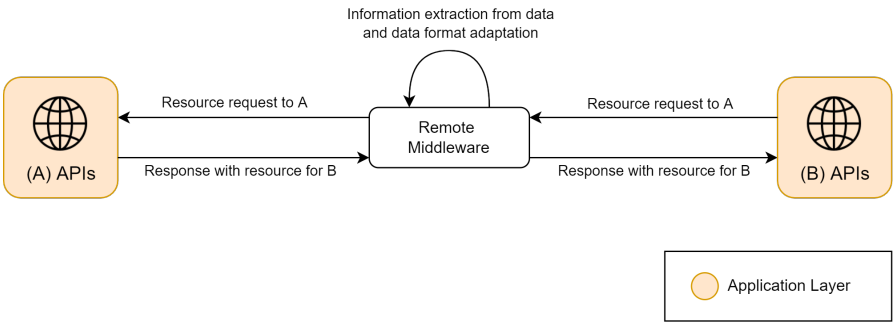


Figure 2.12: Cloud-to-Cloud interaction view between two APIs (A and B) requesting a resource through a remote middleware.

In conclusion, each solution uses the same approach to solve the problem, which is the use of an intermediary software component at an high architecture level. As discussed in §1.4 and §1.3.1, this intermediary can be identified as a **remote middleware** because the Cloud interaction happens between the application level of two IoT ecosystems. Figure 2.12 shows the interaction between two APIs in which a remote middleware extracts information from data and adapts the data format to provide data exchange between the two IoT ecosystems (i.e. A and B).

2.3.2 Gateway-connected integration

From Table 2.1, all manufacturers support this solution. A common point across these manufacturers is the powerful gateway (in §2.1.3 and §2.1.4 called *Hub*), that connects to the Internet and works as an intermediary to handle the connection with multiple IoT devices. Inside the gateway, entities called *drivers* allow 3rd-party IoT devices to be recognized inside the network. Moreover, these drivers cover several basic operations such as IoT device installation, control and disconnection. Nevertheless, the way this solution is employed to 3rd-party manufacturers is a bit different according to gateway design choices.

- Google employs a *Local Fulfillment* [56] to provide local interaction through Google Home devices and nearby 3rd-party IoT devices. This requires 3rd-party manufacturers to develop this Local Fulfillment through the Local Home SDK. In this case, Google Home devices act as powerful gateways because they host Local Fulfillment scripts. Therefore, these scripts act as drivers. However, this solution exploits an Internet connection used by Google Home devices to receive user intents, that are processed through Local Fulfilment scripts. This means that it is not possible to control 3rd-party IoT device when the Internet connection is absent. Lastly, the supported interfaces from Google for this approach are TCP / UDP sockets, HTTP and Bluetooth Low-Energy.

- Amazon uses a *Local connection* with an Alexa-enabled device that acts as a gateway. This solution requires 3rd-party device to implement an industry-standard protocol to recognize the device capabilities and automatically control the device in-app. As mentioned in §2.1.2, the protocols supported are Bluetooth Low-Energy, ZigBee and Matter. In this case, no explicit driver is used inside the gateway because these protocols automatically expose device capabilities through a discovery service mechanism (i.e. device services are exposed by the standard and accessible by the gateway). However, this solution also requires 3rd-party manufacturers to add additional specifications in their IoT device firmware, according to *Works with Alexa* program. For example, in [48] 3rd-party devices do need to add a mandatory component (called *Work with all Hubs*) in ZigBee-enabled devices. Therefore, even though drivers are not explicitly used, extra requirements in the industry protocol are required. This might cause compatibility issues when it comes to multiple IoT ecosystem integration for 3rd-party manufacturers. For example, if the protocol requires an extra functionality that is not completely covered by the standard, this may cause instability in certain IoT ecosystems that do not support it.

- Apple takes advantage of *HomeKit Accessory Protocol* (HAP), which works with an Home Hub to manage 3rd-party IoT devices. With respect to the competitors, Apple approach is the most unconventional because it requires 3rd-party manufacturers to use the HAP. This protocol works on top of IP-based and Bluetooth Low-Energy, which must be adapted in 3rd-party devices. Again, this may cause compatibility issues when a 3rd-party manufacturer wants to add support to multiple IoT ecosystems. However, as shown in §2.1.3, with the introduction of Matter the Home Hub is be able to use Matter independently from the HAP framework.

- Samsung employs an *Edge Driver* to handle 3rd-party devices inside a network. Edge drivers are installed inside a *SmartThings Hub*, that requires 3rd-party

devices to use an industry-standard protocol to interact with. The supported protocols are ZigBee, Z-wave, LAN-based (i.e. UDP/TCP sockets) and Matter[3]. With respect to Amazon approach, Samsung solution allows 3rd-party devices to follow the protocol according to the standard without extra requirements in terms of non-standard compliant components. However, some behaviors of IoT devices can be adjusted and controlled through an edge driver. This driver acts similarly to Google Local Fulfillment, but in addition the edge driver can execute even if the Internet connection is absent. As a result, this approach gives access to custom functionalities at the cost of an extra component that must be developed and maintained.

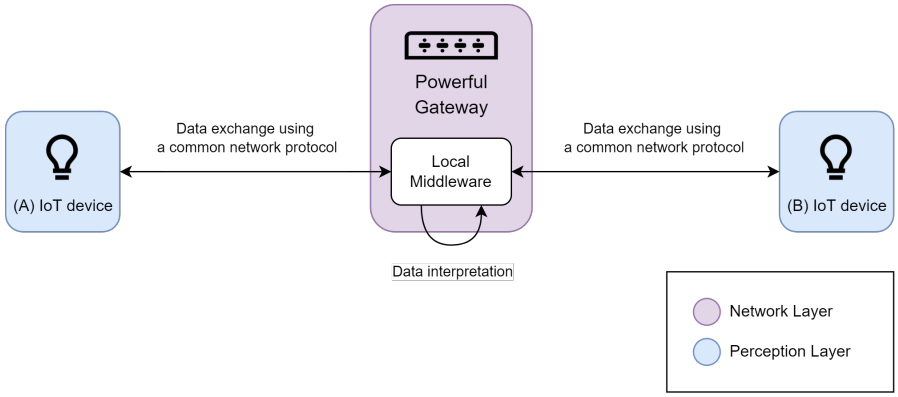


Figure 2.13: Gateway-connected interaction view with a powerful gateway equipped with a local middleware.

In the end, each solution takes advantage of a software or firmware component that works as an intermediary inside a powerful gateway. In the Amazon solution this component is not explicitly present because the interaction from an industry-standard protocol is interpreted internally in the Alexa-enabled devices. Therefore, in all solutions this intermediary can be identified as a *local middleware*, because it works inside the powerful gateway from the network layer. As a matter of fact, the local middleware is able to receive data from a common network protocol (e.g. ZigBee, BLE). Then, data is interpreted and adapted in a new data format understandable by the processing layer. Figure 2.13 represents the interaction described above with two IoT devices from vendor A and vendor B. In this figure, the local middleware is employed before reaching the server to provide data interpretation.

2.3.3 Direct device integration

Table 2.1 shows that only Amazon, Apple and Samsung support this integration solution. This solution uses a direct connection to the end-user IoT interface or the IoT Cloud platform without an intermediary device. As a result, all three integrations shall adapt the 3rd-party IoT device firmware in order to make the integration working. This solution is accomplished in different ways according to the manufacturer.

[3]In August 2022, Matter documentation is not available. However, the support will be added in the future, as reported in [81]

- Amazon uses the *Alexa Connect Kit* (ACK) that includes an Hardware module (optional) and an SDK library to integrate the 3rd-party device. This approach allows the device to be directly connected to *ACK managed services*, enabling the device to be controlled by the Alexa IoT ecosystem. This connection is established with the Cloud through an Internet connection using a WI-FI network. This 3rd-party device must be developed, tested and manufactured entirely by the 3rd-party manufacturers according to Amazon requirements, as shown in [41].

- Apple takes advantage of *HomeKit Accessory Protocol* (HAP) to integrate 3rd-party devices directly, besides the gateway-connected integration. This solution requires 3rd-party devices to be recognized directly through an Apple device using HAP (e.g. Apple smartphones, tablets) and an IP-based or BLE protocol. As a result, this protocol must be implemented in the 3rd-party device firmware to properly work with the Apple ecosystem.

- Samsung exploits two integration variants: a *mobile-connected variant* to connect 3rd-party devices through Bluetooth Low-Energy and a *direct-connected variant* in which the 3rd-party device uses a WI-FI network to connect to the Cloud platform. The former variant (i.e. mobile-connected) requires an *SDK* provided by SmartThings that must be implemented in the 3rd-party device firmware. The latter variant (i.e. direct-connected) requires the 3rd-party device to use the MQTT protocol to exchange messages with the SmartThings ecosystem in Cloud. The MQTT protocol is provided inside a different SDK, equally maintained by SmartThings. In both variants, an ad-hoc firmware called *device app* must be developed to accomplish the 3rd-party integration. As shown in [36], the device app is installed inside the 3rd-party device to provide device functionalities and interpret commands from the SmartThings IoT platform.

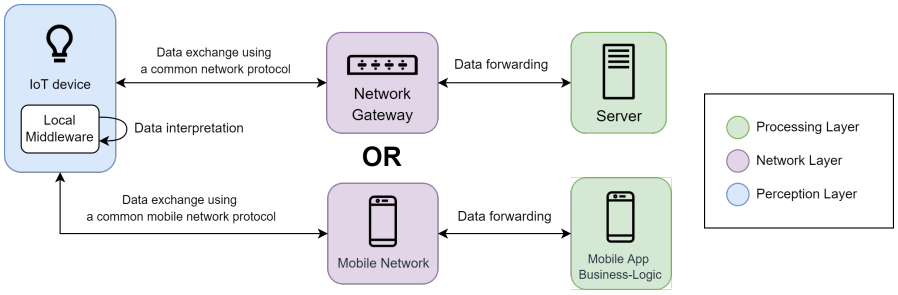


Figure 2.14: Direct device interaction view with either a server or a mobile device.

In these solutions, the intermediary between the IoT ecosystem and the 3rd-party device is not explicitly present, as opposed to Cloud-to-Cloud and Gateway-connected solutions. However, this component is managed by the IoT platform providers in a different form.

- Alexa uses the *ACK SDK module* to translate 3rd-party device behaviors for the Alexa ecosystem. Therefore, since the component works on top of the perception and network layers, this intermediary is a *local middleware*.

- In the Apple ecosystem, HAP interactions are handled through the *HomeKit framework*. This framework works on top of an Apple device connected in the

same network. Hence, the intermediary is a *local middleware*, because it is placed inside the device between the processing layer (e.g. mobile app business-logic, powerful gateway) and the network layer.

- In the SmartThings ecosystem, the intermediary is the *device app* (i.e. the firmware) hosted inside the device. Therefore, this intermediary is a *local middleware*, because it works on top of the perception and network layers.

In conclusion, direct device integration in these three solutions takes advantage of a **local middleware**. Figure 2.14 shows the mobile device approach and the direct connection with the server. Moreover, the connection with the server can happen through the Internet, if the server is only available remotely. In the mobile approach, instead, the mobile network (i.e. the network used by a mobile device to connect to the IoT device) is used in order to forward data to the business-logic of an App. This app is then showed to the user through a front-end interface to provide IoT device control options, belonging to the application layer.

2.4 Protocols in 3rd-party IoT integrations

In this section, we present a summary of industry-standard protocols discussed in the above sections. Then, we give an overview of two protocols used respectively in a Gateway-connected and Cloud-to-Cloud solutions. This overview is required to understand the workflow of real implementation use cases described in the next chapter.

2.4.1 Summarizing industry-standard protocols

Industry-standard protocols have been used across different IoT integration solutions, as discussed in §2.3. These protocols are employed in 3rd-party IoT devices to provide functionalities and capabilities. However, sometimes standard protocols lack additional features that a platform provider might require for the integration. This generates an issue in terms of interoperability because standard protocols implemented in IoT devices must be modified to satisfy platform provider requirements. For example, in §2.3.2 Amazon solution requires an extra component of the ZigBee protocol to be added as a requirement. Another example is Apple, whose integration solution employs a custom protocol that works on top of the standard ones. This way, Apple's HAP is imposed for 3rd-party device integrations, thus requiring higher development and maintenance costs for 3rd-party manufacturers.

	Cloud-to-Cloud	Gateway-connected	Direct
Google	HTTP	TCP/UDP, BLE, Matter	-
Amazon	HTTP	BLE, ZigBee, Matter	TCP/UDP
Apple	-	HAP, Matter	HAP
Samsung	HTTP	TCP/UDP, ZigBee, Z-wave[4]	BLE, MQTT

Table 2.2: Summary of the network protocols used in commercial IoT solutions.

[4] As reported in [81], Matter will be supported in SmartThings gateway-connected integrations. It is not inside this table because there is no official documentation yet.

Table 2.2 summarizes all protocols from the four IoT platform providers. HTTP is the main protocol used for Cloud-to-Cloud solutions. Moreover, Matter plays a key role in all gateway-connected integrations. However, this technology is not yet fully supported by all vendors yet. Therefore, ZigBee, BLE and TCP/UDP are the most common alternatives in gateway-connected integrations. Lastly, direct device integration protocols are different for each implementation.

From this table, it is important to highlight that Matter is commonly adopted by all vendors for Gateway-connected solution – including Samsung in the future. Furthermore, as shown in [33], Google, Amazon, Samsung and Apple took part in a common alliance called CSA IoT to overcome competition in network protocols. This alliance aims at creating *Matter*: an interoperable protocol for gateway-connected integrations. Before Matter, CSA created **ZigBee**, which is a protocol specifically designed for low-powered devices. It gained popularity in recent years in the smart-home domain due to its feature set that covers multiple device use cases and behaviors. This protocol is employed in smart-homes to bring a common standard for gateway-connected integrations.

Nevertheless, in Cloud-to-Cloud solutions no real progress has been made to create a common standard. As a matter of fact, APIs on top HTTP are used across IoT platform providers. Consequently, these APIs are completely customized according to vendor needs. Nonetheless, there is an emerging industry-standard to create 3rd-party APIs in IoT ecosystems. This standard is provided by the KNX Association and it is called **KNX IoT 3rd-party API** [52]. The KNX Association provides services and components to build an entire IoT infrastructure for manufacturers, from the hardware parts to the software. In this case, KNX IoT 3rd-party API – from now on referred as *KNX IoT*[5] – is a software component that allows manufacturers to develop and maintain 3rd-party APIs following an industry-standard protocol. This way, a remote middleware between IoT platforms can be easily re-used across different 3rd-party manufacturers adopting this standard. As a matter of fact, the data format translation would not be required because the data format would be the same. However, the middleware can be used to insert minor adjustments according to expected manufacturers customization.

ZigBee and KNX IoT are two protocols belonging to two different integration solutions, showing two industry-standards for Gateway-connected and Cloud-to-Cloud solutions. Consequently, in this work we analyze these two protocols to investigate main characteristics, data-models and performance (e.g. energy consumption, security). Moreover, as shown in Table 2.2, we consider ZigBee and KNX IoT because the former protocol (ZigBee) is adopted in Amazon and Samsung solutions, while the latter protocol (KNX IoT) is an alternative to custom (HTTP) APIs developed by IoT platform providers.

Furthermore, we analyze in-depth two real implementations from these two standards in the next chapter. One gateway-connected solution is developed for the SmartThings IoT platform, while a Cloud-to-Cloud solution is created for a minor manufacturer.

2.4.2 ZigBee, a wireless protocol for IoT devices

Overview. ZigBee is an industry-standard network protocol used by Amazon and SmartThings in their IoT ecosystems, as discussed in §2.4. This network protocol is used to create a Wireless Personal Area Network (WPAN). In details, WPANs are

[5] We note that KNX IoT includes multiple specifications used for different purposes. In this work, we use *KNX IoT* as an abbreviation of the *KNX IoT 3rd-party API specification*.

networks employed for short-distance low-powered wireless devices. A commonly used WPAN in smartphones is Bluetooth, which works for multiple purposes such as file transfer and audio streaming. ZigBee pertains to WPANs being a low-rate and low-powered network capable to provide interconnection across devices. ZigBee technical specifications are shown in Table 2.3. Moreover, ZigBee has three core characteristics:

- *fail-safe*, i.e. if a node (namely a network-connected device) fails, then the network keeps working with the existing nodes;
- *self-healing*, i.e. the network autonomously replace a node from failure with another node, preserving the previous node role (e.g. if the node that controls network traffic fails, another nearby node takes that role);
- *scalable*, i.e. the network can be expanded or reduced based on the number of devices connected.

These features are important in an IoT context because ZigBee devices consume less power than Wi-Fi or Bluetooth devices, as shown in [15]. Moreover, with mesh networking the ZigBee protocol avoids the use of a central entity to which nodes must be connected to stay in the network and exchange messages.

Frequency band	868/915 MHz; 2.4 GHz
Max N. of nodes	65000 per network
Power consumption	1 mW
Nominal range	10-100 meters
Max signal rate	250 kb/s

Table 2.3: ZigBee specifications reported in [4] and [15].

Network topology. ZigBee is capable to create a mesh network with the connected devices. A mesh network is a particular network topology where nodes are directly and non-hierarchically connected with as many nearby nodes as possible. In this type of network, cooperation is essential to exchange messages through *message multi-hopping*. With message multi-hopping, a node can use other nodes as relays to reach a distant node. For example, since node A cannot directly connect to node B to send a message, node A exploits other nodes in the path.

In PANs, each node has a specific role. There are three main roles, that are also used in the ZigBee specification:

- **Full Function Device** (or ZigBee Router), which is a device with a basic communication model used to send and receive messages from other nodes.
- **PAN Coordinator** (or ZigBee Coordinator), that is a special type of Full Function Device, whose purpose is to coordinate message exchange across the entire network. Moreover, it is unique inside the network and it is responsible to create the network. When a PAN Coordinator fails, the node is replaced with a Full Function Device inside the network.
- **Reduced Function Device** (or ZigBee End-Device), which is a simpler device that requires a Pan Coordinator or a Full Function Device to send and/or receive messages inside the network.

As shown in Figure 2.15 – which is based on [4] –, in the ZigBee Mesh network a ZigBee End-Device is connected to a ZigBee Router or a ZigBee Coordinator. Moreover, ZigBee Routers can connect to nearby devices to provide multi-path message exchange for other nodes.

Apart from the mesh network, ZigBee can also create two other network topologies:

- **Star**, where nodes are ZigBee End-Devices only attached to a ZigBee Coordinator.
- **Cluster-Tree**, which is an extended Star network where ZigBee Routers are used as clusters to provide ZigBee End-Device connectivity.

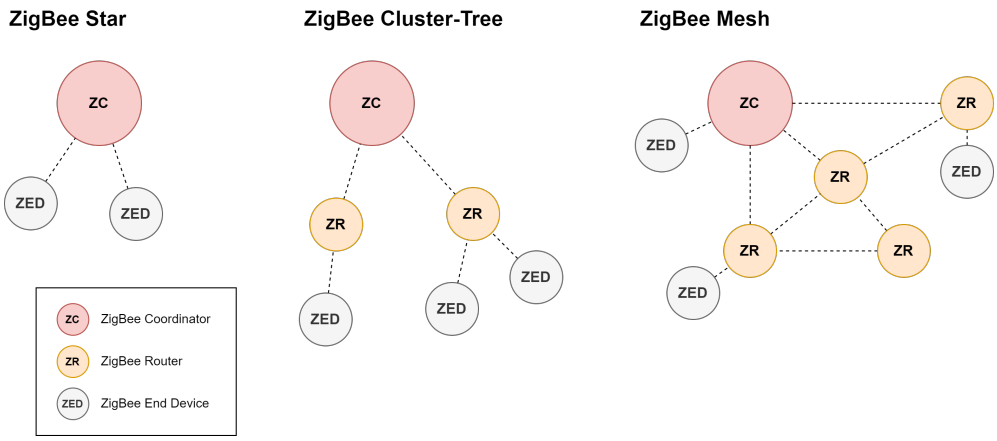


Figure 2.15: Topology views of ZigBee Star, ZigBee Cluster-Tree and ZigBee Mesh.

Architecture overview. As shown in report [4] and Figure 2.16, ZigBee is divided in a four layer-architecture.

- The **Physical layer**, which is defined by the 802.15.4 standard, is the lowest layer of architecture. This layer is the closest to hardware and it is responsible to control message communication with the ZigBee hardware. For example, it handles hardware initialization, channel selection and link quality analysis with other nodes.
- The **MAC layer** is placed above the physical layer and it is defined by the 802.15.4 standard. This layer is responsible to receive data from the physical layer, perform association and dissociation functions (e.g. adding the device inside the network) and also synchronize devices through beacon signals (e.g. using a beacon it is possible determine when the device is able to send or receive messages).
- The **Network layer** is defined by the ZigBee Alliance[6] and works as an interface for the MAC layer. This layer creates the network and handles routing and address allocation through routing tables. Routing tables are used to recreate paths of the connected nodes in the network.

[6]From 2021, the ZigBee Alliance has been re-branded in Connectivity Standards Alliance (CSA). See [83] for more information.

- The **Application layer** is placed above the Network layer and it is used as a host for application objects. These application objects are mapped into three sub-layers.
  - The *Application Framework* (AF) contains manufacturer-defined application objects that are served as capabilities of the ZigBee device. These capabilities are accessible through a total of 240 distinctive service end-point. A service endpoint implements a single function mapped to an application profile. An *application profile* is defined by the ZigBee Alliance and defines message formats and a list of supported clusters. A cluster is a set of capabilities that the device supports (e.g. the *On/Off cluster* provides On/Off switch attributes and commands to the device). These clusters are also defined by the ZigBee Alliance in the ZigBee Cluster Library (ZCL).
  - The *ZigBee Device Object* (ZDO) addresses device discovery (i.e. a ZigBee device detects and identifies another ZigBee device inside the network), service discovery (i.e. a foreign ZigBee device retrieves the clusters and capabilities of another ZigBee device) and binding (i.e. a hook mechanism to provide a client-server interaction for two nodes, where one device acts as a client and sends a command to another device acting as a server).
  - The *Application Support Sub Layer* (APS) provides an interface between the Application layer and the Network layer. Moreover, it is also responsible to interact with the *Security Service Provider*. This provider employs a security mechanism such as key establishment and transport during network operations to secure message exchange across the Network layer and the APS. For example, these keys are exchanged by ZigBee devices to verify their permission to connect to the network.

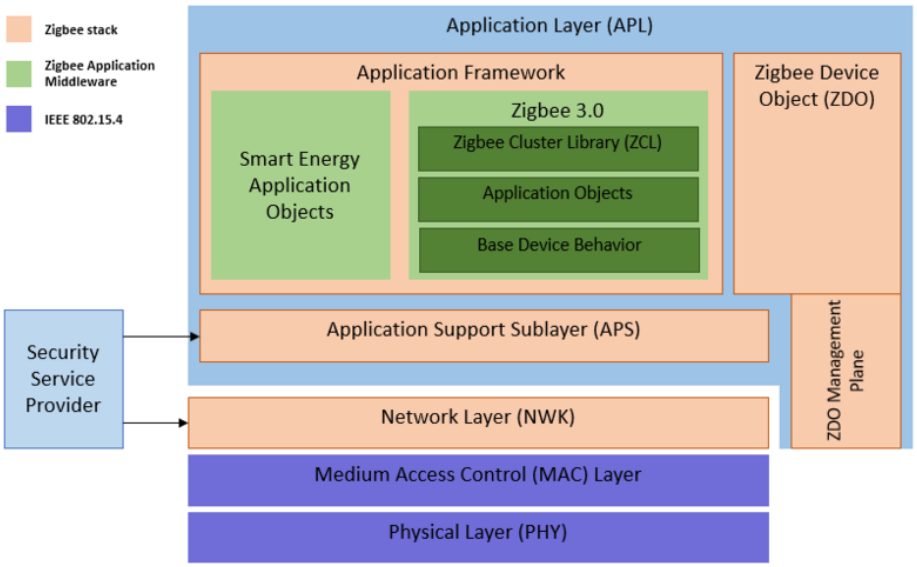


Figure 2.16: ZigBee stack architecture view.
Source: [96]

Discussion. ZigBee is used inside the IoT context to provide low-power device connectivity suitable for smart-home appliances. As a result, both Amazon and Samsung adopt this protocol due to its features of scalability, energy consumption and capabilities mapping. As a matter of fact, ZigBee also provides an application profile that is used for energy optimization called *Smart Energy*. This profile is employed on residential and commercial environments to provide metering (i.e. electricity measurement), pricing (i.e. payment of energy used), scheduling (i.e. when the device executes a pre-defined task), demand response and load control. As a result, this protocol addresses the energy-aware concern from §1.3.3, so that IoT devices are able to report energy usage to end-users.

A 3rd-party manufacturer that decides to adopt this standard can also certify its product through a ZigBee certification. IoT platform providers are entitled to ask 3rd-party vendors to certify ZigBee products before applying to their platform. Consequently, this certification guarantees device compliance with the standard, but causes an increased development cost (i.e. test + certification cost) and production time. In addition, when a manufacturer updates the device firmware, a re-certification can be triggered if necessary.

In conclusion, ZigBee is a powerful industry-standard, adopted for multiple purposes by IoT platform providers. It gives numerous network functions, which perfectly fit the smart-home application domain. However, IoT platform providers adopting ZigBee may require the device to be compliant to the standard so as to be interoperable with other devices, causing an increased production cost.

2.4.3 KNX IoT, a standard for 3rd-party APIs

Overview. KNX is an association that promotes a standard in smart-homes and building automation. As reported in [65], this association is oriented to realize a standard that is accepted by market players and adopted by manufacturers, so as to create a single and affordable technology. This standard aims at providing interoperability between devices and applications.

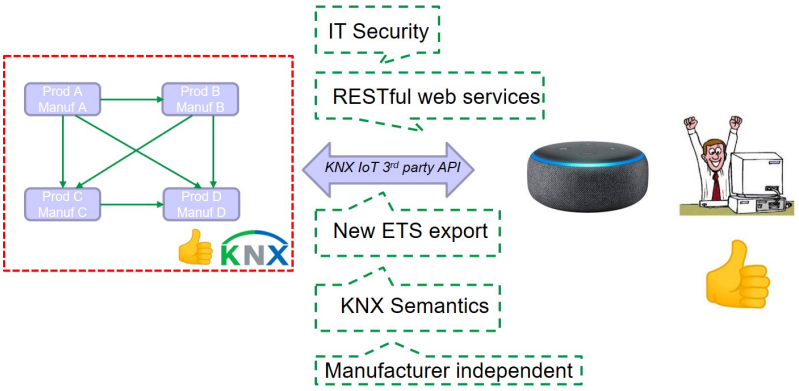


Figure 2.17: KNX IoT 3rd-party API characteristics overview to interact with foreign IoT ecosystems (i.e. Amazon Alexa)
Source: [55]

As motivated in [55], KNX IoT 3rd-party API is a component of this standard, whose purpose is to add interoperability to KNX ecosystems with external 3rd-party vendors

and clients. An external vendor can interact with standard 3rd-party APIs to exchange messages with a KNX platform. In other words, from the definitions in §1.3.1, KNX IoT works on top of the application layer of an IoT architecture. This part takes advantage of a standard protocol to a serve an interface for 3rd-party clients and a Cloud-to-Cloud solution for 3rd-party integrators. Consequently, this standardization leads to interoperability because from the Cloud perspective the same standard can be re-used across multiple IoT ecosystems.

KNX IoT 3rd-party API entities. In this standard, KNX provides freely available APIs specification from [53], that are linked to entities belonging to KNX semantic. Each entity represents an abstraction to interpret different attributes and characteristics of an IoT device. There are several core entities that can be accessed through APIs by 3rd-party manufacturers:

- A **datapoint** is a simple object that represents a readable and writable attribute linked to a device capability. For example, a device that has an On/Off capability can have a datapoint exposing the status of the device, that can be altered with another value.
- A **function** is a collection of datapoints that defines a device capability. For example, a roller shutter switch can have an On/Off capability, but also shades lift and tilt capabilities.
- A **location** identifies a physical zone in a building where functions, datapoints or other entities belong to (e.g. a living room, the kitchen).
- A set of **information** are hosted for 3rd-party manufacturers to discover available services in the API. For example, this set of information contains the server name (i.e. the name of the plant), version of the KNX IoT specification and available API end-points (i.e. where the resources are hosted and available for clients through the APIs).
- A **subscription** is a powerful entity that allows a 3rd-party manufacturer to be notified when an entity is altered. For example, when a light switch is manually turned on, a 3rd-party vendor receives a notification.

Besides this list, other entities that are used in the standard are reported in [53]. In this analysis, the above list is enough to give an overview of the KNX IoT protocol.

API characteristics. KNX IoT covers 3rd-party manufacturers access to a KNX ecosystem through resource-centric REST APIs. The term REST stands for *Representational State Transfer*, meaning that APIs use a REST architectural style for processing data transfer. Therefore, as discussed in [89] and [88], REST APIs follow precise criteria to be considered so. Firstly, REST APIs use the *client-server architecture model*, where a client asks for a resource and a server handles the requests through HTTP. Moreover, client-server communication must be *stateless*, which means that the server completes every client request independently between all previous requests. Hence, each client request is separated and isolated. Secondly, a *uniform interface* must be provided, thus data is transferred in a standard communication form. In other words, this means that the data format from one ecosystem must be adapted to the KNX IoT data format before being served. This interface has four requirements:

- a request shall identify a resource through a URI (Uniform Resource Identifier, e.g. `GET /document/42`);
- clients must have all the required information provided by the server to manipulate resources (i.e. modify or delete a resource);
- a client shall receive a self-descriptive messages from the server to process the representation of received data (i.e. the data format must be self-expressive);
- clients must receive hyperlinks provided from the server to discover related resources.

In addition, REST APIs must be organized in a layered system involved in requests processing operations invisible to the client. Therefore, several components responsible for security, load-balancing and business-logic can be used together to satisfy the client request, without showing the complexity underneath.

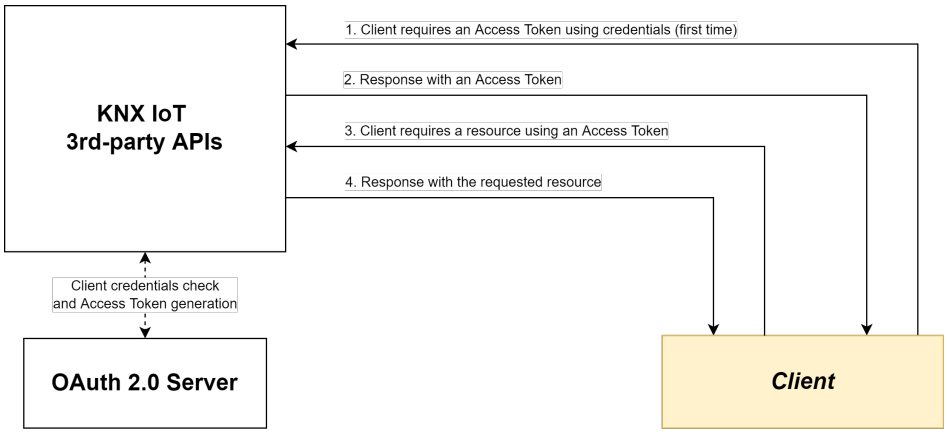


Figure 2.18: KNX IoT 3rd-party API workflow example with a client requesting a protected resource.

API security. Regarding security, KNX IoT 3rd-party API leverages OAuth 2.0 to provide authorization and permissions for clients. This protocol has been partially discussed in §2.3.1. From an architectural perspective, OAuth is a security component that is added in the layered system of the APIs to process authorization requests. Therefore, when a client interacts with APIs, OAuth is invoked to provide security operations described in the RFC 6749 (see [82]). For example, a security operation provides an authentication flow for the client to interact with protected resources. An interaction in a simplified form is the following:

1. The client sends credentials to the KNX IoT 3rd-party APIs using the OAuth data format.
2. These credentials are checked by the KNX IoT 3rd-party APIs through the OAuth server.
3. Then, if credentials are valid, the OAuth server generates an *Access Token*. This token can have a lifetime limit and can be used as a credential to access protected API resources with certain permissions linked to the client account.

4. Afterwards, the token is sent by the KNX IoT 3rd-party APIs to the client in response.

Once the client receives the Access Token, the token must be sent in addition to a resource request. When this token expires, it can be recreated according to the other OAuth operations. To give an idea of this example, Figure 2.18 shows an interaction with a client and a KNX IoT 3rd-party APIs. It is important to note that normally an OAuth server would have its set of APIs to provide security operations. However, as shown in [53] (i.e. KNX IoT 2.0.0 API specification), an authorization end-point is provided directly (i.e. `/oauth/access`), which uses the OAuth server concealed to the client.

2.4.4 On the quest for interoperability

From this overview, both standard protocols aim at interoperability through different approaches. In details, ZigBee is employed in two layers of the five-layer IoT architecture proposed in §1.3.1:

- In the *network layer*, the ZigBee Coordinator is a node that rules network traffic and message exchange of other nodes attached. In fact, the ZigBee Coordinator is a powerful gateway from a proprietary IoT ecosystem that is able to recognize and install new ZigBee-enabled 3rd-party devices.
- Conversely, ZigBee End-devices and ZigBee Routers belong to the **perception layer**. These devices are able to exchange messages with a ZigBee Coordinator based on the network topology (e.g. mesh network) and device capabilities (e.g. a ZigBee End-Device can only send or receive messages with a ZigBee Router).

Regarding KNX IoT, the standard is employed in the *application layer* to expose an API service that is used to exchange data in a conventional data format. This format must be recognized by a client, acting as a middleware de facto. However, 3rd-party Cloud-to-Cloud integration is not free from concerns regarding interface compatibility, API usage and security. Figure 1.12 from §1.4 shows that the application layer from one ecosystem requires the middleware to interact with the application layer of a foreign ecosystem. In this interaction, KNX IoT exploits a client-server communication model that works on top of REST APIs to interpret and transport data in both layers. The API resources and the security protocol to use must be defined by both APIs in order to provide interoperability between services.

In conclusion, in this writing we investigate deeper in the development process of a real implementation. This is important to capture unforeseen constraints, troubles and intricacies for 3rd-party integrators on a practical level. Therefore, the next chapter shows two real implementations of a Gateway-connected integration with ZigBee and a Cloud-to-Cloud interaction using KNX IoT APIs. These integrations are analyzed and compared to understand emerging issues from a 3rd-party integrator perspective in a real use case context.

Chapter 3

Thesis contribution

3.1 Case study definition

In this chapter, we investigate integration solutions of a real case study with a company called Vimar. As the company website reports [85], Vimar covers several commercial sectors belonging to electrotechnical material and electronics (e.g. electrical outlets, intercoms). Vimar is particularly active in home automation as a complementary part of its commercial solutions. Furthermore, Vimar stands out as a device manufacturer since all solutions are completely made by the company itself, except computing units (e.g. microprocessors). The role of this company is especially important for the case study, because Vimar is a device vendor that acts as a 3rd-party integrator for major IoT ecosystems. Consequently, in this case study we had to analyze, design and implement company solutions by impersonating an integrator working for Vimar. The integrator's objective aimed at creating middleware solutions for Vimar IoT devices and foreign ecosystems. These solutions have been realized according to what have been discussed in §2, leveraging market and user needs for the company.

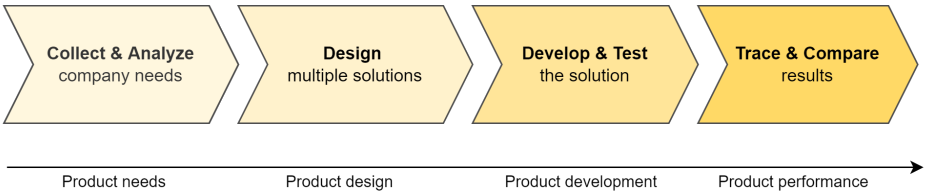


Figure 3.1: Case study phases and objectives in parallel with product realization.

The phases and the objectives of this case study were four:

- We investigated and *collected major company needs* to trace the direction of the case study in sub-parts. Then, we provided an analysis of the best-fitting solutions for foreign ecosystem integrations. To achieve this, we have also analyzed architectural components, features and requirements needed from an integrator perspective.
- Then, we *designed multiple solutions* to build a state-of-the-art device integration. Furthermore, we identified critical parts of each solution, thus simplifying future maintenance and improvements of the solution. This was also essential for the

company to avoid breaking the compatibility of IoT devices with other IoT ecosystems.

- The integration solutions have been developed and tested. Then, we *traced the results* along with emerging problems resulting from the integration.
- Lastly, we analyzed the final product to highlight *pros* and *cons* of each integration. Then, we concluded by showing satisfied company needs compared to the premises.

3.2 Industrial needs

In this section, we identify the company's business needs, which reflect the enterprise and commercial context of Vimar. We thoroughly analyze those needs and broke them down in groups which we addressed in distinct, subsequent phases of this case study. We start with the identification of the IoT devices involved in the case study, so as to understand the context and the product objectives. Then, the case study analyzes company needs and the corresponding solutions. Lastly, we summarize the sub-parts of the case study.

3.2.1 IoT devices for the case study

First of all, we had to assess with the company which category of products required to be studied. In this case, *Smart Home View Wireless* products – from now on called *View Wireless* products – were taken under analysis. These products are a family of IoT devices that covers light switches, actuators with power measurements, roller shutter switches and thermostats. These devices are employed in the Smart-Home context for home appliances control, power consumption and automatic temperature regulation.

From the the technical perspective, this family of products leverages an hybrid nature with two industry-standard protocols implemented in two different firmwares, respectively.

- The former firmware works with **Bluetooth**. In this case, a Vimar IoT device must be connected to a Vimar Bluetooth gateway via mesh network. Moreover, the gateway must be connected via Wi-Fi, thus requiring an Internet modem to provide local and remote connection. Figure 3.2 shows the devices connected using Bluetooth.

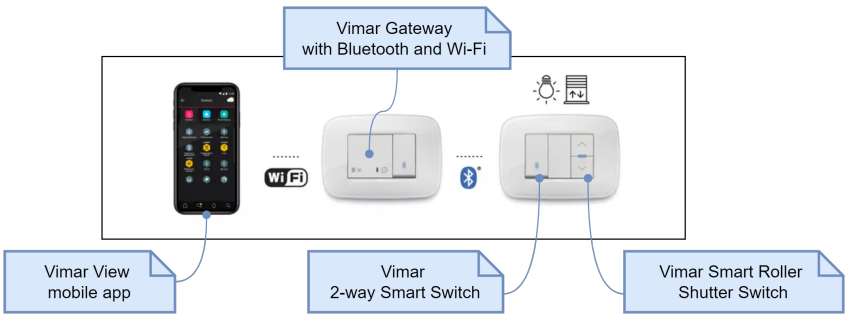


Figure 3.2: Vimar View Wireless products with a Bluetooth-connected gateway integration.

- The latter firmware works with **ZigBee**. In this case, a Vimar IoT device requires a foreign compatible ZigBee gateway to work. Figure 3.3 shows the Vimar IoT devices connected to an Amazon Alexa Hub using ZigBee.

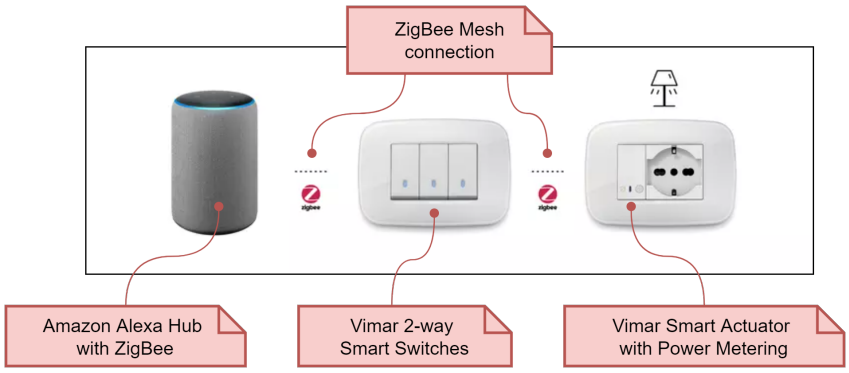


Figure 3.3: Vimar View Wireless products with a ZigBee-connected gateway from Amazon Alexa.

As a result, these solutions aim at satisfying different market needs with respect to a proprietary ecosystem solution (i.e. Bluetooth) and a 3rd-party ecosystem solution (i.e. ZigBee). The end user choosing a Vimar ecosystem can interact with Vimar IoT devices using either a proprietary mobile app (called *View Wireless*) or a virtual assistant from the supported 3rd-party integrations (e.g. Google Home, Amazon Alexa). Table 3.1 shows the supported 3rd-party integrations for the Vimar ecosystem. Furthermore, these information are also reported inside the website [70].

The *Bluetooth* firmware supports 3rd-party integrations in different commercial IoT ecosystems, such as Google Home, Amazon Alexa, IFTTT and Apple Home. The customer must own all Vimar products (i.e. IoT devices and gateway) to use the proprietary app. On the other hand, the *ZigBee* firmware supports Amazon Alexa 3rd-party integration. However, the customer must own an Alexa-enabled Hub to use the device. Other 3rd-party integrations are not officially supported by Vimar.

From Table 3.1, IFTTT – i.e. If-This-Then-That – is the only integration working with the Bluetooth firmware of Vimar View Wireless devices that is not covered in §2. This solution provides user-defined services to run automations with 3rd-party services. For example, it is possible to turn on a light switch whenever we receive an email or a specific notification. Being a service, this solution requires a Cloud-to-Cloud integration to establish the communication.

	Google	Amazon	Apple	IFTTT
BLE firmware	✓	✓	✓	✓
ZigBee firmware	-	✓	-	-

Table 3.1: Supported 3rd-party integrations of Vimar View Wireless device firmwares.

For the case study, we used five models of Vimar devices. We want to clarify that, at the time of this writing, more than five models have been analyzed and integrated. However, part of these integrations cannot be publicly reported due to internal company policies.

- **2-way Smart Switch**. This device is a physical switch that is used to control lightning in the house.
- **Smart Actuator with Power Metering**. This device is used to control a physical plug for large home appliances (e.g. dishwasher, fridge).
- **Smart Actuator Module**. This device is able to control lightning or physical plugs through an internal module that can be controlled through an external switch. The behavior of the firmware is the same of the 2-way Smart Switch.
- **Smart Roller Shutter Switch**. This device controls home roller shutters through a physical device.
- **Smart Roller Shutter Switch Module**. This device is used as a controller of roller shutters through an internal module. This module can be controlled by an end user through an external switch (e.g. up/down). The behavior of the firmware is the same of the Smart Roller Shutter Switch.

After introducing an overview of Vimar products, we analyzed the company needs according to the five products reported above.

3.2.2 SmartThings integration

In the first part of the case study, we investigated with the company a new 3rd-party IoT ecosystem to add in support of View Wireless products. Table 3.1 shows that View Wireless devices leverage on two firmware solutions. The Bluetooth solution covers four 3rd-party integrations, while the ZigBee solution only covers one integration with a foreign ecosystem. Consequently, with respect to §2.1, SmartThings (i.e. Samsung ecosystem) was the only major integration not covered by Vimar devices in either BLE or ZigBee. As a matter of fact, considering SmartThings growth discussed in [94] and §2.1.4, the Samsung ecosystem compatibility became a market need, as well as a company need. Therefore, the main objective for Vimar was to integrate the Samsung solution in View Wireless products, thus satisfying commercial needs for customers.

In order to provide an integration with SmartThings, we discussed three integration solutions reflecting the ones reported in §2.2.

- The first approach was the **Cloud-to-Cloud** integration solution. As shown in §2.3.1, SmartThings supports this type of integration, which requires the design and development of a remote middleware. However, from the application layer perspective, Vimar Cloud would had to supply an external API interface thanks to which the middleware could communicate. Furthermore, this solution would have required the *Bluetooth* firmware on Vimar devices as the only viable solution. In fact, the Vimar gateway handles the Bluetooth network with Vimar devices and so it must be connected to the Vimar Cloud to communicate. Consequently, the entire integration solution aimed at those customers who already have the complete Vimar solution with Vimar gateway and relative devices in their house.
- The second approach was the **Gateway-connected** integration solution. This approach is also supported by SmartThings, as discussed in §2.3.2. In order to accomplish this integration, the ZigBee firmware would have been employed in the development since the ZigBee protocol is compatible with a SmartThings-enabled Hub. As a matter of fact, SmartThings requires a compatible device which implements the ZigBee industry-standard protocol. The device behaviors are

controlled by the SmartThings-enabled Hub with the help of a driver. Consequently, the entire integration solution aimed at SmartThings customers who already own part of the SmartThings solution in their house.

- The third approach was the **Direct device** integration. As shown in §2.3.3, SmartThings also supports this type of integration, requiring a deep modification in terms of firmware. This integration would have required a new firmware adaptation that works either on top of Bluetooth (i.e. mobile-connected integration) or on top of a direct Wi-Fi connection (i.e. direct-connected integration). However, the Wi-Fi integration could not be realized with existing IoT devices, because they are only enabled for Bluetooth and ZigBee from the hardware perspective (see the public device specification sheets in the website [70]). Therefore, a different Bluetooth firmware could have been realized to satisfy a direct connection with a mobile device using the SmartThings app.

With these considerations in mind, SmartThings integration was possible in all three ways. From a company perspective, there were two kind of customers to satisfy. *Vimar customers* – i.e. vertical customers –, owning the complete Vimar solutions, would have found an extra integration available. Conversely, *SmartThings customers* – i.e. horizontal customers –, owning the SmartThings solution, would have been able to use a Vimar device without a Vimar gateway. Since both approaches were capable to satisfy valuable market segments, we identified with the company other decision-making factors to take in consideration.

Time and costs. In order to realize the integration, an estimated time and costs for the development was required. This involved different actors depending on the solutions. In the Cloud-to-Cloud solution, application and Cloud developers were consulted. Conversely, the Gateway-connected and Direct device solutions would have required platform and firmware developers to handle the development – or possible readjustment – of the network protocol (i.e. ZigBee, BLE) and extra components (i.e. device app, drivers).

Capabilities coverage. All solutions had to cover multiple device capabilities depending on the IoT devices to integrate. This was crucial to provide equal device behaviors, as in the complete Vimar solution. For example, a Vimar Actuator with power metering capability had to report the same Watt measurement to the end user both in Vimar ecosystem and in a foreign IoT ecosystem.

Semantics and Data formats. Each ecosystem used its proprietary semantics and data formats in order to abstract devices in classes, attributes and relationships. Therefore, the integration would had to be addressed either at the application layer or at the network and perception layers. In the Cloud-to-Cloud solution, Vimar and SmartThings semantics would had to be acknowledged by developers to build the remote middleware. In the Gateway-connected integration, ZigBee and SmartThings semantics would had to be addressed either for the driver realization. Lastly, in the Direct device integration the firmware internal semantics and the SmartThings semantics would had to be employed in the development of the firmware.

According to time and costs, the company could have decided to lower as much as possible the Time-To-Market (TTM) of SmartThings integration. Therefore, the solution with the lower estimated time to be developed could have taken place. Then,

we analyzed with the company the state-of-the-art capabilities available. We noted that the Cloud-to-Cloud solution covers the same device features of 3rd-party Cloud-to-Cloud integrations with the Bluetooth firmware. Moreover, the Gateway-connected solution covered the same feature set available in the Amazon integration with the ZigBee firmware. Lastly, the Direct device solution required an estimation for each IoT device, since a new firmware was supposed to be created. We also noted that semantics and data formats complexity depended on each integration solution. This means that the integrations depended on the complexity of the abstractions used to provide functionalities in IoT devices.

Furthermore, we highlight that from a state-of-the-art analysis the Gateway-connected solution was already partially implemented because the ZigBee firmware was already available. The local middleware employed in the Gateway-connected solution worked as an independent component that did not required re-adjustments from the device firmware side, at least in the premises.

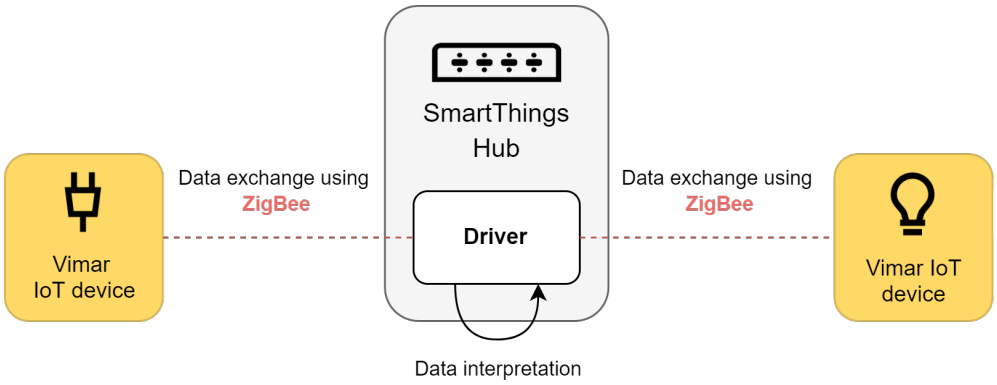


Figure 3.4: SmartThings Hub-connected integration overview with Vimar IoT devices.

Company's first choice was to create a **Gateway-connected solution** by using the already existing ZigBee firmware. This choice would have required lower development time and costs since a SmartThings-compatible protocol (i.e. ZigBee) was already available for Vimar devices. Moreover, the set of features in the ZigBee firmware was the same of the Amazon integration. Therefore, these features had to be linked with the corresponding features in the SmartThings ecosystem, thus requiring a deeper analysis for each Vimar device. We decided to integrate in the SmartThings ecosystem all five devices discussed in 3.3 using the ZigBee firmware.

The Gateway-connected solution appeared to be the best choice at the time of the analysis, because the Cloud-to-Cloud integration and Direct device integration carried negative implications:

- The Cloud-to-Cloud solution would have required a higher development time because both company data-models were supposed to be acknowledged and each device would have required its own realization to properly map features and behaviors. Consequently, this resulted in higher costs in terms of resources (i.e. developers, cloud resources), thus the solution was not taken under consideration;
- The Direct device solution would have employed a new firmware development which was not re-usable in different 3rd-party integrations. Moreover, this solution would have required the constant use of a mobile device connected to the IoT device, lacking remote controls of the latter.

After analyzing the first need, we organized the sub-part of the case study – from now on called Case Study A. Before proceeding, we also investigated the second company need.

3.2.3 KNX IoT 3rd-party API integration

In the second part of the case study, we take a look with the company at 3rd-party APIs. Vimar is a company that takes advantage of standards to re-use architectural components. In this case, Vimar implemented a standard 3rd-party API solution to provide device control for external customers. This standard is called KNX IoT and provides a standard HTTP interaction through REST APIs. From the security perspective, OAuth 2.0 is already implemented as well. We have discussed both standards (i.e. KNX IoT 3rd-party API and OAuth 2.0) in an overview presented in §2.4.3. Vimar is an active partner and promoter of the KNX standard in multiple home automation solutions. Recently, with the introduction of KNX IoT, Vimar took part in the development of the standard to realize 3rd-party APIs specification.

From a company perspective, the KNX IoT 3rd-party API is a cloud component that could have been used for two different use cases.

Access to the IoT ecosystem. The first use case concerned a *standard 3rd-party API access for external customers*. This became important to grant access to the company ecosystem for 3rd-party integrators. For example, an external customer that chooses Vimar devices for an office can create its own application to provide lightning control, power metering and thermostat control. These features can be implemented by an external integrator to handle power consumption and remote control of the office equipment with custom functionalities.

Cloud-to-Cloud integrations. The second use case concerned the *Cloud-to-Cloud integrations* with foreign IoT ecosystems. In this case, a company owning KNX IoT APIs would have been able to employ an external Cloud component to handle Cloud-to-Cloud integrations. For example, this use case enables a company to provide a Cloud-to-Cloud integration with SmartThings, which requires an AWS Lambda Function that interacts with the KNX IoT APIs.

The use cases described above show how KNX IoT APIs could have been used for different purposes. As a matter of fact, Vimar already implemented KNX IoT APIs server-side. However, in order to provide Cloud-to-Cloud integration, we analyzed these use cases to understand middleware's nature. In both cases, the remote middleware acts as an intermediary between two interfaces, as discussed in §1.4. Therefore, being KNX IoT based on REST APIs, we expected a client-server communication model.

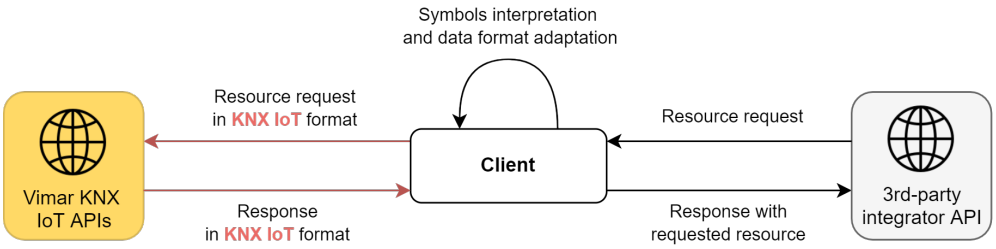


Figure 3.5: KNX IoT client solution overview for Cloud-to-Cloud integrations.

Therefore, our analysis showed that a middleware solution can be regarded as a **client-side entity**, which handles the Cloud-to-Cloud integration. As a result, a client component provided by the company could have helped 3rd-party integrators to understand entities and relationships of the KNX IoT standard. Moreover, a client could have been reused in 3rd-party integrations of external customers as a modular component. For example, an external customer that wanted to create its integration could have been able to employ the Vimar supplied client in its code-base to communicate with the KNX IoT APIs.

These premises suggested that the client-side middleware discussed above was supposed to feature the following properties:

- The client had to be **compliant** with *KNX IoT standard*. This was mandatory in order to provide KNX IoT compatibility with the corresponding semantics and data formats. Moreover, it had to support all the available API end-points, which were supported also server-side. Additionally, the client had to include entities, attributes and relationships of the KNX IoT data-model.

- The client had to be **re-usable**. Therefore, it had to be realized in a form of an *SDK* or a *library*, so as to be implemented in multiple 3rd-party integrations.

- The client had to support **vertical extensions**. This means that 3rd-party integrators were supposed to use different abstraction layers of the client in their integration. These layers were also incrementable, meaning that a new layer can be added to the architecture to provide a new level of abstraction. For example, an integrator can use the client to retrieve raw HTTP response data with the lowest level of abstraction. However, by adding a new vertical component it is possible to provide objects, which are capable of handling the complexity underneath.

- The client had to support **horizontally extensions**. This means that each horizontal component of the client can be extended by a 3rd-party integrator to include new functionalities covering additional needs. For example, an object representing a light switch can be wrapped with extra features (e.g. decorator pattern) to handle additional states and behaviors client-side.

With these features, we designed a client for the second sub-part of the case study – from now on called Case Study B –, which satisfied Cloud-to-Cloud 3rd-party integrations. The Vimar devices used for this integration were the same reported in §3.2.1, with the exception of the Smart Roller Shutter Switch and the Smart Roller Shutter Switch Module. We provided an integration of only a part of devices to demonstrate the capabilities of the client. Future development will be dedicated to the improvement of new devices.

3.2.4 Summarizing company needs

In this section, we summarize major company needs for the case study. We provided with the company a classification of marketing needs and technological requirements of each family of products. Marketing needs describe the expected behaviors and quality for end users. Conversely, technological requirements describe expected device features and specifications. However, due to company policies, marketing needs and technological requirements are not reported in this thesis. We consider that the absence of these needs does not impact the objectives of this case study, described in 3.1.

Table 3.2 shows the two major company needs addressed in §3.2.1 and §3.2.3. We refer to these needs with CS-A (i.e. Case study A, Gateway-connected integration) and CS-B (i.e. Case study B, KNX IoT API client).

#	Description
CS-A	Gateway-connected integration for the SmartThings ecosystem
CS-B	KNX IoT standard API client for Cloud-to-Cloud integration

Table 3.2: Major company needs for this case study divided in Case Study A and B.

We addressed the Gateway-connected integration for each Vimar device under study. The KNX IoT 3rd-party API client, instead, was employed for only three Vimar devices. Table 3.3 shows Vimar devices involved in case study A and B. We must note that each Vimar device had a different number of features to satisfy, according to the integration to accomplish.

#	Device	CS-A	CS-B
1	2-way Smart Switch	✓	✓
2	Smart Actuator Module	✓	✓
3	Smart Actuator with Power Metering	✓	✓
4	Smart Roller Shutter Switch	✓	-
5	Smart Roller Shutter Switch Module	✓	-

Table 3.3: Vimar devices involved in CS-A (Gateway-connected integration) and CS-B (KNX IoT API client).

In the following sections, we present a deep analysis for CS-A. We take a look at SmartThings ecosystem from an integrator perspective. The objective was to realize a Gateway-connected integration for the devices in Table 3.3. Then, the second part of the case study (i.e. CS-B) is analyzed to provide a KNX IoT API client for Cloud-to-Cloud integrations.

3.3 Integrating Gateway-connected IoT devices

In this part we investigate CS-A for Gateway-connected integration of Vimar IoT devices. The analysis starts from an overview of SmartThings Hub-connected solution, in which we explain the main workflow of SmartThings ecosystem. Then, we provide an overview on SmartThings data-model. This overview reflects the 3rd-party integrator needs to create a driver that uses both ZigBee and SmartThings semantics. Furthermore, speaking of drivers, SmartThings provides two different driver solutions. We analyze both solutions to understand pros and cons for an IoT device integration. After designing and developing the driver, we identify emerging problems from the point of view of a 3rd-party integrator.

3.3.1 SmartThings integration workflow

In this section, we analyze the device workflow of a SmartThings Hub-connected integration. In §2.3.2, we discussed the Gateway-connected solution provided by SmartThings.

This solution required a driver component that is installed in a SmartThings-enabled Hub to handle 3rd-party IoT devices. We wanted to understand how the interactions worked and which components were involved in the workflow.

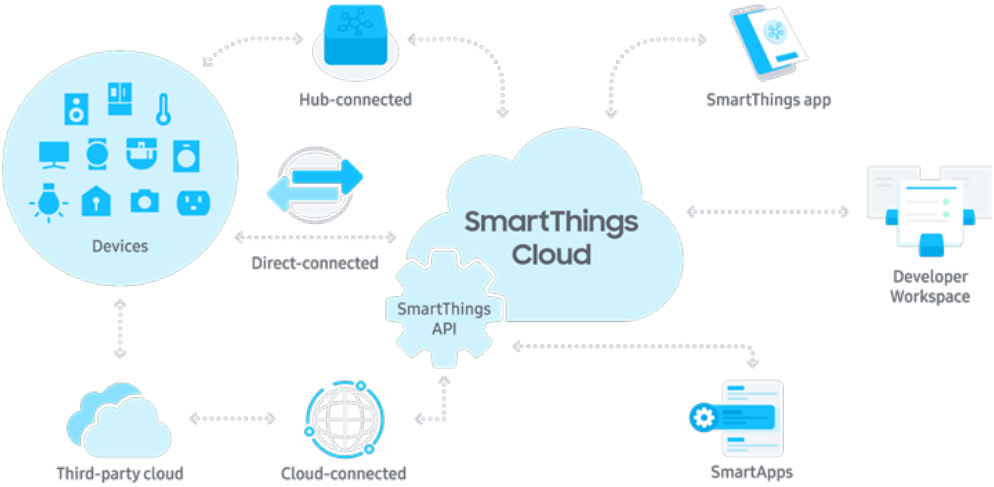


Figure 3.6: SmartThings integration solutions view showing the various high level interactions with the SmartThings Cloud and related SmartThings components (e.g. SmartThings app, developer tools).
Source: [31]

Figure 3.6 depicts the main workflow of different integration solutions of SmartThings. A huge part of SmartThings components such as Hubs, mobile applications and developer tools leverages the SmartThings Cloud. The SmartThings Cloud requires access to the Internet to be establish a connection. However, with the introduction of powerful gateways, part of the computing power is delegated outside the Cloud. Consequently, in this integration solution, we wanted to identify how each component interacts to process data inside the IoT platform.

From the architectural viewpoint, the Hub-connected solution is made of several components belonging to different layers, based on the IoT architecture proposed in §1.3.1.

- **IoT devices** use the ZigBee protocol connected through a mesh network. This network allows nearby devices to connect to exchange messages through paths, as discussed in §2.4.2. Moreover, this mesh network is strictly controlled by the Hub, acting as ZigBee Coordinator. Conversely to the ZigBee standard, the SmartThings-enabled Hub is also the only node that can act as ZigBee Coordinator. Other IoT devices cannot impersonate the ZigBee Coordinator function. This design choice made by Samsung supplants the self-healing feature of ZigBee, because when the Hub fails, the entire mesh network cannot be controlled by other ZigBee nodes.

- A **SmartThings-enabled Hub** leverages the role of ZigBee Coordinator. This device handles message exchanges and network traffic between IoT devices. Moreover, it uses *drivers* to register either proprietary or 3rd-party IoT devices. SmartThings products, as well as 3rd-party products, can join the network based on the drivers installed in the Hub. The Hub is also responsible for the

Internet connection that brings data exchange with the SmartThings Cloud. The SmartThings-enabled Hub can also handle pre-defined routines and automations made by end users. These routines are executed inside the Hub. Hence, if the Internet connection is absent, routines will still be available for the users. However, the users are not able to directly control the Hub, unless the Hub itself is connected to the Internet. Therefore, if a user defines a routine that involves two IoT devices in the ZigBee network, the Hub registers and executes the routines even if the Internet connection is absent. Moreover, routines are also available for heterogeneous family of products. For example, when a SmartThings occupancy sensor detects a movement, the Hub is notified and executes the routine by invoking a turn on command for a 3rd-party light.

- The **SmartThings Cloud** is the main processing unit of the SmartThings architecture. It manages the end user Samsung account in the SmartThings platform. The Samsung account is employed across the entire SmartThings architecture to identify user devices (e.g. Hub, ZigBee-connected devices, mobile app) and permissions (e.g. Hub control, account linking for 3rd-party devices). The SmartThings Cloud is also responsible for the development tools and portals, which are used by 3rd-party integrators. Furthermore, the SmartThings Cloud provides also a set of Public APIs for Cloud-to-Cloud integration solutions.

- The **SmartThings App** is a mobile application for end users. With this application, a user can control connected home appliances, routines and automations. Moreover, in this app a user can install new SmartThings-compatible devices, organize locations (e.g. living room, bedrooms) and manage account settings. Furthermore, this mobile app is the main user interface to control the devices in the SmartThings ecosystem.

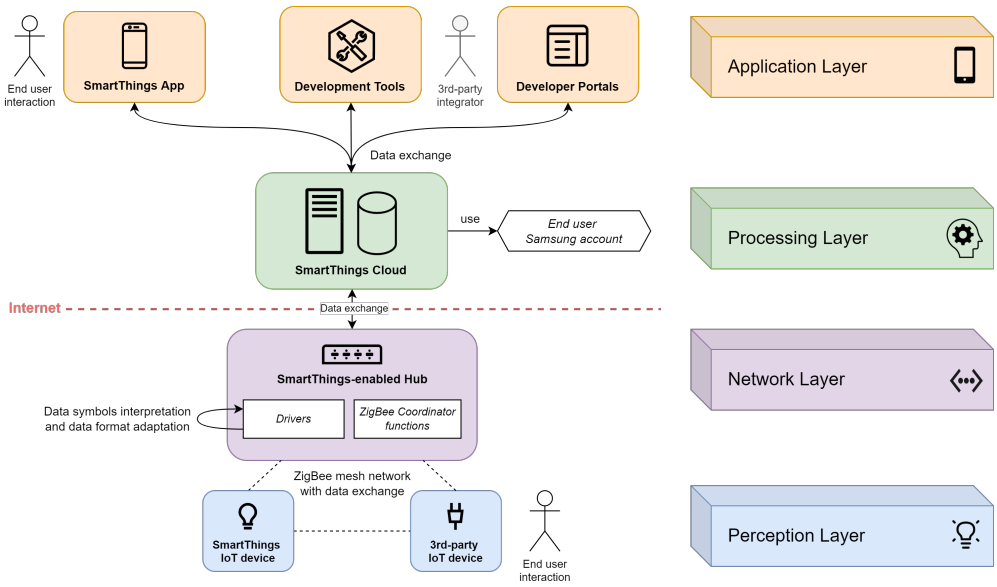


Figure 3.7: SmartThings Hub-connected solution workflow based on the five-layer IoT architecture. The business layer is omitted in this figure to show the main interactions involved in the case study.

Figure 3.7 summarizes the architecture described above. We wanted to highlight a parallelism with the proposed architecture. As shown, IoT devices belong to the perception layer and connect through the SmartThings-enable Hub via ZigBee mesh network. The Hub, which belongs to the network layer, handles the Internet connection to the SmartThings Cloud to exchange data regarding commands or status requests of IoT devices. Then, the SmartThings Cloud belongs to the processing layer and provides data exchange for mobile apps, tools and portals – which belong to the application layer. Since Samsung does not provide an official documentation of the internal SmartThings Cloud architecture, we assumed that a set of internal APIs is supplied for mobile apps and developer tools. These internal APIs should not be confused with SmartThings Public APIs, that are currently in preview at the time of this writing, as shown in the official website [78]. Public APIs are used to interact with the entire SmartThings platform (e.g. automations, locations, devices) for 3rd-party app developers.

In conclusion, the workflow of the SmartThings ecosystem showed at high level the interaction between the main components of the architecture (i.e. IoT devices, SmartThings Hub, SmartThings Cloud and applications). For CS-A, the SmartThings-enabled Hub was the component of interest, because it contained the *driver* that acts as a middleware between the 3rd-party IoT device and the SmartThings ecosystem. Consequently, before dealing with driver development, we have analyzed SmartThings semantics to understand design choices and any possible emerging issue related to the available features for the middleware implementation.

3.3.2 SmartThings semantics: an overview

SmartThings provides different solutions for IoT device integrations, discussed in §2.1.4. Hub-connected solution is one of the integration solutions that works on top of a SmartThings-enabled Hub, which we identify as a powerful gateway in §2.3.2. The Hub is able to provide interoperability between IoT devices using the ZigBee protocol. As a matter of fact, interoperability is achieved with an extra component called driver, that is used to interpret ZigBee commands, convert data formats and send SmartThings-formatted payloads to the SmartThings Cloud. We noted that this workflow can also work backwards: when the SmartThings Cloud transmits a command to the IoT device, the driver handles the communication to the IoT devices.

The SmartThings Cloud identifies an IoT device using a semantics specifically designed for the SmartThings ecosystem. This semantics is strictly related to drivers, as it contains entities and relationship to recognize a foreign IoT device and interpret incoming data. SmartThings semantics is divided in entities that we report in the following list:

- **Device profiles.** A device profile is the definition of properties and functionalities belonging to an IoT device. In the device profile, three different entities are present:
  - **General information.** The first entity is a set of general information specifing the ID of the device profile, the status (e.g. development, production) and the device profile name.
  - **Metadata.** Metadata is a set of additional information, identifying device type, vendor identifier and extra properties.
  - **Components.** A component is an entity used to assemble multiple capabilities. Each device profile must have at least one component and each

component has at least one capability. Optionally, one or more categories can be specified inside a component. We have defined capabilities and categories in a separated list below.

A *component* can contain two types of entity:

- **Capabilities.** A capability is an abstraction of a device function. The SmartThings Cloud is able to report the status of a device component (e.g. relays status). Moreover, this status can vary through commands sent by the IoT device or the SmartThings Cloud (e.g. the relay is physically set to open and the SmartThings Cloud is notified of this status change).
- **Categories.** A category identifies the SmartThings-defined type of device that the end user is interacting with. Furthermore, the category specifies also parts of the UI (e.g. the default icon) used in the SmartThings mobile application.

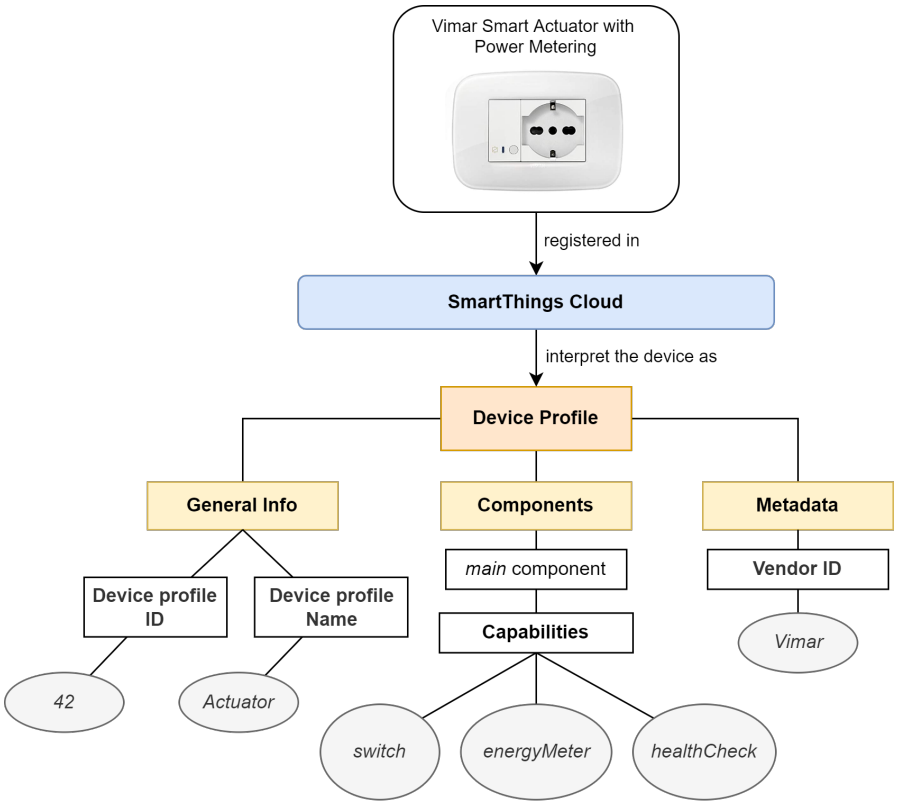


Figure 3.8: Example of a device profile that defines the identities and properties of an IoT device inside the SmartThings ecosystem.

Figure 3.8 shows the structure and the relationship of identities and properties, which are parts of the device profile. Device profiles are employed inside the SmartThings ecosystem to represent the abstraction of an IoT device. In this case, SmartThings expresses the device profile in a JSON-formatted payload, so as to be interpretable for external 3rd-party integrators.

The device profile shows the functionalities of the IoT device expressed in capabilities. These capabilities can have attributes and commands belonging to the SmartThings semantics and data formats. For example, in Figure 3.8 the Vimar Smart Actuator with Power Metering has a capability called *switch*. This capability is documented in SmartThings with corresponding attributes and commands. Figure 3.9 shows how a switch capability is implemented, having an attribute called *switch* and two enumerated commands, which are *on* and *off*.

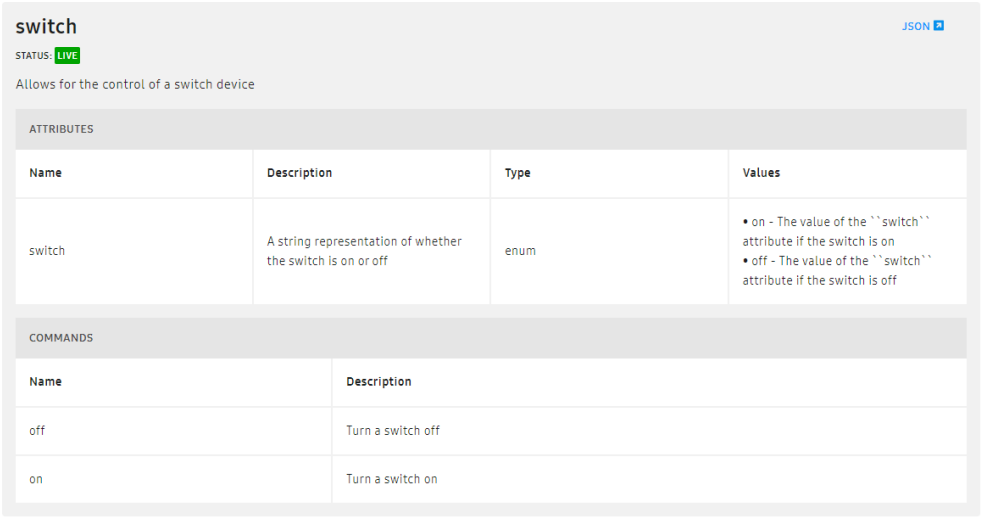


Figure 3.9: Example of a SmartThings switch capability that is used to control a switch device with corresponding attributes (i.e. switch) and commands (i.e. on, off). *Source:* [31]

Device profiles constitute a part of the SmartThings semantics to handle IoT devices in the SmartThings ecosystem. In order to make ZigBee devices communicate with a SmartThings-enabled Hub, a driver must handle the entire device profile, including device ID, ZigBee clusters and additional properties. In the following sections, we introduce a comparison of drivers using the SmartThings semantics and working inside the Hub. These drivers can be adopted for the Gateway-connected integration with corresponding pros and cons. Therefore, we started by analyzing these drivers and then we chose the best-fitting option for our integration.

3.3.3 Legacy vs new drivers

At the time of this case study, we investigated drivers solutions for the SmartThings-enabled Hub. We have worked closely with a SmartThings team to understand the differences between two drivers solutions provided for the integration. At the time of the analysis, SmartThings was introducing a new driver solution in preview that was called *Edge Drivers*. However, before Edge Drivers, another solution called *Device Handlers* was employed for 3rd-party integrations. Since Device Handlers were still available as a *legacy* solution, we wanted to analyze the differences against Edge Drivers to capture the corresponding pros and cons before realizing the integration.

Device Handlers. Device handlers are lightweight drivers, made of a single script

file written in Groovy programming language. A single driver contains a set of functions used in SmartThings to convert ZigBee commands in SmartThings events, and vice-versa. SmartThings events are the abstraction made by SmartThings to interpret requests of any kind coming from different entities. For example, with an event we can notify the device status to SmartThings or handle a specific payload sent by the ZigBee device. The main characteristics of device handlers concern the difference in handling 3rd-party devices. As a matter of fact, a device handler contains all the supported 3rd-party IoT devices belonging to a specific category (i.e. switches, light bulbs) provided by SmartThings. For example, in the official public repository [75], device handlers are organized by manufacturers and by device main characteristics. This categorization generates bewilderment, because part of device handlers are provided and maintained by SmartThings and another part is maintained by 3rd-party manufacturers, with the supervision of SmartThings. Moreover, a 3rd-party manufacturer can decide whether to create a new device handler with its customization or re-use the already provided device handlers with less customization.

The device handler is made of different parts that are required to support a 3rd-party device. These parts are the following:

- **Driver information and capabilities**. A device handler defines its information regarding name, ID and author. Furthermore, SmartThings capabilities are defined to enable device functionalities. These capabilities requires ad-hoc command handling functions to interpret the device behavior.

- **Fingerprints**. A fingerprint is used to identify a ZigBee device through its exposed properties. These properties are contained inside a ZigBee cluster called *Basic*, which holds the main identification attributes of the device. For example, the cluster contains attributes such as model id, vendor information, hardware version, firmware version and related characteristics. The SmartThings fingerprint is able to identify an IoT device by using ZigBee clusters and ZigBee application profiles defined in the device. These information are accessible through the service discovery mechanism that we discussed in §2.4.2.

- **Message parsing function**. When an IoT device sends a command, the device handler employs a message parsing function defined inside the device handler. This function interprets symbols from the ZigBee data format and builds the corresponding SmartThings event to send to the SmartThings Cloud. Consequently, in this part, the semantics of ZigBee and SmartThings are employed to properly translate device behaviors. A 3rd-party integrator can customize this part according to its device behaviors.

- **Command handling functions**. A device handler is equipped with auxiliary functions defined by SmartThings for the ZigBee integration. These functions can be employed for initial device configuration, device status refresh and commands handling (e.g. on/off command invocation, roller shutters direction management). Moreover, extra functions with custom names can also be used to support existing functions.

- **User interface customization**. A device handler can also customize the user interface in the mobile application. However, at the time of this case study, this functionality has been deprecated in favor of a SmartThings-defined UI that is common across the entire mobile application. Before deprecation, it was possible

to add custom icons, colors and widgets to the mobile application, thus allowing 3rd-party developers to make different UI choices.

Edge Drivers. An edge driver is a package of scripts, written in the Lua programming language. This package embeds 3rd-party implementations hierarchically organized. Each driver has a generic implementation that defines the attributes and the capabilities of the driver. Furthermore, each driver can include a sub-driver which is manufacturer-defined. The hierarchy dept has no limit, hence multiple sub-drivers and sub-sub-drivers – and so on – can be included. For example, an Edge Driver that concerns a *Light Switch* can be defined with the corresponding capabilities and functions. Then, a sub-driver can be included to add dimming controls, thus regulating the light intensity through the supported SmartThings capability – which is called *switchLevel* in the SmartThings documentation. Moreover, a manufacturer can even add a deeper level of sub-driver to interpret specific device behaviors from additional ZigBee clusters – such as Light color, from the previous example. Thanks to this approach, SmartThings achieved an ordered way to realize an Edge Driver, according to manufacturer needs. However, when a top level package is modified, the nested sub-drivers inherit those modifications. Therefore, this can be a risk in terms of maintainability, because this can break compatibility with existing integrations. Nevertheless, the SmartThings team develops and maintains top level packages, and also supervise the development of nested sub-drivers realized by 3rd-party manufacturers.

Edge Drivers are composed of similar functionalities that are also present inside device handlers.

- **Driver configuration**. The entire driver configuration is defined inside a YAML file (called *config.yml*). This file defines the name of the drivers, the protocol supported (i.e. ZigBee) and the `packageKey`, which is used to identify the driver inside the SmartThings ecosystem.

- **Fingerprints**. A fingerprint is used to identify an IoT device based on the ZigBee attributes exposed. Similarly to Device Handler, fingerprints define the *model id* and the *manufacturer id* of the supported device. These information can be added inside a common YAML configuration file (called *fingerprints.yml*), which contains 3rd-party device fingerprints defined by 3rd-party manufacturers. Furthermore, a fingerprint binds a device profile to an IoT device, thus defining its capabilities and categories.

- **Device profiles**. A device profile defines the SmartThings capabilities according to the UI widgets. Therefore, a 3rd-party manufacturer can define the items of the UI in the mobile application. However, SmartThings manages these widgets, so UI customization is limited. We noted that the device profiles contains a versioning system for each capability. Therefore, each capability can be updated UI-side according to the version defined in the device profile. This means that a different UI can be used for the same capability. Consequently, a driver behavior can also be affected, according to the attributes and commands of a specific capability version. At the time of this writing, Edge Drivers are still available in beta version, therefore only one version of each capability is present.

- **Capabilities**. Capabilities are defined in the device profile of a 3rd-party device and in the driver business-logic. This means that a sub-driver inherits all the capabilities of a parent driver that declares the capabilities. This approach also ensure re-usability of components (i.e. parent driver) at the cost of less

independence (i.e. if a capability is added to the parent driver, the sub-driver automatically includes that capability).

- **Functions**. The package of an edge driver includes Lua files used to define device behaviors through functions. These files contain all the required functions to interpret payloads from ZigBee devices, understand the message and create events. This process also works towards ZigBee devices and it is similar to the message parsing function defined in the device handlers. However, device handlers use only one function to handle message parsing, while Edge Drivers employ *function handlers*. These handlers can be customized according to the device behaviors supported by the driver. For example, an handling function for *switch* capability manages commands to turn on or off an IoT device. A 3rd-party manufacturer can override the default on/off handler providing a function callback in the driver definition.

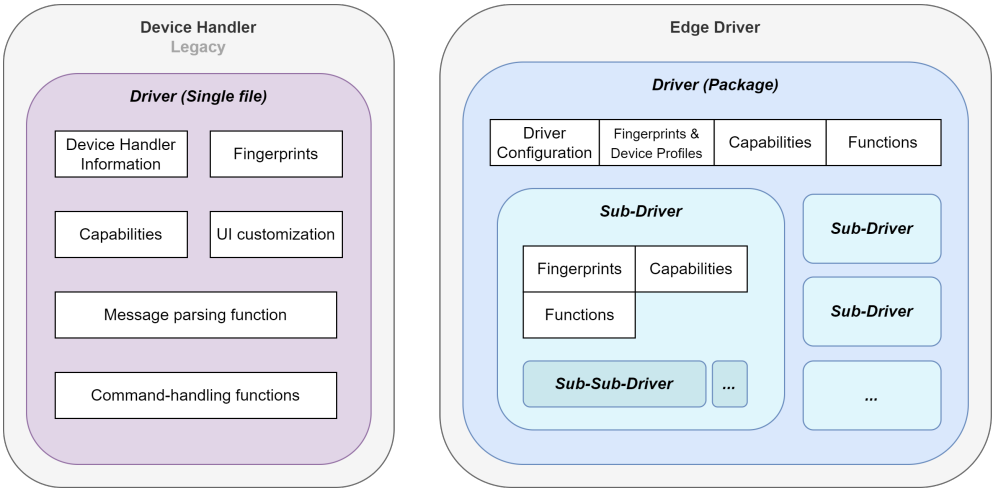


Figure 3.10: Structures of a Device Handler (legacy solution) and an Edge Driver (newer solution). The former solution works within a single file, while latter solution is structured with a hierarchy of packages.

Discussion. At the time of this writing, SmartThings was working on the conversion of Device Handler into Edge Drivers. However, the previous solution shares some differences that are important to highlight to understand pros and cons.

- **Single file vs Package hierarchy**. The single file approach creates more confusion for 3rd-party integrators as opposed to package hierarchy. This is caused by the fact that many device handlers share code forking techniques to add customization inside SmartThings-maintained code. In this case, many manufacturers used *if-statements* to provide portion of code that shall be executed only for their devices. This approach fails in terms of good-practices for development and maintenance. Conversely, a package hierarchy approach addresses this issue by giving 3rd-party vendors their *portion* of driver. However, a hierarchy can cause problems whenever a sub-driver relies on top of a driver definition. For example, if a top driver function is changed, all the sub-drivers are affected, thus causing instabilities.

- **UI Customization**. Device handlers supported UI customization that became deprecated at the time of the case study. As a result, SmartThings approach limited 3rd-party manufacturers in customizing the user experience. Hence, user experience is led by SmartThings itself, according to its design choices. However, Edge Drivers support a bit of customization through capabilities versioning, which is still poorly-defined at the time of this writing.
- **Local Routine Execution**. Device handlers support local routine execution only for those drivers that are developed and maintained by SmartThings. Hence, custom device handlers require an Internet connection to execute routines on nearby devices. Conversely, Edge Drivers are installed inside SmartThings-enabled Hub. Therefore, local execution is widely supported for any kind of driver.
- **Unit Testing**. Edge Drivers support the creation of unit tests. These unit tests are employed to simulate device behaviors before submitting a new driver to the SmartThings platform. Device Handlers do not support this feature, but instead they use simulators that are embedded inside the device handler file script. In this case, it is clear to see that device handler approach fails in separating business-logic with tests for 3rd-party manufacturers.

After working with device handlers, Edge Drivers became the choice for the SmartThings Hub-connected integration. In fact, we discovered that the legacy approach will not be supported in the future by the SmartThings team, even though Edge Drivers were in still beta at the time of the case study. We note that with the company we realized both driver solutions to deeply study both approaches. However, the Edge Driver solution has been chosen at the time of the analysis as the final products, due to the pros in terms of local routine execution and package hierarchy management. In the following section, we go deeper in the design choices to create an Edge Driver for the devices involved in CS-A.

3.3.4 Trial implementation

In this section, we introduce an analysis of Edge Drivers integrations for five Vimar devices. First, we present an overview of the development process, that we discuss following a real integration use case. We start from the 2-way Switch to illustrate the Edge Driver implementation. This implementation was the same across all IoT devices of CS-A. Then, for each device we analyze expected behaviors and functionalities. Lastly, we discuss problems and concerns regarding the integration solution. At the time of this writing, since Edge Drivers were still in beta version, we suggest to follow the up-to-date SmartThings documentation [39] to reproduce the experiments.

```yaml
# Fingerprints (fingerprint.yml)
zigbeeManufacturer:
  - id: "Vimar/xx592-2-way-smart-switch"
    deviceLabel: "Vimar 2-way Smart Switch"
    model: "On_Off_Switch_v1.0"
    manufacturer: "Vimar"
    deviceProfileName: "vimar-on-off-bulb"
```

Snippet 3.1: Example of a fingerprint file for an Edge Driver configuration.

Development process. We present the development process by taking the Vimar 2-way Smart Switch as an example. This smart switch enables the user to control lightnings connected inside the house. Therefore, the device reports to the Hub the status of a switch. In order to create a compatible driver for a 3rd-party IoT device, we discuss five phases of the development process, which we employed for the integration.

1. **ZigBee device analysis**. At first, we inspected the device properties belonging to the ZigBee protocol. For the Vimar 2-way Smart Switch, we identified the ZigBee *On/Off* cluster (i.e. Cluster ID: `0x0006`) as the responsible for switch controls. We checked the availability of this cluster inside Vimar devices using *ZigBee Service Discovery*. In order to provide these information, we used an external tool[1], which acted as a ZigBee Coordinator to retrieve clusters and attributes from nearby ZigBee devices. Another approach would have required an analysis of the internal firmware specification of the device. In addition to the *On/Off* cluster, we gathered all the required device information using the ZigBee *Basic* cluster (i.e. Cluster ID: `0x0000`). These information concerns manufacturer name (i.e. Attribute ID: `0x0004`) and model identifier (i.e. Attribute ID: `0x0005`), that are required by SmartThings for the fingerprint definition.

2. **SmartThings capability analysis**. In SmartThings, the main feature of this device corresponded to the *switch* capability, which was reported in the SmartThings capabilities list [77]. This capability has one attribute (i.e. *switch*), identified by a string value (i.e. either *on* or *off*), and two commands to turn the switch either on or off. We verified that the behavior of the ZigBee Cluster corresponded to the behavior of the SmartThings capability. In this case, we bonded the *switch* capability to the *on/off* cluster, meaning that we took a note of this capability, since it would have been used for the device profile creation.

3. **Edge Driver selection**. We chose an existing Edge Driver following the capabilities needed. In this case, we worked with the public repository [76] for SmartThings-maintained Edge Drivers. Therefore, this repository has been downloaded and the files inside have been edited to realize our implementation. We selected the closest driver that suited our needs, according to the capabilities provided by the existing device profiles. For the 2-way Switch, we used the *zigbee-switch* Edge Driver, which satisfied basic switch operations.

4. **Edge Driver configuration**. We implemented the actual driver starting from the configuration. We created a new fingerprint inside a YAML file (i.e. *fingerprint.yml*). The fingerprint required the model and vendor identifiers from the ZigBee *Basic* cluster. Then, extra entries were required, such as the device identifier (related to the SmartThings platform), the device label name (i.e. the actual device name that appears to the end user) and the device profile (that contains the declaration of the capabilities to use). Snippet 3.1 shows an example of a fingerprint file employed for the Edge Driver configuration. The fingerprint defines a device profile that the device uses in order to provide capabilities. Device profiles are defined in YAML files inside the Edge Driver folder (i.e. *profiles/mydeviceprofile.yml*). These profiles can be re-used according to 3rd-party device needs. In this case, we employed the *on-off-bulb* profile where the *switch* capability is present. Snippet 3.2 shows an example of a device profile definition.

[1] Due to internal company policies, we cannot provide the name of this tool.

5. **Business-logic development**. The business-logic of the Edge Driver has been implemented according to the device features. We noted that the Edge Drivers use a naming convention for the *main* file, which is called *init.lua*. This file can be used to add new sub-drivers, which are organized in folders contained inside the parent driver. When we developed the driver, we could choose either to create a new sub-driver, thus re-implementing inherited functions for a complete customization, or to use an already implemented sub-driver, thus re-using existing components without customizations. It was also possible to reuse the default implementation of the driver. In this case, the Vimar 2-way Smart Switch did not need extra features, besides the existing ones. Therefore, the fingerprint configuration was sufficient to make the device work with the default functionalities.

```yaml
# Device Profile (vimar-on-off-bulb.yml)
name: "vimar-on-off-bulb"
components:
- id: "main"
  capabilities:
  - id: "switch"
    version: 1
  - id: "refresh"
    version: 1
  categories:
  - name: "Light"
```

Snippet 3.2: Example of a device profile for an Edge Driver configuration. Capabilities are listed in this file and versioned, according to the SmartThings documentation. A category can be used to identify the device type and the corresponding icon in the SmartThings mobile application.

2-way Switch and Actuator Module. The 2-way Smart Switch from Vimar was developed according to the ZigBee specification, that used the ZigBee *On/Off* cluster. This cluster was also employed in the Smart Actuator Module, which behaved in the same way as the 2-way Smart Switch. Consequently, both development processes were the same. This means that both devices were registered inside the *zigbee-switch* Edge Driver with the corresponding fingerprints and device profiles. However, a first problem was raised in terms of fingerprint identification. The ZigBee firmwares of these devices have been differentiated by Vimar, although the functionalities were the same. However, the Smart Actuator Module used the same ZigBee *model id* of the 2-way Smart Switch. Consequently, during the Edge Driver configuration phase, it was not possible to differentiate one device model to the other. This condition was critical in terms of device maintainability, because if a sub-driver was needed for future improvements, both devices would have shared the same fingerprint. After identifying this problem, we opted for a *model id* change at firmware level. This change was expected, because from the initial studies an Edge Driver requires these information to differentiate drivers. However, we note that this is not the case for different manufacturers, because the *manufacturer id* helps to avoid *model id* collisions. Moreover, this constraint for the ZigBee protocol may cause compatibility problems in foreign IoT ecosystems, whenever the model id is exploited to differentiate device models.

Actuator with Power Metering. The Actuator with Power Metering required the *On/Off* cluster and the *Electrical Measurement* cluster (i.e. Cluster ID: `0x0b04`). This last cluster was used to measure Watt power detected by the actuator. In fact, this actuator was supposed to be attached to plugs to collect power consumption. From the SmartThings ecosystem, a corresponding capability called *powerMeter* was used to retrieve this information. In this case, the development process required the use of a sub-driver (i.e. *zigbee-switch-power*) to implement the functionalities of this device. However, a custom *sub-sub-driver* was employed to satisfy extra requirements. As a matter of fact, the sub-driver implemented a power measurement formula to calculate the total Watt consumed. This formula has been changed to adjust the measurement output with the correct unit of magnitude. Therefore, we used a custom sub-sub-driver to fix the behavior by overriding the function handler with a custom function, fixing the output. Snippet 3.3 shows an excerpt of the original Edge Driver business-logic calculating the power measurement.

```
-- zigbee-switch-power driver (init.lua)
local function active_power_meter_handler(driver, device, value, zb_rx)
  local raw_value = value.value
  local divisor = device:get_field(constants.ELECTRICAL_MEASUREMENT_DIVISOR_KEY) or
  ↪  10
  raw_value = raw_value / divisor
  device:emit_event(capabilities.powerMeter.power({value = raw_value, unit = "W"}))
end
```

Snippet 3.3: Example of a function handler extracted from a *init.lua* sub-driver (i.e. *zigbee-switch-power*). This function parses the value of the power measurement and emits an event to the SmartThings ecosystem with the corrected output reading.

Roller Shutter Switch and Switch Module. The Roller Shutter Switch and the Roller Shutter Switch Module are two devices used to control window shades. These IoT devices are fairly more complex in terms of functionalities, with respect to the previously analyzed devices. The ZigBee cluster required for both devices was the *Window Covering* cluster (i.e. Cluster ID: `0x0102`), which handled several different attributes concerning lift and tilt positioning (e.g. window shades limits, current values). The SmartThings ecosystem required a corresponding capability to define the supported functionalities. In this case, even though tilt and lift positioning was supported by Vimar devices, only lift positioning was available as a capability. In details, three capabilities were used:

- *windowShade* was used to support open, close and pause commands of window shades;
- *windowShadeLevel* was employed to support a specific positioning of the window shades at a given level (e.g. 100 corresponds to open, thus setting the level to 50 moves the shades to a half-open position);
- *windowShadePreset* was used to allow the user to set a pre-defined level to open window shades.

These three capabilities have been defined inside the Edge Driver using function handlers. In this case, an existing driver was available to support Vimar devices

features. Therefore, we created a new sub-driver from the *zigbee-window-shades* driver. This Edge Driver required a complete re-implementation according to Vimar Roller Shutters behaviors.

A first problem that emerged from the implementation was the lack of *current lift values updates* provided by the Vimar device. In details, when a shade level was set by the user from the app, the Roller Shutter switch did not sent any kind of status update until the destination was reached. We noted that this behavior was accepted by the ZigBee standard. However, it was not accepted by the SmartThings mobile application because when an update status event was missed, a timeout error showed up inside the application. This became a problem to be fixed, thus avoiding bad user experience for this integration.

Another problem was found in the fingerprint for the devices recognition. The Roller Shutter switches shared the same problem with the 2-way Switch and the Actuator Module, i.e. the *model id* resulted in a collision. Therefore, this issue required a ZigBee firmware update to differentiate both devices, even though the same Edge Driver was employed in both integrations.

Discussion. We have analyzed each Vimar device of the case study in order to provide integration solutions. As discussed above, we identified new emerging problems regarding the local middleware implementation. As a matter of fact, these problems have been addressed according to company choices, whose purpose aimed at reducing firmware modification as less as possible, due to higher costs related to development, tests and firmware release. We have classified four emerging issues during the integration process.

- **Identification**. The first issue is related to the identification of Vimar devices. The 2-way Smart Switch and the Smart Actuator Module required a firmware update in order to differentiate the *model id*. In the same way, also the Roller Shutter Switch and Roller Shutter Switch Module required this modification. This issue was strictly related to the SmartThings ecosystem, which was not capable to use other attributes to differentiate IoT devices with the same *model id*. Consequently, a 3rd-party manufacturer must check whether its devices have *model id* collisions in the ZigBee *Basic* cluster.

- **Edge Driver and firmware version**. The second issue concerns the bind between an Edge Driver and the ZigBee firmware version. In this case, when the 3rd-party device has a firmware update, the Edge Driver must continue to work. However, if the firmware adds extra functionalities (e.g. new ZigBee clusters), a 3rd-party manufacturer must edit the existing edge driver to differentiate the new firmware behavior. Since a SmartThings fingerprint exploits the *model id*, the 3rd-party manufacturer must change the *model id* to provide this difference. Consequently, SmartThings should support different ways to differentiate the same device model based on the firmware version. For example, the *Basic* cluster has two attributes that can help to differentiate firmware versions: *Application Version* (i.e. Attribute ID: `0x0001`) and *SWBuildID* (i.e. Attribute ID: `0x4000`). This issue is important because when a 3rd-party vendor provides a new firmware update to commercialized product, the firmware update might be delayed by device owners. As a result, this event may cause inconsistencies to 3rd-party device behaviors.

- **Expected vs Actual behaviors**. The Roller Shutter Switch and Roller Shutter

Switch Module required a complete new sub-driver to handle window shades control in the SmartThings app. Furthermore, the expected behavior of the app is completely different with respect to Vimar device behaviors. Even though these devices were also ZigBee certified, timeout errors raised up in the SmartThings mobile app due to the missing shade level update event. Nevertheless, SmartThings documentation did not provide any kind of help in terms of expected behaviors. Consequently, in the future, SmartThings should provide a way to demonstrate the actual application behavior with respect to the integration solution.

- **Sub-driver dependencies**. The last issue concerns sub-driver dependencies. As previously discussed, Edge Driver's hierarchy brought a big change in the context of SmartThings driver solutions. However, a hierarchy approach to organize drivers and sub-drivers might break compatibility when the root driver code – or the parent sub-sub-driver code – is modified. As a result, the hierarchy creates a strong dependency, which can lead to complex driver maintenance. Therefore, SmartThings should supervise Edge Drivers integration to limit manufacturers in the creation of a high level of depth.

These problems have been partially addressed during the realization of Vimar device integrations. As noted, SmartThings integration may require extra steps for 3rd-party vendors in order to make the Edge Drivers working. A firmware update might be required to integrate expected behaviors of the SmartThings application. On the other hand, changing the attributes at protocol level might cause incompatibility with foreign IoT ecosystem integrations. Therefore, even though at a first glance the Hub-connected integration was expected to be faster in terms of development and cheaper in terms of resources, emerging issues resulted critical for user functionalities and future maintenance. The lack of feature support (e.g. no tilt positioning for the roller shutter switches) depended on the SmartThings capabilities. Future maintenance, instead, must be addressed correctly from the SmartThings team to avoid fingerprint collisions.

At the time of the experiments, an extensive study has been made with the company before proceeding with the implementation reported above, since we identified a set of features and tests for each device. We report the results of these tests in the last section of this chapter to present quantitative results of the integration. Moreover, these tests compare the Edge Driver integration with the initial study made with the legacy Device Handlers. The purpose of this report is to highlight the feature set covered by each solution, thus providing full interoperability of Vimar devices in the SmartThings ecosystem. Before proceeding to the results, we present CS-B concerning the KNX IoT client creation for Cloud-to-Cloud integration.

3.4 Designing a KNX IoT client

In this part, we investigate CS-B to design a KNX IoT client for Cloud-to-Cloud integration. We start analyzing an overview of the client requirements and client properties. Then, we provide a study on the KNX IoT client architecture employed for the integration. This study highlights the use of the semantics provided from different interfaces. We note that the internal Vimar semantics and data format is not analyzed in details, due to company policies. The goal of this case study is to trace the client functionalities to support 3rd-party Cloud-to-Cloud integration. Moreover, we analyze

the workflow of the client employed for Cloud-to-Cloud integration. In the end, we identify emerging problems from the point of view of a 3rd-party integrator.

3.4.1 Requirements of a KNX IoT client

In this section, we analyze requirements regarding the functionalities of the client. These requirements concern the nature of the client, the expected 3rd-party integrator needs and the design choices from a developer viewpoint. First of all, at the time of the analysis we classified the requirements based on the 3rd-party KNX IoT standard APIs:

- The client should have the *basic functionalities* to handle HTTP requests to REST APIs. These functionalities concern the API end-point access through the HTTP specifications. In this case, an error handler would have been included, thus covering multiple API responses.
- The client should provide *security mechanisms* according to the OAuth standard protocol in the KNX IoT specification. This includes the realization of an authentication handler able to secure API interactions.
- The client should provide *abstraction layers* to simplify the 3rd-party integrator work. These layers can be used to manipulate data at different levels by using a specific semantics to interpret symbols. For example, at a lower level data is untouched, meaning that no conversion of symbols is employed. In higher levels, data is manipulated according to a specific semantic, so as to provide different programming objects for the corresponding semantics. This means that we expected data in a form of a JSON payload in the lowest layer, while at the highest layer we expected data in a form of a language-specific object with attributes.

In the following paragraphs, we discuss each requirement in isolation.

Basic functionalities. The KNX IoT client was supposed to handle REST API connection to access resources. This required the client to support basic HTTP handlers, so as to compose a request and receive a response. Consequently, an HTTP library that provides these functionalities was be included in the client. Moreover, the APIs required the use of a specific JSON data format to receive and send requests. We noted that the format of the JSON was regulated by the KNX IoT specification, which follows a standard called JSON:API [51]. This standard explains how to build APIs in JSON according to a specification schema. This standard is also recognized by IANA (Internet Assigned Numbers Authority) as a supported media type (i.e. `Content-Type` field) for HTTP requests. Consequently, the client was also supposed to support this media type to use the APIs.

Security mechanisms. In the context of a KNX IoT client, the OAuth protocol provides interaction mechanisms to secure the authentication in the KNX IoT 3rd-party APIs. In addition to OAuth, TLS (i.e. Transport Layer Security) [12] is employed by the standard to secure the communication between the client and the server. For this client, we identified two main operations to include inside the client.

- The first operation is the *protected resource access*. This operation leverages an *Access Token*, which is placed in the HTTP header field (i.e. `Authorization`

field) of an HTTP request. This token is based on the JWT industry-standard (JSON Web Token), which is defined in the RFC 7519 [50]. Moreover, the Access Token can have an expiration time, after which the API access is denied.

- The second operation is the *session refresh*. This operation takes advantage of the available API end-point (i.e. `/oauth/access`), which allows updating the session using an entity called *Refresh Token*. When a session is created with the OAuth server, a client receives an Access Token and, optionally, a Refresh Token. The Access Token is used to access protected API resources on every HTTP request, while the Refresh Token is used to get a new Access Token, when the Access Token expires.

Parts of these operations have been previously discussed in §2.4.3.

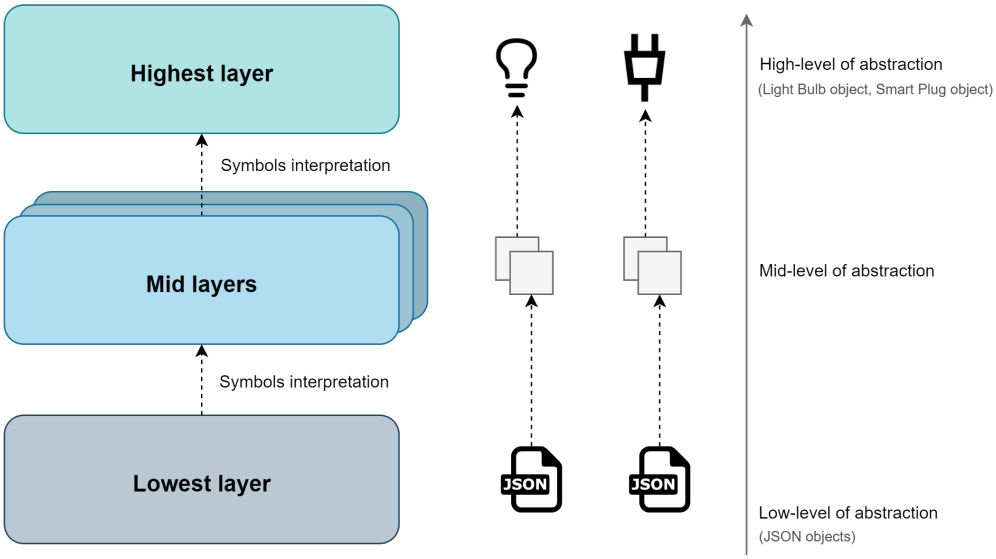


Figure 3.11: Example of a client architecture composed of abstraction layers. At the lowest layer, JSON data symbols are not interpreted, while at the highest layer light bulb and smart plug object definitions are available.

Abstraction layers. We designed the KNX IoT client to manipulate data in different levels of abstraction. This means that the client provides to 3rd-party developers multiple ways to interact with the APIs. Each way corresponds to a different level of complexity. The complexity is based on the semantics used to interpret data symbols. Therefore, a developer can choose to use either top-level objects or low-level objects for its integration. KNX entities (i.e. functions, datapoints, etc.) are provided by the KNX IoT 3rd-party API in a form of a JSON payload. The client can convert data to obtain complex objects organized in data structures to provide top level entities. A top level entity hides the complexity underneath, thus reducing the development work of 3rd-party developers. Hence, if HTTP requests are already handled by the client, 3rd-party integrators can concentrate on the top level interaction.

The example of a light-dimmer device may clarify the notion. In KNX IoT, a light dimmer can be interpreted as a KNX function, which has two datapoints. The former datapoint is used to indicate the status of the light (i.e. either *on* or *off*).

The latter datapoint is employed for the light intensity (i.e. a percentage from 0% to 100%). When the client requires a KNX function to the KNX IoT 3rd-party APIs, the two related datapoints can be discovered, because the REST API model provides extra knowledge of related resources, as discussed in 2.4.3. However, when the client interacts with the API, the resulting payload is in JSON format, whose symbols must be interpreted by the developer. Therefore, if the client provides a light bulb as an abstract object, a developer can directly use the light bulb – based on a KNX function object from the lower layer – with two properties (i.e. light status and light intensity). Additionally, the light bulb object can contain two functions to manipulate the two properties (e.g. set light status and set light intensity). We must note that this abstraction might help when creating a Cloud-to-Cloud integration. As a matter of fact, when the client is employed as a remote middleware, multiple data-models (e.g. the KNX IoT data-model and the 3rd-party data-model) must be included to convert top-level objects in corresponding requests. Therefore, the use of abstraction layers aims at satisfying multiple development approach, meaning that any layer can be used to accomplish the integration using different data-models. Figure 3.11 shows an example to represent data of a real object in different abstraction levels.

3.4.2 Sought client-side properties

In this section, we analyze client properties that we have singled out in §3.2.3. The client must met these properties because each property defined a characteristic required by the company. Therefore, we discuss how each property have been integrated in the client and whether extra properties were required to fulfill further needs.

Reusability. We identified *reusability* as the first characteristic to determine the nature of the client. A reusable (client) module is designed as an *SDK* or a *library*. These forms are easier to include in 3rd-party integration projects. As a matter of fact, a library can be included in either applications or serverless functions. In both cases, a library can also be hosted in a package manager, so as to be distributed across developers. Another important concern was the programming language to use. Nowadays, many programming languages can be adopted to create libraries. A report from the Northeastern University [62] shows that Python, Javascript and Java are the leading languages for 2022. Consequently, for CS-B, we chose Python as the programming language to build the library. Python offers simple syntax and easy learning curves, compared to other languages, as discussed in [93]. Moreover, it is actively used in data science by the academic community to create powerful applications thanks to the large amount of libraries available. Being Python actively used for commercial and academic applications, we expected our library to be re-usable by developers for 3rd-party integrations.

Standard compliance. The client was supposed to meet the KNX IoT standard specification, so as to be used for 3rd-party integrations. Therefore, the basic functionalities have been implemented according to the available REST API endpoints. This means that the HTTP request component of the client supports the media type specification and the authorization header, discussed in §3.4.1. We also introduced support for the CRUD operations available in each end-point and we tested the corresponding call to verify the correct response handling by the library. We also included error handling for the HTTP part. Therefore, when the client receives an error after a request, the error will be correctly handled, so as to be presented in a form of an object with

standard-based error codes.

Vertical extendability. We introduced the concept of a client-side entity, which can be vertically incremented to support multiple levels of abstraction. To clarify this concept, we note that the client is *vertical* since it is built as a stack architecture made of multiple components. Each component is a layer of the stack architecture with a corresponding level of abstraction. Moreover, the client is subject to *increments* (or extensions), because a new component can be added to the stack architecture to provide a new layer of abstraction for the objects underneath. However, since these components are added as increments, each component depends on the next lower component. A client with such properties meets the requirement discussed in §3.4.1, because new layers can be added in the architecture of the client to provide new levels of abstraction. However, this causes a strong dependency between components, which can be helpful to simplify the complexity, but can be dangerous when it comes to maintenance. As a matter of fact, with an increasing number of component, the level of abstraction increases. On the other hand, when a component in the middle of the stack requires modifications, this may cause incompatibility with the overlaying components.

Horizontal extendability. Lastly, we introduced the concept of an horizontally-extendable client. We used this concept to describe a client made of a stack architecture, whose layers can be extended in functionalities. This means that each layer can be modified to add new features in the same level of abstraction. Consequently, a 3rd-party integrator can decide to add new functions to a layer without touching the existing code base. This is a premise that leads to an objected-oriented approach to extend existing classes with custom functionalities. For example, a top level layer having a class representing a light can be extended to create a light dimmer.

After this analysis, we developed a KNX IoT client that meets the requirements from §3.4.1 and the properties just discussed. In the following section we analyze the corresponding architecture and workflow of the client to understand the interactions with the KNX IoT API server.

3.4.3 Client architecture and components workflow

In this section, we provide an analysis of the client architecture and the corresponding components workflow. We start with collecting the various components of the architecture. Then, each component is discussed and compared with respect to properties and requirements. Lastly, we present the components workflow of the architecture, simulating a Cloud-to-Cloud integration.

Client architecture. At first, we addressed the components of the architecture. In this case study, we used a 3-layer architecture to represent different level of abstractions. Each level is dependent on the lower level. Moreover, the aim of this architecture allows a developer to either use existing entities of the lower layer (e.g. a KNX function employs an HTTP request to retrieve its datapoints) or extend existing entities of the lower layer (e.g. a vendor-specific entity extends a KNX IoT Function to provide attributes and operations). The 3-layer architecture is described below.

- Traversing the architecture bottom up, the first layer is the **HTTP layer**. This layer is used to handle HTTP requests and authentication. In this layer, we covered the REST API communication to access API end-points and retrieve

resources in a form of JSON. When a developer requires a resource, an HTTP request handler is instantiated according to the available end-points. Then, the HTTP layer is also responsible for response and error handling. Therefore, the payload had to be properly organized in an object without any symbolic interpretation. In other words, the received payload remains in a JSON format, unless an error occurs. Furthermore, an error handler was employed to interpret HTTP error codes, thus providing error meanings. For example, if the client receives error code 404 after requesting a resource, the error handler must interpret the error code and the relative payload in output (i.e. *Resource not found*). The payload in output describes the error using a standard-specific JSON format. This layer is also responsible for the authentication header when a resource is accessed. Therefore, the creation of an HTTP header is delegated to the authentication handler. Then, this HTTP header is used by the HTTP request handler to perform the request.

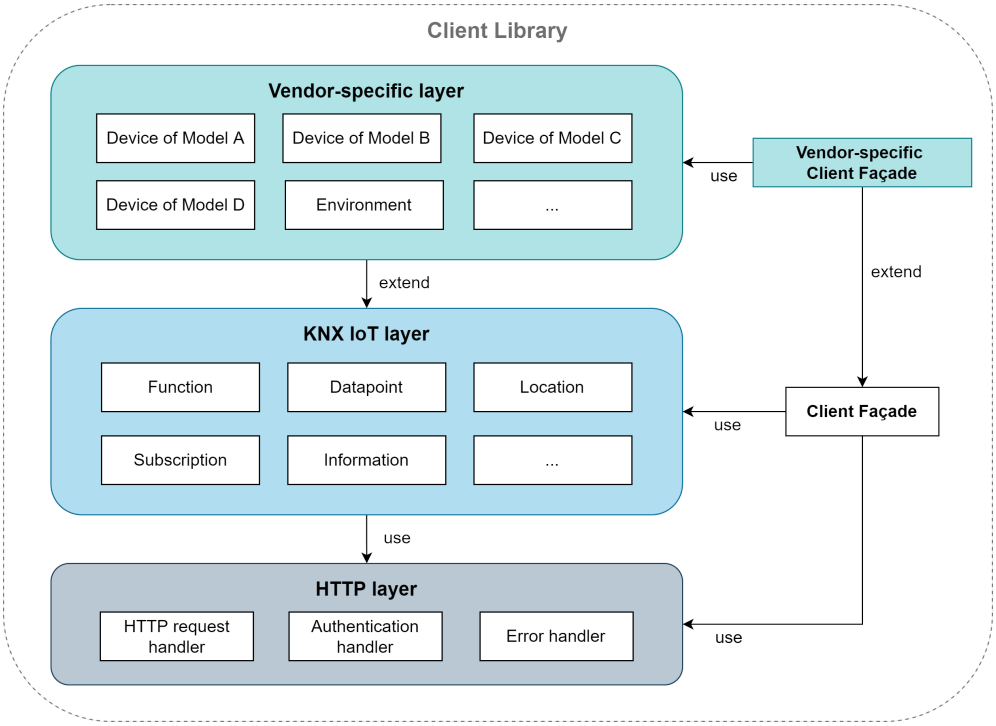


Figure 3.12: Client architecture overview with different layers to interpret the data-models.

- The layer above the HTTP layer is called **KNX IoT layer**. We employed this layer to include KNX IoT entities representation through objects. This layer ensures a simpler object manipulation for the 3rd-party integrator, because data coming from the HTTP layer is interpreted from a JSON format using the KNX IoT semantics. Therefore, KNX IoT objects can be used to handle attributes, properties and relationships for Functions, Datapoints, Locations and Subscriptions. For example, a Function object can interpret an HTTP response payload (i.e. a JSON payload from the KNX IoT API server) and re-use an existing HTTP request handler to retrieve its datapoints.

- The top layer of the client library is the **Vendor-specific layer**. This layer defines the vendor-specific objects based on the vendor-specific semantics. As a result, KNX IoT objects from the lower layer are extended in order to hide the complexity of the KNX IoT semantics to the developer. Therefore, the top layer is employed for device modeling. In other words, an abstract model of a device with the corresponding attributes and functions can be created in this layer to manipulate vendor-specific devices. Moreover, a developer that uses an object from vendor-specific layer can control a vendor-specific device without directly using the layers underneath.

In addition to this layer, a 3rd-party integrator can access client objects by using a *Client Façade*. A client façade provides functions and operations using the layers of library, while hiding the structure of the library to a utilizer. For example, a client façade may contain HTTP requests handlers for each KNX IoT API end-point and also functions to return KNX IoT objects. When a vendor-specific integrator uses this library, it can also extend this façade. The extended façade can be employed to cover objects from the vendor-specific layer, thus adding functionalities to the base client façade. Figure 3.12 shows the layers of the KNX IoT client architecture. From this figure, we included the vendor-specific façade that extends the – base – client façade.

Client workflow. We now illustrate an example of a client interaction based on a Cloud-to-Cloud interaction. The library is employed inside a client that acts as a remote middleware. We assume that the client is developed by the 3rd-party integrator, using the data-model of the integrator. We note that the *vendor-specific layer* can refer to the data-model of the KNX IoT API provider (e.g. Vimar internal data-model). The interaction of the remote middleware happens in 8 phases:

1. The integrator API requires a resource to the client.
2. The client uses a built-in function of a vendor-specific object of the client library to retrieve the requested resource. We assume that the vendor-specific layer extends the KNX IoT layer. Therefore, the data format is simply adapted for the KNX IoT layer.
3. Data passes from the KNX IoT layer to the HTTP layer. The data format is changed to create HTTP requests leveraging KNX IoT API data format.
4. The HTTP layer builds an HTTP request that is sent to the KNX IoT 3rd-party APIs to get the requested resource.
5. The KNX IoT API server replies with the requested resource.
6. The HTTP request handler passes the payload to a KNX IoT object, which handles the requested resource through data symbols interpretation.
7. Data symbols of the KNX IoT layer are interpreted by the vendor-specific layer, which returns the output of the function called in Phase 1.
8. The client receives the requested resource and forwards it to the integrator API, following the integrator semantics and data format.

Figure 3.13 captures the workflow explained above. We note that this workflow is a specific use case for a Cloud-to-Cloud integration. A 3rd-party integrator can also use

either the KNX IoT layer or the HTTP layer, according to its needs. However, at the higher layer the complexity reduces because the 3rd-party integrator does not need to understand the KNX IoT 3rd-party API interaction underneath. In other words, the client library handles the way – i.e. the standard – to communicate with the API server, while providing a simpler interface to the user with read-to-use functionalities.

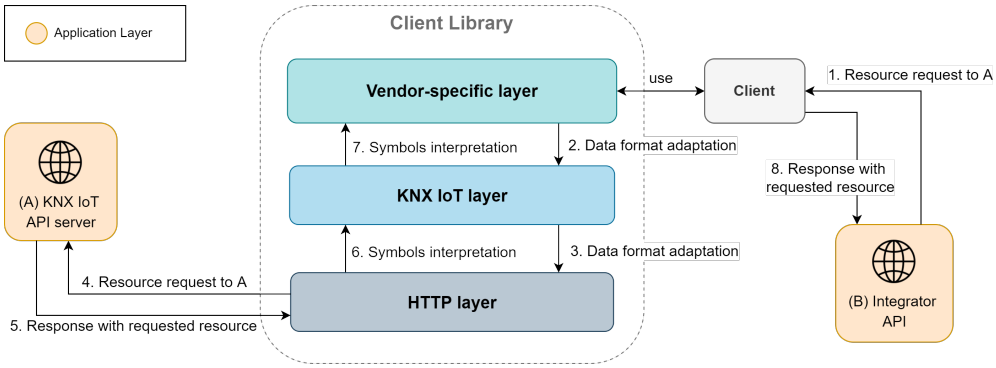


Figure 3.13: Example of a client workflow for a Cloud-to-Cloud integration.

Discussion. We designed and developed the client library to provide 3rd-party integrators a remote middleware solution, capable of interpreting data in multiple layers of abstraction. However, we noted that the highest layer (i.e. the vendor-specific layer) could be interpreted by 3rd-party integrators as an additional abstract layer to learn before realizing the integration. The aim of this library was to reduce the complexity of the API interaction and permit multiple development approaches. Therefore, a 3rd-party integrator could choose either to work with HTTP requests or use objects from the vendor-specific data-model – without knowing how the KNX IoT 3rd-party API standard works.

For CS-B, we noted that the vendor-specific layer of the library has been equipped with the Vimar internal semantics. Furthermore, the use of Vimar semantics for 3rd-party integrations simplifies the interaction and the support from the company for external developers. As a matter of fact, in order to understand a KNX IoT object, Vimar semantics is required, so as to correctly interpret data symbols. In this case study, we identified two problems concerning the KNX IoT client development:

- The first problem concerns *vendor-specific semantics*. A company can choose to hide its internal semantics to avoid exposing it to external developers. Although the KNX IoT client can work without the vendor-specific layer, the symbols interpretation from the KNX IoT layer may require a knowledge of the vendor-specific semantics. Therefore, a company that offers KNX IoT 3rd-party APIs for 3rd-party integrations must use a data-model that can be employed publicly. For example, a KNX ecosystem uses a data-model called KIM (KNX Information Model) that can replace the vendor-specific layer in the KNX IoT client architecture.

- The second problem concerns *remote middleware performance*. In this case, a Cloud-to-Cloud interaction leverages the KNX IoT client to elaborate data in multiple layers. Nevertheless, the API performance may impact the user experience, because latency times are affected by internal abstractions to provide KNX IoT 3rd-party APIs.

The first problem had to be addressed by the company itself, which decided what data-model to use. On the other hand, we investigated the second problem using a test-bench to measure latency time.

3.4.4 Performance evaluation

We decided to measure with a test-bench the *latency time* of an API interaction with the KNX IoT client. This parameter is important because it helps to understand the performance of the KNX IoT client, affecting the user experience. As a matter of fact, when a user interacts with a mobile application to control an IoT device, a measurable amount of time passes before having a visual feedback from the app. Therefore, we measured the time that passes from the user interaction to the received notification. This composes the latency time that we wanted to analyze. In details, we provide a formula to measure latency:

$$L = T_O - T_I$$

where:

- T_I is the time when the KNX IoT API Server receives a request from a KNX IoT Client application.
- T_O is the time when the KNX IoT Client receives a notification when the device status changes, according to the request.

In order to provide these measurements, we used the KNX IoT client realized for Vimar and the corresponding KNX IoT 3rd-party APIs. Moreover, a Vimar IoT device was employed to trigger the device status notification.

- The **KNX IoT client application** was placed inside a machine with a Fiber Internet connection. The Internet connection speed provided for the machine was about 1000MBit/s in download and 300MBit/s in upload. We also note that the client application was hosted inside a machine via an Ethernet connection that directly connected to the Internet modem within a 10 meter distance. Furthermore, the client machine was located in the North-East of Italy.
- The **KNX IoT 3rd-party API server** was hosted remotely in a server with the same Internet speed available of the KNX IoT client. However, the server was hosted in the West of Europe.
- The **IoT device** was connected through a powerful gateway from Vimar using Bluetooth mesh technology in a 1 meter distance area. The powerful gateway was connected to the internet using a 2.4 Ghz Wi-Fi connection within a 5 meter range from the Internet modem.

For this experiment, the KNX IoT client had two components, that were used to measure the latency time. The former component was the *client application* – equipped with the client library –, which was used to send the request and register T_I. The latter component was the *webhook*. This component was used to register T_O upon receiving an IoT device update. Moreover, the webhook exposed a public HTTP end-point (i.e. */updates/*) that the KNX IoT 3rd-party API server invoked to send status updates. This operation was regulated by the KNX IoT 3rd-party API specification.

Experiment workflow. The experiment was divided in six phases that we summarize below:

1. The client sends a payload with a toggle command (i.e. on/off) to the KNX IoT 3rd-party API end-point (i.e. `PUT /datapoints/{datapoint_id}`) through the client library.
2. The API server receives the payload and elaborates the request. In this phase, the API server gives in response to the KNX IoT client an HTTP status code 204. This means that the server has received the request and no content is provided in response. Therefore, the KNX IoT client webhook starts waiting for the asynchronous notification that happens in the last phase.
3. The API server elaborates the response and sends a command to the targeted IoT device.
4. Once the IoT device changes its status, an update notification is sent to the server through the powerful gateway.
5. The status update notification is received by the server. Then, the notification is forwarded to the KNX IoT client webhook.
6. The KNX IoT client webhook receives the asynchronous notification and the interaction ends.

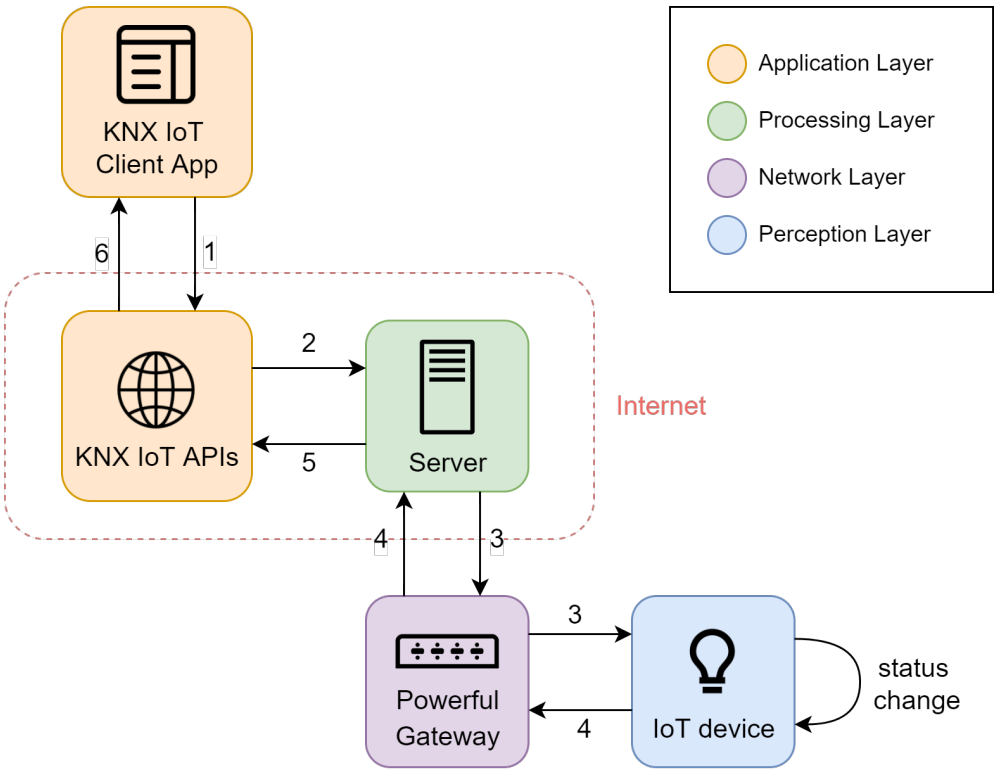


Figure 3.14: Test-bench workflow to capture the latency time of a KNX IoT client application changing the status of a device via Cloud-to-Cloud integration.

Figure 3.14 shows the interaction described above with the corresponding phases. We executed the experiment by repeating the requests every 10 seconds. In each request,

we toggled the status of the device. Therefore, the first request sent a *turn on* request to an IoT device, then the second request sent a *turn off* request to the IoT device, and so on. We organized 5 different sessions of 24-27 minutes in which we measured approximately 150 requests per session.

Expected results. Before executing the experiment, we reported an amount of expected latency of the integration solution. To do this, we studied the requirements from the other competitors. First of all, Google Home [69] reported three metrics to measure the quality of latency:

- the ideal latency is less than 0.2 seconds;
- the sufficient latency is between 2-5 seconds;
- the critical latency is above 5 seconds.

Amazon Alexa [32] did not report any kind of expected latency, but supplied some best practices to reduce latency time for the Smart Home integrations. Since Alexa uses skill to provide the Cloud-to-Cloud integration, the Alexa skill time limit is set to 6 seconds by default. However, a 3rd-party integrator can increase this limit, thus requiring an higher cost for the execution of the AWS Lambda function. SmartThings and Apple did not provide a minimum requirement regarding the expected latency for 3rd-party integrations. Speaking of academic literature, article [13] shows that the employ of cloud communication increases the total latency between a request and a response. As a result, we had to consider many factors that could be partially prevented for these experiments:

- **Internet connection.** The Internet connection of the KNX IoT client could have been subject to slowdowns caused by bandwidth contention with other connected devices. Therefore, for this experiment we used only the client machine connected to the Internet modem and we ensured that the Internet connection is stable. In the same way, the powerful gateway was the only device active and connected to the Wi-Fi modem.
- **Cloud location.** The location of the Cloud could have been one of the main responsible for high latency communication. Since the Cloud server was located in the Western Europe and the Client was located in the North East of Italy, we calculated the latency for a normal HTTP request to the KNX IoT server. This latency measurement helped us in learning the minimum expected latency between the client and the server for a minimal request. Figure 3.15 shows the latency from the client machine to the API server, in which the client receives an API response to get information on the supported API versions (using the `/.well-known/knx` end-point). Moreover, this request did not affect IoT devices, meaning that the KNX IoT API only replied with server-related information (e.g API version, base URL). We discussed below the obtained results.
- **Cloud computing time.** Depending on the architecture of the Cloud, the request could have had a higher latency time. In this case, the latency could not be measured directly for our experiments, since the Cloud infrastructure required a complex analysis of each single component. Moreover, due to company policies, we could not access the entire Cloud infrastructure. Nevertheless, we assumed that the time registered from the start of the interaction and the status change notification was sufficient to calculate the total amount of latency perceived by the user.

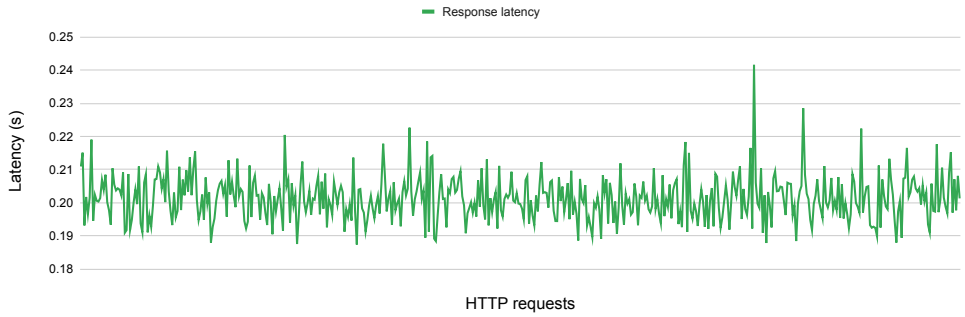


Figure 3.15: Latency measurement of Vimar KNX IoT 3rd-party APIs for a session of 18 minutes and 23 seconds. The latency is calculated between a client placed in the North-East of Italy and a server located in Western Europe. The client request does not affect the status of IoT devices.

N. of HTTP requests	500
Average latency	0.201s
Min. latency	0.187s
Max. latency	0.241s
Interval after each successful request	2.000s

Table 3.4: Latency results from Figure 3.15. For each request, the latency is calculated as the difference between the time of the API response and the time of the API initial query. After each API response, there are 2 seconds interval of idle time.

To conclude, we organized the measurement sessions with the company to trace the latency of the interaction affecting an IoT device. For this experiment, we used a Smart Actuator with Power Metering connected via Bluetooth to a Vimar gateway. Before starting the experiment, we measured the average latency from the machine to the server. As shown in Figure 3.15 and Table 3.4, we noted that the average latency exceeds 200 ms. Therefore, the minimum expected latency was greater than 0.200s. We decided with the company to adopt a metric similar to those in place for Google Home and Amazon Alexa. Table 3.5 shows the thresholds defined with the company.

Thresholds	Min latency	Max latency
Ideal	0.200s	1.000s
Good	1.001s	2.000s
Tolerable	2.001s	5.000s
Not tolerable	5.001s	10.000s

Table 3.5: Latency thresholds for the experiments to trace the quality of the measurements. These thresholds are defined with the company to understand the performance level of the APIs.

At the time of the case study, we noted that the KNX IoT API server provided by Vimar was under development during the experiments execution. Hence, we could expect missing notifications during the sessions, since the entire Cloud architecture could have been subject to slowdowns or other limits.

3.5 Discussion

In this section we discuss the results we obtained from experiments CS-A and CS-B. Both parts of the case study have been analyzed in the previous sections and many concerns have been discussed, related to issues, performance and design choices. For the SmartThings Hub-connected solution we expose integration tests results. These results aim at the analysis of the feature coverage between two drivers integrations developed for the SmartThings ecosystem. On the other hand, for the KNX IoT client we expose the API performance results. These results reports the average latency to complete a request using the KNX IoT 3rd-party APIs provided by Vimar. Lastly, for both case studies we compare emerging problems and integration complexity from a 3rd-party integrator perspective.

3.5.1 SmartThings drivers feature coverage

In this section, we report SmartThings test results related to the local middleware realization (CS-A). We have performed these tests internally with Vimar in order to highlight the completeness of the SmartThings integration, compared to the expected Vimar device requirements. Due to company policies, these tests cannot be exposed in detail. Therefore, we report a quantitative amount of tests based on the legacy driver solution (i.e. *device handler*) and the new driver solution (i.e. *edge driver*).

A test represents a feature of the Vimar IoT device. If the test passes, the feature works in the SmartThings integration. Otherwise, if the test fails, the feature does not work in SmartThings with the driver in use. Each test has been performed in the same environment following the same conditions. The goal of these measurements is to trace the number of working features for each Vimar device of this case study. This is important to show the satisfied requirements for a 3rd-party integrator that chooses one of the two driver solutions provided by SmartThings.

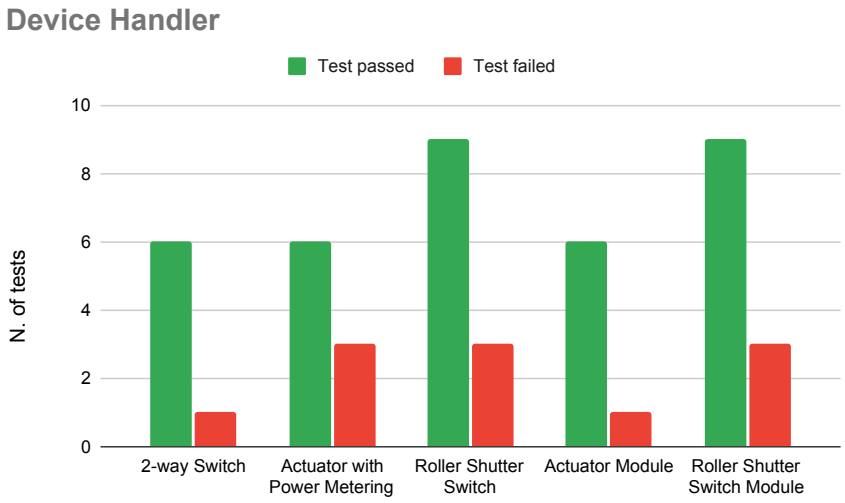


Figure 3.16: Test results for SmartThings Device Handler integration with Vimar devices.

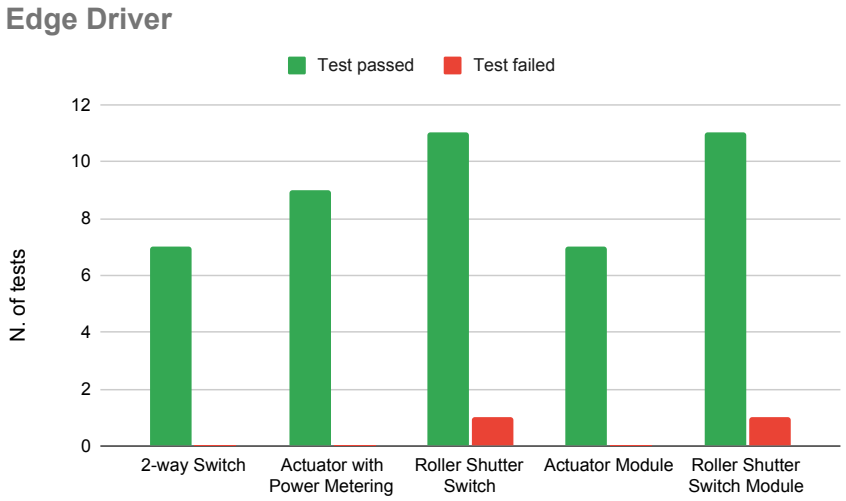


Figure 3.17: Test results for SmartThings Edge Driver integration with Vimar devices.

Figure 3.16 shows the results of Vimar integration tests using the Device Handler solution. From this graph, it is possible to see that each device has been tested with a different amount of tests. The 2-way Smart Switch and the Actuator module have the highest test passed percentage (i.e. 85,71%) with respect to the other devices. Each integration has at least one failed test. This means that the Vimar integration with device handlers partially achieves the overall expected functionalities of Vimar devices.

Figure 3.17 shows the results of Vimar integration tests via Edge Driver integration. In this case, only the Roller Shutter Switches have one single failed test. This feature is related to the expected behavior of the window shade widget in the SmartThings application, as discussed in 3.3.4. The overall test coverage achieves all the required features expected by Vimar device integrations.

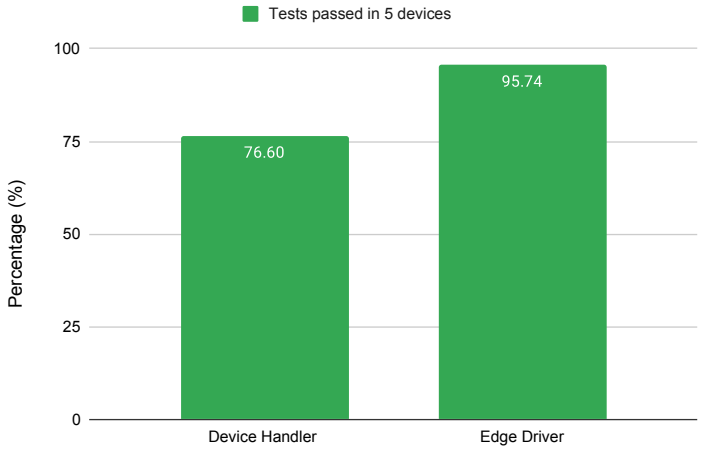


Figure 3.18: Percentage of tests passed comparing the SmartThings device handler solution and the SmartThings edge driver solution with Vimar devices.

Figure 3.18 collects the percentage of tests passed over a total of five devices. The Edge Driver integration has a higher number of features covered, while Device Handler integration is able to cover only three quarters of the total features.

3.5.2 KNX IoT client performance

In this section, we present the test results of the KNX IoT client interactions with the KNX IoT 3rd-party APIs (CS-B), as discussed in §3.4.4. We provided a total of five sessions to measure latency. In each session we executed the client to change the status of a Vimar IoT device. When the IoT device changes the status, the user gets notified. The amount of time from the initial interaction and the notification is the latency.

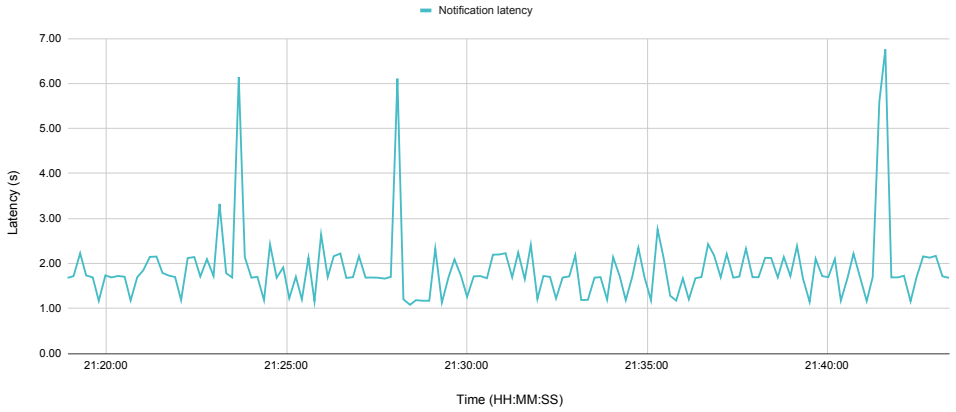


Figure 3.19: Test session #1 measuring KNX IoT client interaction latency. This test session is 24 minutes and 27 seconds long with a total of 140 requests.

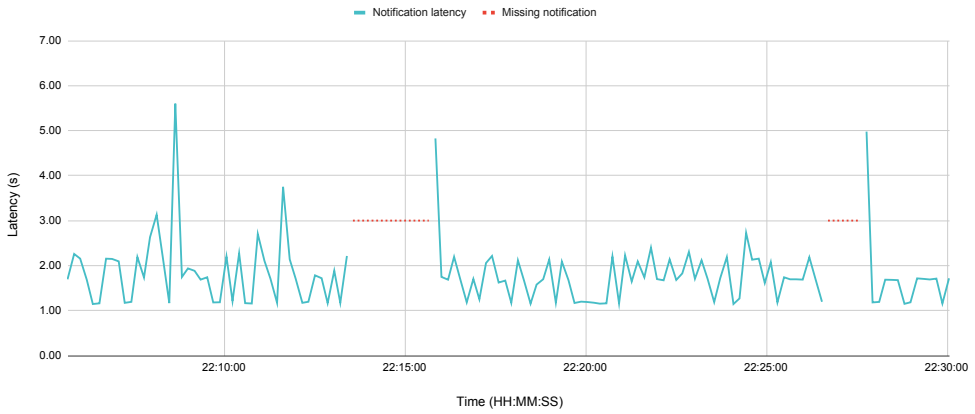


Figure 3.20: Test session #2 measuring KNX IoT client interaction latency. This test session is 25 minutes and 27 seconds long with a total of 147 requests.

Figure 3.19 captures the latency of the KNX IoT client interaction. The interaction requires an average time of 1.883s to perform the required action over 140 requests.

This average time is widely above the ideal threshold that we expected. Moreover, the graph shows a few spikes above the tolerable threshold, reaching at most 6.756s of latency. We traced back these spikes to the KNX IoT API server handling the incoming requests with a few slowdowns, which were network dependent.

Figure 3.20 captures the second session of latency measuring. The client registered an average interaction latency of 1.812s over 128 requests. From a total of 147 requests, 19 requests have not received a notification feedback, even though the KNX IoT client received a positive HTTP status code (i.e. 204). We considered these missed notifications to be caused by the KNX IoT 3rd-party API server, because the service notification component of the server was not working as expected. As a matter of fact, after investigating this issue, we found an accumulation of requests server-side. At the time of the experiment, we reported the problem to the company, so as to fix the issue for the future release. This event caused 2 missing notification periods of 126 seconds and 53 seconds, respectively. Furthermore, 3 spikes have been traced: one above the tolerable threshold (5.599s) and two below the tolerable threshold (4.825s, 4.977s).

Figure 3.21: Test session #3 measuring KNX IoT client interaction latency. This test session is 26 minutes and 50 seconds long with a total of 154 requests.

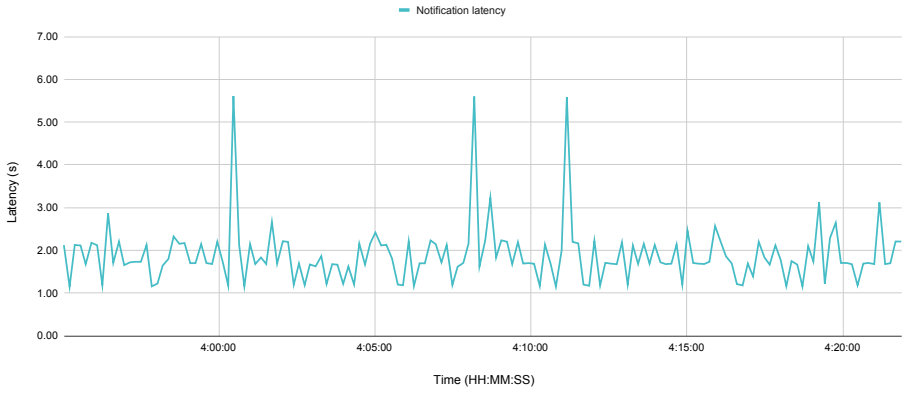


Figure 3.22: Test session #4 measuring KNX IoT client interaction latency. This test session is 27 minutes long with a total of 155 requests.

Figure 3.21 and Figure 3.22 shows a trend similar to Test session #1. As a matter of fact, each graph shows no missing notification and an average time of 1.804s (over 154 requests) and 1.883s (over 155 requests), respectively. In both cases, we have three major spikes for each graph reaching above the tolerable threshold.

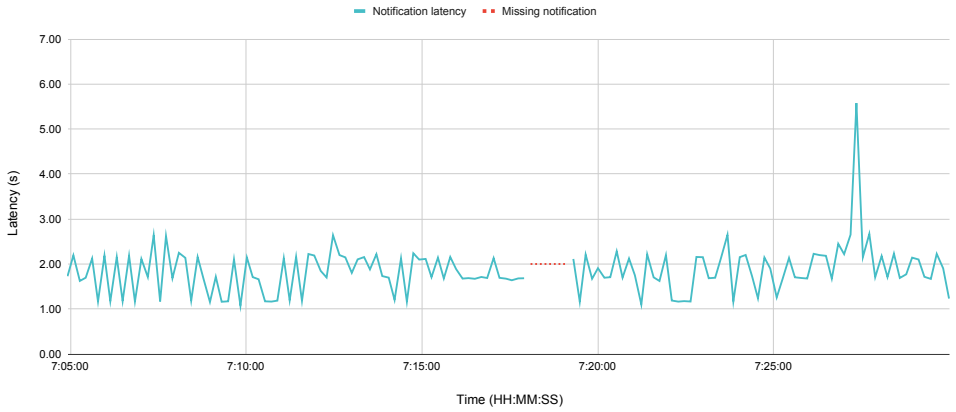


Figure 3.23: Test session #5 measuring KNX IoT client interaction latency. This test session is 25 minutes and 4 seconds long with a total of 144 requests.

Figure 3.23 captures the fifth session of latency measuring. The client shows an average latency of 1.848s over 137 requests. From a total of 144 requests, only 7 requests have not received a notification. This trend is similar to Figure 3.20, where a few requests missed the status change notification. In this case, the blackout notification period lasts for 84 seconds. In this session, we found one single spike above the tolerable threshold (5.576s).

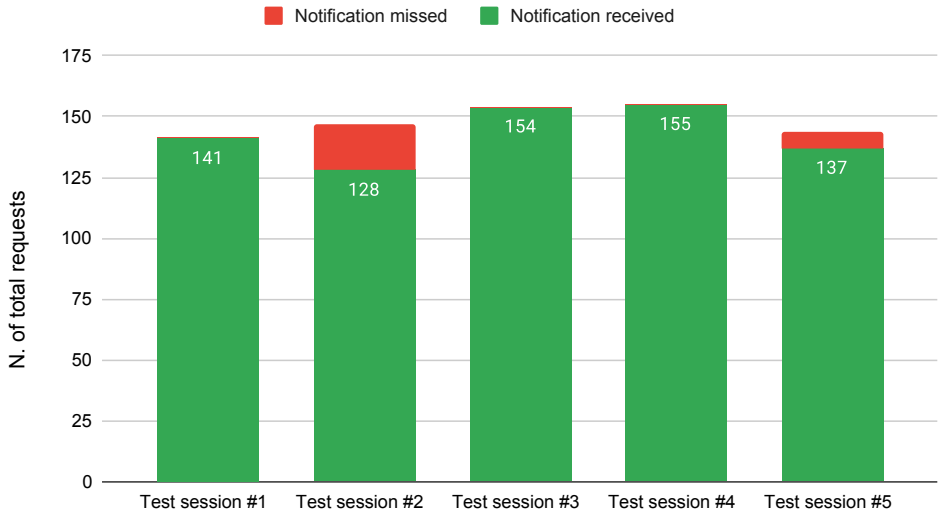


Figure 3.24: Total number of requests over five test sessions.

Figure 3.24 shows the total number of requests over five test sessions. We find missing notifications in only 2 sessions (i.e. Test session #2 and Test session #5). In the other sessions, each request has been correctly notified by the KNX IoT API server. Furthermore, Figure 3.25 shows that the average amount of time remains the same over the five test sessions. The average latency is under the 2 seconds threshold, being equal to 1.847s.

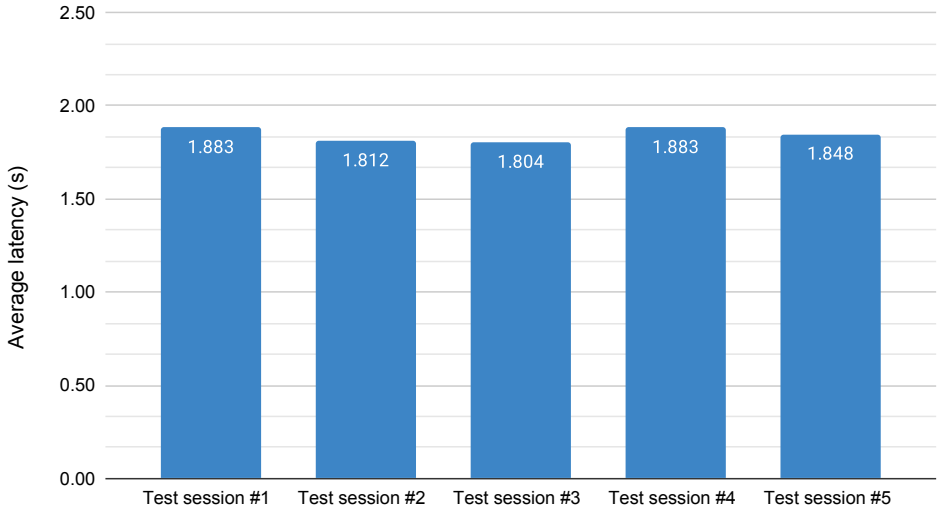


Figure 3.25: Average latency times over five test sessions.

3.5.3 Main findings

In this last section, we discuss the findings of the CS-A and CS-B. For both cases, we compare the common emerging issues to highlight pros and cons of each integration.

Case Study A. We have analyzed the integration of five different IoT devices to provide interoperability through a Gateway-connected approach. In this study, we have traced the phases of the integration from the perspective of a 3rd-party manufacturer. We have worked with the company to understand the best-fitting middleware solution provided by SmartThings, concerning drivers. In this study, we have concentrated on the features of each device to identify the level of Vimar device compatibility with respect to a foreign ecosystem. As shown in 3.5.1, the device handler solution is able to cover a part of the expected features (76.60% of tests passed) over five IoT devices. Conversely, the edge driver solution covers almost all the expected features (95.74% of tests passed). This result explains how SmartThings implementation improved with the introduction of new drivers – i.e. local middleware solutions – in the SmartThings-enabled Hub. Nevertheless, the Gateway-connected integration showed a few issues with respect to the initial considerations.

- **Development resources.** The first issue concerns development resources. At the time of the analysis, we expected a little amount of resources (i.e. developers, testers, tools) to employ the SmartThings integration. As a matter of fact, the ZigBee firmware was already available for Vimar devices. Therefore, the

driver development was considered the main activity to provide the integration. However, the identification process through SmartThings fingerprints required a firmware modification. This modification was necessary to avoid driver collision of two different models of Vimar devices in the ecosystem. Consequently, this led to higher costs for the integration development, even though the modification was minimal. Furthermore, we note that the IoT devices under study were already commercialized. Hence, a big amount of customers already owns Vimar IoT devices, and thanks to the SmartThings integration, five Vimar devices are now compatible with the SmartThings ecosystem. However, these devices require a firmware update before accessing the SmartThings platform, otherwise the interoperability with the foreign ecosystem is not assured. As a result, SmartThings was not capable to provide a detailed differentiation mechanism to prevent the fingerprint collision issue.

- **Features coverage.** The second issue concerns the feature coverage of the IoT devices. For the SmartThings integration, we expected the same amount of features covered in the ZigBee firmware. However, the SmartThings capabilities do not cover the entire feature set of ZigBee clusters. This lack of features was caused by the SmartThings platform, which requires developers to conform to the available capabilities. However, at the time of this writing, SmartThings is gradually opening to custom capabilities solutions, so as to allow manufacturers to cover the entire feature set as much as possible (see [35]). This is currently limited to non-certified SmartThings products, hence it is not fully supported in current integrations. For future device improvements, we expect an additional customization of Edge Drivers to include custom capabilities in IoT devices.

- **Behaviors.** We have identified a mismatched behavior between the driver and the SmartThings app with the driver for the Smart Roller Shutter Switch and the Smart Roller Shutter Switch module. When a user invokes a set level command from the app, a timeout error raises in the SmartThings app, even though the IoT device is correctly operating. As a result, we have provided a deep study of the SmartThings app to prevent this issue. Furthermore, the lack of documentation regarding the expected behaviors of the app required higher times to understand and correct the problem. With the help of the driver, part of this behavior has been corrected to prevent the timeout error. We note that this behavior mismatch is a clear example to show the compatibility limitation of an IoT ecosystem with foreign 3rd-party devices, even though a standard (i.e. ZigBee) is employed for the integration. Therefore, the platform should provide for the future a way to regulate the app behavior based on the driver behavior to avoid this issue.

Case Study B. We have analyzed the design of a KNX IoT client library to provide – from a platform provider perspective – a way for customers to use a proprietary ecosystem. Moreover, this integration was employed to understand the potentiality of a Cloud-to-Cloud integration based on the KNX IoT client. For this case study, we discuss two main issues:

- **Features coverage.** In our findings, we note that the development process required a deep study of the KNX IoT semantics, as well as the Vimar internal semantics to understand how to create abstract objects. Moreover, the integration of the three Vimar devices in the library (i.e. 2-way Switch, Actuator Module and Actuator with Power Metering) has been accomplished to provide a proof of concept of real devices. As a matter of fact, each device has been imported in the

library according to the objects of the Vimar semantics. The entire feature set of each Vimar device was also covered by the client library. Hence, a 3rd-party manufacturer adopting the library is able to leverage the highest abstractions (i.e. Vimar semantics) without learning the KNX IoT semantics.

- **API performance.** For this integration, we wanted to highlight the performance of the KNX IoT integration to test a real use case scenario by using the client library. As shown in 3.5.2, the overall results are pretty solid in terms of latency. As a matter of fact, the average latency time over 5 session of approximately 25 minutes is 1.847s, which is inside the good threshold defined in Table 3.5. However, the experiments showed a few notification misses caused by the API service notification component. Additionally, a few latency spikes above the tolerable threshold have been traced during the experiment. We want to point that these events were expected, being the KNX IoT 3rd-party API server in a pre-relase version at the time of the experiment. Nonetheless, in certain cases a user would still find delays or blackout periods, whenever is unable to receive a feedback of the devices through the Cloud. We note also that this condition can happen under different circumstances, such as high network traffic, absence of Internet connection and also low range distance of the IoT device to reach the Wi-Fi connection. To prevent part of these problems, a company can improve the performance of the Cloud solution to reduce latency times inside the Cloud architecture.

Analysis comparison. We draw two main conclusions from the experimental results discussed in this chapter.

First, we note that the Gateway-connected integration required a deep analysis to understand the driver behaviors and the feature set covered in ZigBee and SmartThings. On the other hand, the KNX IoT client for Cloud-to-Cloud integrations provides all the available functions covered in the company Cloud through the KNX IoT 3rd-party APIs. This ensures a higher versatility in terms of feature coverage, with respect to the ZigBee firmware, which is limited by the SmartThings driver capabilities.

Secondly, we want to highlight the performance of a potential Cloud-to-Cloud integration through KNX IoT 3rd-party API standard. The Cloud-to-Cloud integration provides latency times which depend on different factors. As discussed in 3.4.4, the connectivity distance between a gateway and the Cloud location plays a key role to reduce latency, because at a higher distance the response times of the APIs can be higher, as shown previously in Figure 3.4. Moreover, a Cloud component may be subject to slowdowns due to network traffic. As a matter of fact, a Cloud component is employed to serve multiple clients. Hence, with a high number of requests the connectivity might be unstable. Moreover, the Cloud-to-Cloud approach delegates the entire part of the computing time to the Cloud through abstractions, rather than the powerful gateway. This adds to the IoT architecture extra computing time to handle user requests. As a matter of fact, we must note that a Gateway-connected approach – that uses a local middleware – leverages the powerful gateway to handle computing operations. Therefore, the latency time of a powerful gateway depends on the processing unit (i.e. CPU) and on the network management capability of the network protocol. The Cloud connection that happens through the Internet is independent in this circumstance. In other words, an end user request sent with a mobile application that connects to the powerful gateway through a local connection (e.g. Wi-Fi, Bluetooth) can potentially be satisfied with a lower latency, being computed without the Cloud – i.e. passing through a minor number of components. On the other

hand, Cloud-to-Cloud integration provides higher latency times, since a single request has more levels of abstractions to go through before being satisfied.

To bring this thesis to a close, in the next and final chapter, we provide a retrospective discussion with respect to the local middleware and remote middleware solutions that we analyzed in this chapter. Then, we analyze the future of the IoT study field, regarding user-interaction performance, technology comparison and new emerging industry-standard protocols.

Chapter 4

Conclusions

4.1 Interoperability over standards

In this thesis, we have analyzed two middleware solutions, concerning a Gateway-connected integration and a Cloud-to-Cloud integration. The former integration concerns a SmartThings Hub-connected integration of five Vimar IoT devices using a local middleware solution. The latter integration pertains to the KNX IoT 3rd-party API standard integration of three Vimar IoT devices using a remote middleware solution. Both integrations leverage standard protocols to achieve interoperability. In this section, we analyze the findings on the ZigBee protocol and the KNX IoT 3rd-party API protocol to discuss the commissioning in a real-world use case.

Zigbee. The ZigBee protocol proved to be a versatile protocol to define device functionalities. As a matter of fact, after analyzing the firmware implementation with the company, we noticed that the ZigBee firmware covered most of the features expected for the device, compared to the Bluetooth firmware. Although the standard specifies several clusters and attributes to cover all possible functionalities of a device, three issues concern the ZigBee protocol in terms of interoperability.

- *Manufacturers require customization, even though the standard does not necessarily provides for it.* This is expected because a standard aims at the definition of common features adopted by most devices in the industry market. When an IoT device is designed with ZigBee, the required functionalities must be included in the standard, otherwise they should not be implemented with non-standard approaches. Nevertheless, board manufacturers allows the creation of non-standard clusters, which are not covered by the ZigBee Cluster Library. This approach harms the objective of a standard, despite offering customization for manufacturer-specific purposes.
- *The standard defines the features but not the behavior of the device.* The ZigBee standard covers most of the attributes and commands to interact with IoT devices. However, two IoT devices can potentially have different behaviors, even though the same ZigBee clusters are employed. For example, the Vimar Smart Roller Shutter Switch employed for the SmartThings integration updates the level of the shades at the end of the re-positioning operation, due to hardware limitation. A Roller Shutter Switch from a different vendor might be able to update this value during the re-positioning operation. In both cases, these devices

are compliant with the standard, having the same exact cluster (i.e. *Window Covering*). Therefore, in ZigBee, the behavior of the device is not fully defined by the standard. Looking at a platform perspective (e.g. SmartThings), the exposed UI should be able to provide all the device-specific functionalities, where the standard is limited. As a matter of fact, in SmartThings the local middleware (i.e. the driver) is able to fix the behavior of each IoT device. Therefore, if this condition is is met, 3rd-party IoT devices can behave correctly in foreign ecosystems.

- *Only a part of the standard is actually employed by platform providers.* With the SmartThings ecosystem, the Hub-connected integration requires the use of a ZigBee-enabled gateway. This gateway acts as a ZigBee Coordinator to manage the IoT devices connected in a mesh network. We noticed that a big limitation of the SmartThings platform was found with a Single-Point of Failure device, which is in fact the gateway itself. One of the main characteristics of ZigBee is *self-healing*. Hence, when a ZigBee Coordinator fails, it can be replaced with a ZigBee Router device. Nevertheless, SmartThings employs a provider-specific customization, which is exempt from the standard ZigBee specification. This approach limits the potentiality of the standard to satisfy platform provider needs. Moreover, SmartThings restricts the access of the ZigBee network, thus enabling the gateway to control device registration and interactions. As a matter of fact, the ZigBee Coordinator handles the devices installation and the corresponding message exchange within the ZigBee network. Therefore, the SmartThings-enabled Hub is the *only* responsible of the control of the ZigBee network. Moreover, the Hub exploits the Cloud to send status updates and receive commands. On this subject, we also note that routines are controlled over the gateway, even though the ZigBee Cluster Library provides support with the *Scenes* and *Groups* clusters. These clusters provide attributes and commands to create automations and routines using the ZigBee protocol. However, SmartThings uses its own implementation included in the gateway, so as to control the IoT devices.

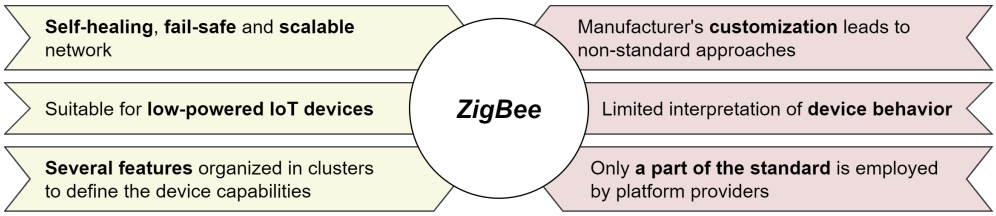


Figure 4.1: ZigBee integration pros and cons with respect to the results of §3.

As discussed above, the ZigBee standard is extremely versatile but potentially not exploited in most functions by platform providers. This is normal, because each provider aims at satisfying platform-specific needs. For example, we have seen that SmartThings uses ZigBee in a non-conventional way to register new devices and control the ZigBee network. However, this approach might lead to device firmware adaptation for 3rd-party integrators to satisfy platform-specific functionalities. Moreover, a 3rd-party integrator that certifies its devices with the ZigBee standard can count on a compliant product, even though the final commercial solution (i.e. the IoT device in the 3rd-party ecosystem) is only partially ZigBee compliant. This condition results in

an additional cost – in terms of development and maintenance – to adapt the device firmware for the ZigBee certification and the 3rd-party platform. Consequently, we note that this approach is advantageous for the IoT ecosystem control and customization, but it can be expensive to provide interoperability based on standard-compliant products.

KNX IoT 3rd-party APIs. From the analysis of the previous chapter, the KNX IoT standard has been employed to provide 3rd-party integrators a way to use the company IoT ecosystem. Our case study helps understand two main issues. On the one hand, we have analyzed the approach of the company to build KNX IoT APIs through a proprietary semantics. On the other hand, we have designed a way to integrate a Cloud-to-Cloud solution with the KNX IoT 3rd-party APIs, based on different development approaches. In this case study, the quest of interoperability is achieved in an abstract sense, because the application layer is taken under analysis. As a matter of fact, KNX IoT provides a REST API server interacting with the Cloud component, which belongs to the processing layer. We note that the solution accomplish interoperability, since the Vimar IoT ecosystem becomes available to 3rd-party developers. The role of these developers can be either to create their custom 3rd-party application solution or improve the availability of View Wireless devices in 3rd-party ecosystems. Therefore, small manufacturers with a proprietary ecosystem can decide to add Vimar support to their ecosystem, thus enlarging the available 3rd-party devices. Moreover, this solution enables Vimar to re-use its solution for a Cloud-to-Cloud integration with larger partners. Speaking of KNX IoT 3rd-party API standard, we noticed that the current adoption in large IoT ecosystems (e.g. Amazon Alexa, Google Home) is still absent. This is normal, because this standard is relatively new at the time of this analysis. However, the addition of this standard to adapt current manufacturer-specific semantics (and data-models) is supported by KNX. As a matter of fact, platform providers may employ their own product languages to handle device abstractions with custom semantics and data formats, as discussed in [54]. In our case study, two issues concern the KNX IoT 3rd-party API client.

- *Abstraction layers increase dependencies while hiding complexity.* The client library is structured in a 3-layer stack architecture, where each layer increases the level of abstractions. The mid layer (i.e. the KNX IoT layer) handles the KNX IoT objects using the HTTP layer (placed underneath) to populate data structures. This generates a dependency between the two layers, because the upper layer hides the complexity of the lower layers, so as to provide a simpler interface. In fact, a simple interface facilitates the usability of the client library for a developer, thus providing ready-to-use functions and objects. We should note that the standard, instead, defines the way to reach these functionalities based on a specification. Hence, the library allows a developer to customize each layer according to the required functionalities. However, this implies that the developer is aware of the interaction and the workflow between layers of the library, thus requiring a higher level of complexity. In other words, a developer that wants to extend the KNX IoT layer must learn the entities of the HTTP layer, thus incurring a dependency. As a result, the maintenance of the library is a critical operation, because it may break the compatibility of existing implementations. In this case, we note that the company provides interoperability of IoT devices through each layer of the library in different abstract forms. Hence, 3rd-party developers are able to handle IoT devices according to the layer used in the

integration.

- *Reusability over a single programming language.* The client library is provided to 3rd-party developers as a component of their integration solution. Each developer can decide how to design its integration, based on the requirements of its interface. In any case, we should note that the client is reusable as long as the library can be included in 3rd-party integrations. For example, a user that uses the Swift programming language to provide an application can use the KNX IoT client library, as long as it is compatible with the programming language. This implies that the client library must be replicated in different programming languages to support multiple integration solutions. However, this translates to a cost for the maintenance of each client library. In this case, we note that the company achieves interoperability by supporting not only the standard – which exposes a way to use functionalities –, but also the API client – which handles the way to use functionalities and provides a ready-to-use interface in a programming language.

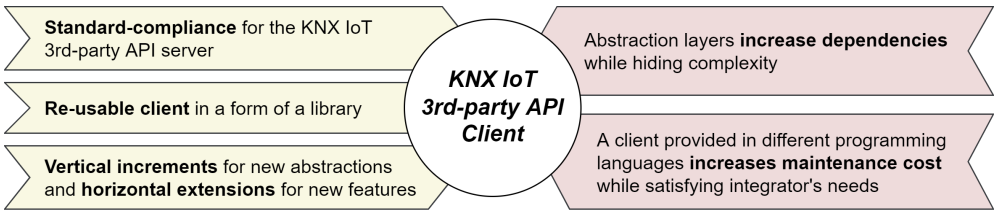


Figure 4.2: KNX IoT client pros and cons with respect to the results of §3.

From a company perspective, we note that a client library is an extra resource that must be maintained if provided as an integration solution. Nevertheless, it is also important to promote the integration solutions of 3rd-party developers. On the quest of interoperability, we have found that the client library achieves interoperability through different levels of abstractions and through re-usability. Both characteristics are the two critical features that a company must support for Cloud-to-Cloud solutions. Otherwise, the lack of abstraction requires a complex language interpretation, and the lack of re-usability – across programming languages or frameworks – does not cover the needs of 3rd-party developers.

4.2 A retrospective on the contribution of this work

The case study that we have presented in this work focused on the quest for interoperability through middleware-supported solutions. We note that the original middleware definitions from §1 are in fact appropriate for the role of the intermediary component in a Gateway-connected solution and in a Cloud-to-Cloud integration. In both solution, the collaboration with a company employed in the IoT context was necessary to trace a realistic scenario of the findings. Moreover, emerging problems have been discussed to highlight new integration challenges. In the following paragraphs, we discuss a brief retrospection of our work to trace what has been done and what requires a deeper inspection for the future.

Defining interoperability in IoT. We started by identifying the concept of interoperability inside the IoT world. This required a survey on IoT architectures, with a precise focus on smart-homes. We provided a 5-layer IoT architecture inspired from the academic literature to define an IoT ecosystem. Then, we have discussed smart-homes as an emerging IoT field with different modern challenges, regarding security, privacy and device-awareness. Furthermore, we have analyzed the concept of interoperability in smart-home IoT architectures. As a matter of fact, IoT architectures employ interoperability using software components. These components act as intermediaries (e.g. drivers) to control 3rd-party devices. Moreover, software components can control IoT devices through Cloud-connected ecosystems leveraging Cloud intermediaries (e.g. Cloud connectors, API client). From these premises, we have formalized the concept of middleware in the IoT context. The middleware analysis focused on the approaches to integrate different middleware solutions inside the 5-layer architecture. Consequently, with the first part of this work we have paved the way to the analysis of commercial solutions for 3rd-party device integration.

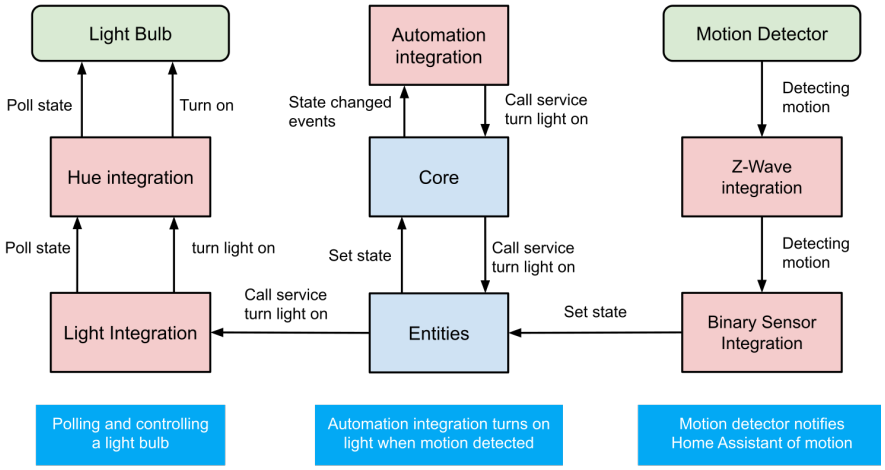


Figure 4.3: Home Assistant interaction example with Philips Hue light bulb and a motion sensor.
Source: [47]

Integration solutions. We have reported a survey of modern commercial solutions to provide integration of 3rd-party IoT devices. As shown in §2, we have studied major companies having a big influence on the IoT market. In this regard, we note that several platform providers can be worth studying so as to capture different approaches of IoT integration. For example, Home Assistant [47] is an emerging self-hosted platform to handle 3rd-party device integrations through Gateway-connected and Cloud-to-Cloud integrations. This platform employs a server that acts as a powerful gateway to manage IoT devices inside the house. The server is also the provider of applications and Cloud-to-Cloud connections, thus enabling the user to avoid data externalization. Hence, with this solution, the IoT platform can potentially be controlled offline, without requiring an Internet connection – as long as Cloud-to-Cloud integration are not employed. Similarly, Phoscon [66] from Dresden Elektronik is another self-hosted platform to control 3rd-party IoT devices through ZigBee-enabled devices. Furthermore, we note that several companies are also offering Platform-as-a-Service solutions (PaaS) to minor

companies and device manufacturers. For example, Microsoft Azure IoT Suite [61] and Particle [29] offers PaaS solutions to build a custom IoT platform belonging to a wide range of business model, in addition to smart-home devices.

Company case study. We have divided the company case study in two parts to highlight two sides of the challenges of interoperability. On the one hand, we acted as external 3rd-party integrators, adding compatibility to IoT devices in a foreign platform. On the other hand, we acted as platform providers, designing a way to interact with the company platform for 3rd-party integrators. In each approach, we employed a different mind-set to overcome the interoperability problem. Indeed, the former part of the case study required a deep analysis of the Gateway-connected solution from SmartThings and IoT devices from the company. The latter required a study of the KNX IoT language in addition to the company data-model. From our analysis, we had the opportunity to study different driver solutions regarding the SmartThings integration, so as to trace feature coverage, complexity and compatibility problems. We note that the Device Handler solution could have been an alternative to Edge Drivers as long as a custom independent driver was developed to avoid re-using an existing Device Handler. This would have helped in terms of future maintenance to reduce dependencies with existing device handlers. Nonetheless, most functionalities would have not be compatible, as compared to the Edge Driver solution. Hence, the Edge Drivers are still a valuable trade-off in terms of maintenance and feature coverage, at the cost of a moderate level of dependency (i.e. hierarchy of drivers). Regarding the second part of the case study, we developed a client library capable of interacting with APIs. Moreover, the client was also capable of receiving status updates of the IoT devices in a reasonable latency, thus measuring the performance of a Cloud-to-Cloud integration. We note that the alternatives to a client library based on a stack architecture were limited. In fact, a straightforward integration would have required a custom solution to directly bind one interface to the other. This solution can be employed for a single integration. Otherwise, multiple integrations require re-usable components to handle objects from the same interface (i.e. KNX IoT 3rd-party APIs) with the correct data-model interpretation. Therefore, a client library reduces code duplication. Moreover, when the KNX IoT interface is updated, the modifications of the middleware solutions are also reduced, being the library re-usable in multiple integrations (e.g. if several Cloud-to-Cloud integrations using the client library require an update, then the client library can be updated once and deployed to each integration, reducing time and costs).

The concept of middleware. After analyzing different middleware solutions, we stepped into a real use case scenario to study the profound nature of the middleware. Our findings shows that the concept of middleware can be employed in different context, belonging to Cloud and IoT architectures. Article [7] shows that in modern applications the middleware is closely connected to the concept of APIs. In detail, the authors of the article explain that a middleware can simplify sophisticated applications since developers can focus on the business-logic and the interactions of systems, besides components communication. In fact, the KNX IoT client library addresses the communication with the KNX IoT 3rd-party APIs while providing a modular and abstract way to handle systems interactions. Moreover, the business-logic of a remote middleware handles the data interpretation to provide control over interface-specific objects, thus interpreting correct device behaviors. Speaking of local middleware solution, this approach is also equivalent to driver integration. In fact, an edge driver describes the business-logic through function handlers to rule an IoT device according to different interfaces.

Future improvements. In this work, we have analyzed only two solutions regarding Gateway-connected and Cloud-to-Cloud integrations. We note that a Direct device integration would help understand a different local middleware implementation, as compared to Gateway-connected integration. Indeed, Direct device integration requires a local middleware equipped inside the device itself to work directly with either the Cloud or a mobile device. Therefore, the IoT device must be equipped with a Wi-Fi (or Bluetooth) module. Moreover, the IoT device firmware must employ the intermediary component to provide an interface (e.g. MQTT) towards the foreign IoT ecosystem. Nonetheless, building a single device that is capable of all three integrations requires a lot of time and work, because many different components of the IoT architecture must be employed to accomplish each integration. Eventually, this integration would be helpful to trace performance results. For example, in all three integrations, the latency that elapses in a request between the perception layer and the application layer could indicate which solution has the fastest response time.

Speaking of the 5-layer architecture, we note that the business layer does not directly concern the interoperability of an IoT device, being the highest layer of the stack. However, we can interpret the business layer as an additional part of the IoT integration, which is worth studying to analyze the added-values in the architecture. In fact, an IoT ecosystem is able to collect data that can be employed to provide statistics through the business layer. Article [5] shows that large-scale data processing units can be employed in Smart Buildings to provide energy management and HVAC (Heating Ventilation and Air Conditioning) management. These applications leverages data exploration through algorithms involving Machine Learning, to produce prediction models, and also Data Mining, to extract meaningful information.

Speaking of protocols, the Connectivity-Standard Alliance has created a new emerging standard protocol called Matter, as discussed in §2. This new protocol provides interoperability with device manufacturers and platform providers. At the time of this writing, Google Home, Amazon Alexa, SmartThings and Apple HomeKit will support this new protocol in their integration through a Gateway-connected solution. For the future, we note that Matter is worth studying to understand how the protocol behaves compared to existing standards. Moreover, an analysis of this protocol can include feature coverage and also performance across different manufacturers solutions. In the following section, we want to give a brief overview of the main characteristics of Matter to show the direction of a new emerging standard protocol.

4.2.1 Matter: a new emerging standard

In this thesis, we have analyzed the impact of standards in the context of IoT integrations. Matter is one of the newest emerging standard that we previously discussed in §2. In fact, Google Home, Amazon Alexa and Apple HomeKit have introduced in their documentation the compatibility with this standard. However, we want to point out that, at the time of this writing, this standard is not yet commercialized. An online news article [58] reported the release date of Matter for the end of 2022, as also announced by the CSA. At the current state, the standard has been partially defined. In this section, we provide a few highlights of Matter concerning the architecture and the main solutions to provide interoperability.

Architecture overview. Matter aims at providing interoperability on top of existing IoT solutions. In order to accomplish this purpose, Matter works on top of the

IPv6-based transport protocols. Therefore, in a ISO/OSI stack architecture Matter is defined on top of the Host layers (i.e. Transport, Session, Presentation and Application layers). Figure 4.4 shows a stack architecture with different network protocols used by Matter. These protocols are defined by IEEE standards and belong to the Media layers (i.e. Physical, Data Link and Network layers). In fact, Matter is compatible with Wi-Fi and Ethernet to allow remote control to the user through the Internet. These protocols are also employed to control powerful IoT devices, which require an high network bandwidth (e.g. video cameras, intercoms). Furthermore, Matter is also compatible with Bluetooth LE and 802.15.4-based devices. We note that the ZigBee protocol is not directly compatible with Matter. In fact, Matter requires a bridge between a ZigBee Hub and a Matter-enabled Hub, thus adding Matter capabilities to ZigBee devices.

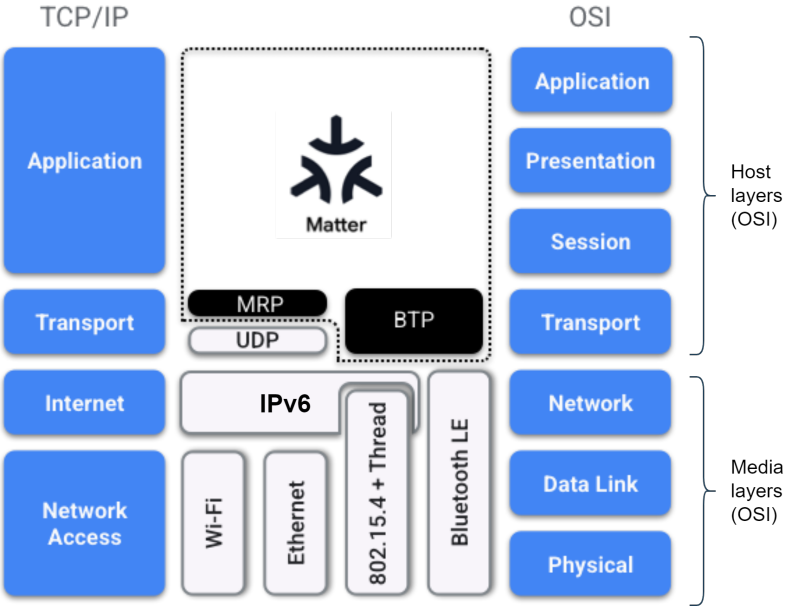


Figure 4.4: Matter architecture overview in parallel with the ISO/OSI stack layers and the TCP/IP model. Note: we specify *IPv6* (instead of *IP*), Host layers and Media layers, compared to the original figure.
Source: adapted from [91]

In addition to these protocols, Matter supports Thread [92], which is an emerging open-standard protocol that provides interoperability using the 802.15.4 networks. This protocol employs a mesh network to connect multiple IoT devices without requiring a vendor-specific hub. Thread can be integrated with existing hubs (e.g. Alexa-enabled Hub, Google Home devices), thus reducing the cost of the IoT network infrastructure. These hubs are called *Thread Border Routers* and act as a sub-network handlers, as shown in Figure 4.5. Moreover, each IoT device inside the Thread network acts as an access point, which handles the connections and the interaction with other IoT devices. Therefore, with Thread there is no Single-Point-of-Failure, because when an IoT device fails, it can be immediately replaced with any other node inside the network, similarly to ZigBee.

Discussion. We note that Matter leverages Thread to control low-powered IoT devices, while Thread employs the IEEE 802.15.4 standard to manage 802.15.4-based IoT devices. Therefore, Matter aims at the structure of the IoT application (i.e. the IoT device business-logic and the corresponding functionalities and security mechanisms), while Thread, Wi-Fi, Ethernet and BLE are employed to manage the connection with IoT devices. We want to point out that Matter achieves high compatibility with existing IoT devices facilitating implementation in commercialized solutions through software updates. In fact, since Matter is an application component, it is not hardware limited, hence the adoption rate is potentially high. On the other hand, Thread can be employed as an alternative of existing ZigBee/Z-wave devices, since the hardware (e.g. radio antenna) is compatible with the IEEE 802.15.4 standard. As a matter of fact, since ZigBee is not supported by Matter, a few problems arise in terms of interoperability, because manufacturers must provide bridge-based solutions to enable ZigBee devices in Matter networks. A bridge acts as a middleware de facto, which provides interoperability between two industry standards (e.g. Matter and ZigBee, Matter and Z-Wave). This component must be employed inside a powerful gateway to provide compatibility, thus opening a new scenario of a *Bridge-connected* (or *Gateway-to-Gateway*) integrations. For the future, the evolution of new integrations shall be analyzed to understand the implications and the performance of IoT solutions.

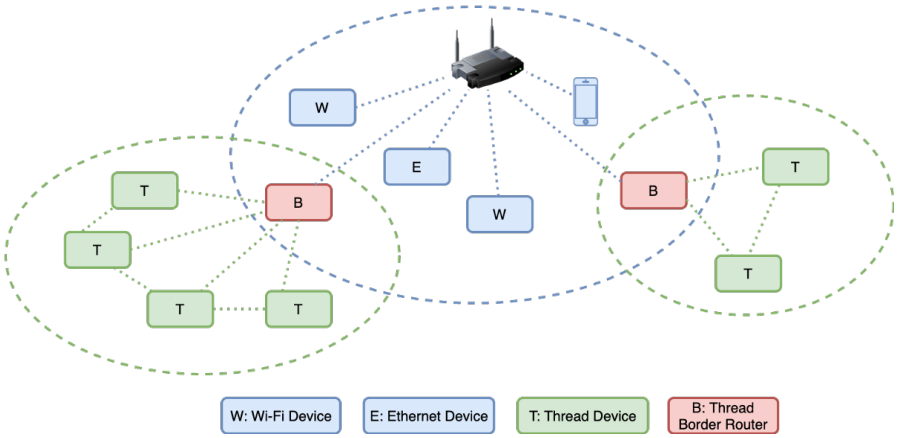


Figure 4.5: Matter topology example using Thread in a simple mesh network with etherogeneous IoT devices.
Source: [60]

4.3 Final notes

This thesis provides a complete analysis of 3rd-party integrations seeking interoperability of smart-home devices. We note that the growth of smart-homes brought several issues to the attention of the industry and the academic literature. In fact, new experiments and applications provide a way to address existing problems and propose new solutions. We have analyzed in detail the concept of middleware, as the main component that promotes interoperability in different forms. At the time of this writing, we have learned that the modern technologies are evolving in new forms to accomplish industry needs. In their turn, industry needs come from commercial and market needs provided by customers and clients. The aim of smart-homes is

to provide a new living comfort, while also addressing critical environmental issues through energy consumption control. Furthermore, social implications arise from the fact that the industry have control over data collected from the habits of customers, which is a particularly important aspect of the privacy research field. From the security viewpoint, IoT devices can be vectors of attack, especially in external integrations where security protocols are employed across different ecosystems. A recent research [7] shows that new IoT networks will be able to interact autonomously with their surroundings, without requiring a human intervention. The strict consequence of this evolution leads to the concept of *pervasive computing* – also called *ubiquitous computing*. Pervasive computing is defined as the entirety of contextualized services originating in a digital world, which are perceived through the physical world. Hence, computers are no longer identified as single devices or network of devices, because every IoT device provides a set of services contextualized to the device purpose. The way these devices will be employed in the future depends on the evolution of interoperability in the context of IoT.

To conclude, this thesis analyzes the IoT world to capture the workflow of architectural components, which allow warranting interoperability. Standard protocols are able to achieve interoperability, but sometimes they are unable to capture the advanced behavior of actual devices, leading manufacturers to customize their solutions. On the other hand, proprietary protocols enhance customization, at the cost of a minor adoption and support. Another important issue analyzed in the thesis is the interpretation of the data-model through data-format and semantics. We have seen that data-models are generally company-defined according to device requirements. In fact, the role of the middleware provides control over interfaces, while interpreting different data-models to define the business-logic of IoT devices. For the future evolution, we expect to see how the upcoming standards will be able to address the emerging challenges, and how far they will win the adherence of the industrial actors.

Bibliography

Articles

[1] Nasr Abosata et al. «Internet of Things for System Integrity: A Comprehensive Survey on Security, Attacks and Countermeasures for Industrial Applications». In: *Sensors* 21.11 (2021). ISSN: 1424-8220. DOI: 10.3390/s21113654. URL: https://www.mdpi.com/1424-8220/21/11/3654.

[2] Zahrah A. Almusaylim and Noor Zaman. «A review on smart home present state and challenges: linked to context-awareness internet of things (IoT)». In: *Wireless Networks* 25.6 (Aug. 2019), pp. 3193–3204. ISSN: 1572-8196. DOI: 10.1007/s11276-018-1712-5. URL: https://doi.org/10.1007/s11276-018-1712-5.

[3] Sharu Bansal and Dilip Kumar. «IoT Ecosystem: A Survey on Devices, Gateways, Operating Systems, Middleware and Communication». In: *International Journal of Wireless Information Networks* 27.3 (Sept. 2020), pp. 340–364. ISSN: 1572-8129. DOI: 10.1007/s10776-020-00483-7. URL: https://doi.org/10.1007/s10776-020-00483-7.

[4] Muthu Chellappa, Shanmugaraj Madasamy, and R. Prabakaran. «Study on ZigBee technology». In: (Apr. 2011), pp. 297–301. DOI: 10.1109/ICECTECH.2011.5942102.

[5] Abdellah Daissaoui et al. «IoT and Big Data Analytics for Smart Buildings: A Survey». In: *Procedia Computer Science* 170 (2020). The 11th International Conference on Ambient Systems, Networks and Technologies (ANT) / The 3rd International Conference on Emerging Data and Industry 4.0 (EDI40) / Affiliated Workshops, pp. 161–168. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.03.021. URL: https://www.sciencedirect.com/science/article/pii/S1877050920304506.

[6] Menachem Domb. «Smart Home Systems Based on Internet of Things». In: (Feb. 2019). DOI: 10.5772/intechopen.84894.

[7] Alexandros Gazis and Eleftheria Katsiri. «Middleware 101». In: *Commun. ACM* 65.9 (Aug. 2022), pp. 38–42. ISSN: 0001-0782. DOI: 10.1145/3546958. URL: https://doi.org/10.1145/3546958.

[8] Dimitrios Georgakopoulos and Prem Prakash Jayaraman. «Internet of things: from internet scale sensing to smart services». In: *Computing* 98 (Oct. 2016). DOI: 10.1007/s00607-016-0510-0.

[9] Won Min Kang, Seo Yeon Moon, and Jong Hyuk Park. «An enhanced security framework for home appliances in smart home». In: *Human-centric Computing and Information Sciences* 7.1 (Mar. 2017), p. 6. ISSN: 2192-1962. DOI: 10.1186/s13673-017-0087-4. URL: https://doi.org/10.1186/s13673-017-0087-4.

[10] Hyun-Wook Kim et al. «Development of Middleware Architecture to Realize Context-Aware Service in Smart Home Environment». In: *Computer Science and Information Systems* 13 (June 2016), pp. 427–452. DOI: 10.2298/CSIS150701010H.

[11] X. Krasniqi and E. Hajrizi. «Use of IoT Technology to Drive the Automotive Industry from Connected to Full Autonomous Vehicles». In: *IFAC-PapersOnLine* 49.29 (2016). 17th IFAC Conference on International Stability, Technology and Culture TECIS 2016, pp. 269–274. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2016.11.078. URL: https://www.sciencedirect.com/science/article/pii/S2405896316325162.

[12] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. «On the Security of the TLS Protocol: A Systematic Analysis». In: (2013). Ed. by Ran Canetti and Juan A. Garay, pp. 429–448.

[13] Dr. Kailash Kumar. «IoT-Edge Communication Protocol based on Low Latency for effective Data Flow and Distributed Neural Network in a Big Data Environment». In: *Microprocessors and Microsystems* 81 (2021), p. 103642. ISSN: 0141-9331. DOI: https://doi.org/10.1016/j.micpro.2020.103642. URL: https://www.sciencedirect.com/science/article/pii/S0141933120307894.

[14] In Lee. «The Internet of Things for enterprises: An ecosystem, architecture, and IoT service business model». In: *Internet of Things* 7 (2019), p. 100078. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2019.100078. URL: https://www.sciencedirect.com/science/article/pii/S2542660519301386.

[15] Jin-Shyan Lee, Yu-Wei Su, and Chung-Chou Shen. «A Comparative Study of Wireless Protocols: Bluetooth, UWB, ZigBee, and Wi-Fi». In: (2007), pp. 46–51. DOI: 10.1109/IECON.2007.4460126.

[16] Suk Kyu Lee, Mungyu Bae, and Hwangnam Kim. «Future of IoT Networks: A Survey». In: *Applied Sciences* 7.10 (2017). ISSN: 2076-3417. DOI: 10.3390/app7101072. URL: https://www.mdpi.com/2076-3417/7/10/1072.

[17] Marco Lombardi, Francesco Pascale, and Domenico Santaniello. «Internet of Things: A General Overview between Architectures, Protocols and Applications». In: *Information* 12.2 (2021). ISSN: 2078-2489. DOI: 10.3390/info12020087. URL: https://www.mdpi.com/2078-2489/12/2/87.

[18] Oleksiy Mazhelis and Pasi Tyrväinen. «A framework for evaluating Internet-of-Things platforms: Application provider viewpoint». In: *2014 IEEE World Forum on Internet of Things, WF-IoT 2014* (Mar. 2014). DOI: 10.1109/WF-IoT.2014.6803137.

[19] Anne H. Ngu et al. «IoT Middleware: A Survey on Issues and Enabling Technologies». In: *IEEE Internet of Things Journal* 4.1 (2017), pp. 1–20. DOI: 10.1109/JIOT.2016.2615180.

[20] Mahda Noura, Mohammed Atiquzzaman, and Martin Gaedke. «Interoperability in Internet of Things: Taxonomies and Open Challenges». In: *Mobile Networks and Applications* 24.3 (June 2019), pp. 796–809. ISSN: 1572-8153. DOI: 10.1007/s11036-018-1089-9. URL: https://doi.org/10.1007/s11036-018-1089-9.

[21] Mario Paredes-Valverde et al. «IntelliHome: An internet of things-based system for electrical energy saving in smart home environment». In: *Computational Intelligence* 36 (Nov. 2019). DOI: 10.1111/coin.12252.

[22] Sureshkumar Selvaraj and Suresh Sundaravaradhan. «Challenges and opportunities in IoT healthcare systems: a systematic review». In: *SN Applied Sciences* 2.1 (Dec. 2019), p. 139. ISSN: 2523-3971. DOI: 10.1007/s42452-019-1925-y. URL: https://doi.org/10.1007/s42452-019-1925-y.

[23] Yousaf Bin Zikria et al. «Next-Generation Internet of Things (IoT): Opportunities, Challenges, and Solutions». In: *Sensors* 21.4 (2021). ISSN: 1424-8220. DOI: 10.3390/s21041174. URL: https://www.mdpi.com/1424-8220/21/4/1174.

Books and Reports

[24] P. Naur and B. Randell. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968*. Scientific Affairs Division, NATO, 1969. URL: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[25] Mao-Yung Weng et al. *Context-aware home energy saving based on Energy-Prone Context*. 2012, pp. 5233–5238. DOI: 10.1109/IROS.2012.6385762.

[26] Ivana Podnar Žarko, Krešimir Pripužić, and Martin Serrano, eds. *Interoperability and Open-Source Solutions for the Internet of Things*. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-16546-2. URL: https://doi.org/10.1007%2F978-3-319-16546-2.

Web References

[27] *Add support for Matter in your smart home app, Video from Apple developer documentation*. URL: https://developer.apple.com/videos/play/wwdc2021/10298/.

[28] *Amazon Alexa Statistics, Facts, and Trends, PolicyAdvice*. URL: https://policyadvice.net/insurance/insights/amazon-alexa-statistics/.

[29] *An integrated IoT Platform-as-a-Service, Particle*. URL: https://www.particle.io/.

[30] *Apple Ecosystem Explained, Techjourneyman*. URL: https://techjourneyman.com/blog/apple-ecosystem-explained/.

[31] *Architecture overview, SmartThings developers*. URL: https://developer-preview.smartthings.com/docs/getting-started/architecture-of-smartthings.

[32] *Best Practices to Reduce Latency on your Smart Home Skill, Amazon developers.* URL: https://developer.amazon.com/en-US/blogs/alexa/device-makers/2020/11/Best-Practices-to-Reduce-Latency-on-your-Smart-Home-Skill.

[33] *Build with Matter, Connectivity Alliance Standard.* URL: https://csa-iot.org/all-solutions/matter/.

[34] *Cloud-to-cloud, Google Home developers.* URL: https://developers.home.google.com/cloud-to-cloud.

[35] *Custom capabilities, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/capabilities/custom-capabilities.

[36] *Develop Your Device App, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/direct-connected/device-app.

[37] *Developing apps and accessories for the home, Apple developer documentation.* URL: https://developer.apple.com/apple-home/.

[38] *Devices in Google Home ecosystem, Google.* URL: https://home.google.com/explore-devices/.

[39] *Edge Drivers Guide, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/hub-connected/first-lua-driver.

[40] *Enthusiasts Guide to HomeKit, Linkdhome.* URL: https://linkdhome.com/articles/complete-homekit-guide.

[41] *Get Started with ACK, Amazon Alexa developers.* URL: https://developer.amazon.com/en-US/docs/alexa/ack/get-started.html.

[42] *Get Started with Cloud Connected Devices, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/cloud-connected/get-started/.

[43] *Get Started with Direct Connected Devices, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/direct-connected/get-started.

[44] *Get Started with Hub Connected Devices, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/hub-connected/get-started.

[45] *Get Started with Mobile Connected Devices, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/mobile-connected/get-started.

[46] *Home Alone with HomeKit - Part I, Medium.* URL: https://vincent-coetzee.medium.com/home-alone-with-homekit-part-i-91667264f563.

[47] *Home Assistant website.* URL: https://www.home-assistant.io/.

[48] *Introducing New Works with Alexa (WWA) Requirements For Zigbee Devices, Amazon Alexa developers.* URL: https://developer.amazon.com/en-US/blogs/alexa/device-makers/2020/07/introducing-new-works-with-alexa-requirements-for-zigbee-devices.

[49] *IoT: Why the magic is in the ecosystem, TechTarget.* URL: https://www.techtarget.com/iotagenda/blog/IoT-Agenda/IoT-Why-the-magic-is-in-the-ecosystem.

[50] *Json Web Token (JWT), RFC 7519.* URL: https://www.rfc-editor.org/rfc/rfc7519.

[51] *JSON:API, a specification for building APIs in JSON, JSONAPI website.* URL: https://jsonapi.org/.

[52] *KNX Internet of Things, KNX Association.* URL: https://www.knx.org/knx-en/for-professionals/benefits/knx-internet-of-things/.

[53] *KNX IoT Schema Server, KNX Association.* URL: https://schema.knx.org/.

[54] *KNX IoT Semantics, KNX Association.* URL: https://support.knx.org/hc/en-us/articles/4402060368658-KNX-IoT-Semantics.

[55] *KNX IoT: the Summum of Interoperability, KNX Association.* URL: https://support.knx.org/hc/en-us/articles/4402060301458-KNX-IoT-the-Summum-of-Interoperability.

[56] *Local fulfillment, Google Home developers.* URL: https://developers.google.com/assistant/smarthome/concepts/local.

[57] *Local Home SDK, Google Home developers.* URL: https://developers.home.google.com/local-home.

[58] *Matter smart home standard delayed until fall 2022, TheVerge.* URL: https://www.theverge.com/2022/3/17/22982166/matter-smart-home-standard-postponed-fall-2022.

[59] *Matter, Google Home developers.* URL: https://developers.home.google.com/matter.

[60] *Matter: Thread Border Router in Matter.* URL: https://blog.espressif.com/matter-thread-border-router-in-matter-240838dc4779.

[61] *Microsoft Azure IoT Suite, Microsoft.* URL: https://azure.microsoft.com/en-us/solutions/iot/.

[62] *Most popular programming languages for 2022, Northeastern University.* URL: https://www.northeastern.edu/graduate/blog/most-popular-programming-languages/.

[63] *Nearly 70 percent of US smart speaker owners use Amazon Echo devices, TechCrunch.* URL: https://techcrunch.com/2020/02/10/nearly-70-of-u-s-smart-speaker-owners-use-amazon-echo-devices.

[64] *OAuth 2.0, Official OAuth website.* URL: https://oauth.net/2/.

[65] *Our mission, KNX Association.* URL: https://www.knx.org/knx-en/for-professionals/What-is-KNX/Our-mission/.

[66] *Phoscon App website.* URL: https://phoscon.de/en/app/doc.

[67] *Smart Home Development options, Amazon Alexa developers.* URL: https://developer.amazon.com/en-US/docs/alexa/smarthome/development-options.html.

[68] *Smart Home Development options, Amazon Alexa developers from Archive.org.* URL: https://web.archive.org/web/20220817145234/https://developer.amazon.com/en-US/docs/alexa/smarthome/development-options.html.

[69] *Smart Home FAQs, Google Home developers.* URL: https://developers.google.com/assistant/smarthome/faq#response-latency.

[70] *Smart Home View Wireless, Vimar.* URL: https://www.vimar.com/it/it/smart-home-15427122.html.

[71] *Smart-home ecosystem from Samsung, Official webpage, SmartThings.* URL: https://www.smartthings.com/.

[72] *Smart-home ecosystem, Amazon Alexa.* URL: https://www.amazon.com/alexa-smart-home/b?node=21442899011.

[73] *Smart-home ecosystem, Apple.* URL: https://www.apple.com/ios/home/.

[74] *Smart-home ecosystem, Google.* URL: https://home.google.com/welcome/.

[75] *SmartThings online repository with public Device Handlers, GitHub.* URL: https://github.com/SmartThingsCommunity/SmartThingsPublic/tree/master/devicetypes/smartthings.

[76] *SmartThings online repository with public Edge Drivers, GitHub.* URL: https://github.com/SmartThingsCommunity/SmartThingsEdgeDrivers/tree/main/drivers/SmartThings.

[77] *SmartThings Production Capabilities, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/devices/capabilities/capabilities-reference.

[78] *SmartThings Public APIs in preview, SmartThings developers.* URL: https://developer-preview.smartthings.com/docs/api/public.

[79] *Strangler Fig pattern, Microsoft documentation.* URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler-fig.

[80] *StranglerFigApplication, Martin Fowler Blog.* URL: https://martinfowler.com/bliki/StranglerFigApplication.html.

[81] *The new head of SmartThings discusses Matter and the future of the smart home, The Verge.* URL: https://www.theverge.com/23055296/samsung-smartthings-smart-home-matter-interview.

[82] *The OAuth 2.0 Authorization Framework, RFC6749.* URL: https://www.rfc-editor.org/rfc/rfc6749.

[83] *The Zigbee Alliance Rebrands as Connectivity Standards Alliance, BusinessWire.* URL: https://www.businesswire.com/news/home/20210511005933/en/The-Zigbee-Alliance-Rebrands-as-Connectivity-Standards-Alliance.

[84] *Using the HomeKit Accessory Protocol Specification, Apple developer documentation.* URL: https://developer.apple.com/homekit/faq/.

[85] *Vimar - Energia Positiva, Vimar website.* URL: https://www.vimar.com/en/int.

[86] *What Are Microservices? How Microservices Architecture Works, Middleware.io.* URL: https://middleware.io/blog/microservices-architecture/.

[87] *What is a Middleware, RedHat.* URL: https://www.redhat.com/en/topics/middleware/what-is-middleware.

[88] *What is a REST API, RedHat.* URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api.

[89] *What is a RESTful API, AWS.* URL: https://aws.amazon.com/what-is/restful-api/.

[90] *What is an Alexa-Enabled Smart Home, Amazon Alexa developers.* URL: https://developer.amazon.com/en-US/docs/alexa/smarthome/what-is-smart-home.html.

[91] *What is Matter, Google Home developers.* URL: https://developers.home.google.com/matter/primer.

[92] *What is Thread, Thread Group.* URL: https://www.threadgroup.org/What-is-Thread/Thread-Benefits.

[93] *Why Is Python Programming Considered the Top Language, Bairesdev.* URL: https://www.bairesdev.com/technologies/why-is-python-top-language/.

[94] *Why Samsung's SmartThings arm is pivoting entirely into software, Business Insider.* URL: https://web.archive.org/web/20220213144551/https://www.businessinsider.com/exclusive-why-samsungs-smartthings-is-going-all-in-on-software-2020-6.

[95] *World's most valuable technology companies in 2022, Bankrate.* URL: https://www.bankrate.com/investing/most-valuable-tech-companies/.

[96] *Zigbee overview, ST.com.* URL: https://wiki.st.com/stm32mcu/wiki/Connectivity:Zigbee_overview.