4 Reference architecture

In this chapter we present our IoT *Reference Architecture* (RA). This IoT RA is, among others, designed as a reference for the generation of compliant IoT concrete architectures that are tailored to one's specific needs. For other usages of the IoT Architectural Reference Model see Section 2.1.

The IoT Reference Architecture is kept rather abstract in order to enable many, potentially different, IoT architectures. Guidance on how to use all the parts of the IoT Reference Architecture can be found in Chapter 5.

Both in devising this chapter and in presenting the outcomes of our deliberations, we are adhering to the framework of architectural views and perspectives, as described in the software engineering literature and standards (for more details see [Rozanski 2011]). The use of well-known concepts makes it easier for architects from other domains to feel comfortable in the IoT world and this framework was thus a rather natural choice. To be more precise, we used the definitions of views from [Woods 2011], as well as their architectural-perspective catalogue. We adopted both according to IoT-specific needs. One has to be careful though, about the definition of views and viewpoints as these differ between authors. Nonetheless, there are no conceptual differences to traditional approaches and someone with a background in designing any kind of system should not have a steep learning curve. Notice though that architectural views and perspectives were originally defined for concrete architectures and not for reference architectures. Views that are very use-case dependent, for instance the IoT Physical-Entity view and the context view, are therefore not covered here. For a more detailed discussion of this aspect see Section 5.2. Furthermore, since a reference architecture covers a wide range of use cases, it is of course void of use-case-specific details (for instance usage patterns and the related interactions of the system's functional components), such aspects are not covered in the IoT Reference Architecture but have to be attended during, for instance, the architecture-generation process.

The structure of the chapter is as follows: First, we give a short overview on architectural views and perspectives. We then go on with presenting views that constitute the IoT Reference Architecture. The functional view and its viewpoints are described in great detail. At the time of writing there was indeed so much information at hand that we decided to only present an overview of the functional view here and to cover, for instance, the detailed definitions of the functional components of the functional-decomposition viewpoint in Appendix C. Next, the information view is introduced as well as the deployment and operational view. The remainder of the chapter is then devoted to architectural

perspectives. We describe four architectural perspectives (evolution and interoperability; performance and scalability; trust, security, and privacy; and availability and resilience). How architectural perspectives influence the architecting process is not covered here but in the Guidelines Chapter (see Section 5.2.10).

4.1 Short definition of Architectural Views and Perspectives

A system architecture, and thus by default, a reference architecture, needs to answer a wide range of questions. Such questions can, for instance, address:

- Functional elements;
- Interactions of said elements;
- Information management;
- Operational features;
- Deployment of the system;

What the user of an architecture expects, is an architectural description, viz. "a set of artifacts that documents an architecture in a way its stakeholders can understand and that demonstrates that the architecture has met their concerns" [Rozanski 2005]. Instead of providing these artifacts in a monolithic description, one often chooses to delineate them by so-called architectural views. The idea behind so doing is to focus on system aspects that can be conceptionally isolated. Architectural views make both the derivation of the architecture and its validation easier. The above bullet-point list provides examples of such views. A more detailed discussion of views and how we adapted them to the reference-architecture realm is provided in the next section.

In the past it has been found that views are unfortunately not enough for describing system architectures and that many stakeholder aspirations are rather of a qualitative nature [Rozanski 2011]. Such qualitative aspirations cut across more than one view. Such aspirations are referred to architectural perspectives, of which privacy is but one example. A more detailed discussion of architectural perspectives is provided in Section 4.3.

The joint use of architectural views and perspectives in architecture descriptions is described in more detail in the pertinent literature [Rozanski 2011].

4.2 Architectural Views

Views are used during the design and implementation phase of a concrete system architecture. They are defined in the following way:

"A view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders." [Rozanski 2011]

A view is composed of viewpoints, which aggregate several architectural concepts in order to make the work with views easier. The IEEE standard 1471 defines viewpoints as follows:

"A viewpoint is a collection of patterns, templates, and conventions for constructing one type of view. It defines the stakeholders whose concerns are reflected in the viewpoint and the guidelines, principles, and template models for constructing its views." [IEEE Architecture]

Some typical examples for viewpoints are

- **Functional view**: functional-decomposition viewpoint; interaction viewpoint; interface viewpoint;
- **Information view**: information-hierarchy viewpoint; semantics viewpoint; information-processing viewpoint; information-flow viewpoint.

4.2.1 Usage of Views and Perspectives in the IoT RA

As mentioned in the introduction to this chapter, the IoT Reference Architecture is use-case- and application- independent and is therefore not compatible to the concept of views and viewpoints one-by-one. But the idea behind the concept is nevertheless helpful and was thus adopted for the use within the IoT Reference Architecture. As discussed above the following views were left out from the IoT Reference Architecture but are discussed in Section 5.2:

- Physical-Entity View;
- Context View.

Concerning the Functional View, of the above three viewpoints, interactions are not covered in the IoT Reference Architecture, since the number of arrangements of the Functional Components and also their invocation is practically infinite. Instead, we chose to cover some typical –but yet high-level- interaction patterns in the Guidelines chapter (see Section 5.2.10).

The same is true for the deployment and operational View. However, there are aspects to both that are practically invariant over the IoT domain and these aspects are covered in Section 4.2.4. Also, what is an aspect of the deployment view in one architecture can be an aspect of the operation view in another architecture. Situating these aspects in either or is contingent on, among others

- **Requirements** (usability; institutional rules and traditions; etc.) and

- **Design choices** made (commission on manufacturing floor; shipment and installation by experts; operation by experts).

The following sections present the IoT Functional View, IoT Information View, and the IoT Deployment and Operational View of the IoT RA.

4.2.2 Functional view

4.2.2.1 Devising the Functional View

The Functional View builds further on the Functional Model as can be seen in Figure 31 below.

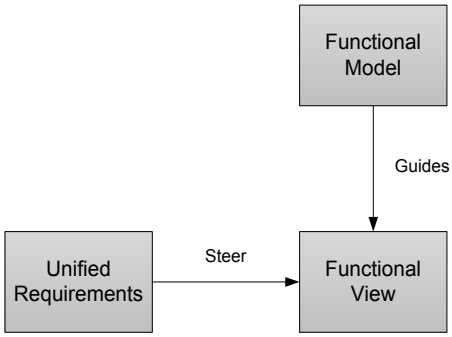

Figure 31: Functional view process.

In a first step, the Unified Requirements are mapped to the different Functionality Groups of the IoT Functional Model.

Next, clusters of requirements of similar functionality are formed and a Functional Component for these requirements defined.

Finally, the Functional Components are refined after discussion with the technical work packages.

The viewpoints used for constructing the IoT Functional View are hence:

1) The Unified Requirements;
2) The IoT Functional Model;

Once all Functional Components are defined, the default function set, system use cases, sequence charts and interface definitions are made, which can all be found back in Appendix C.

The Functional View diagram is depicted in Figure 32 and shows the nine FGs of the Functional Model. Note that:

- The Application FG and Device FG are out-of-scope of the IoT-A Reference Architecture and are coloured in yellow;
- Management FG and Security FG are transversal FGs and are coloured dark blue.

For each of the FGs, the *Functional Components* (FC) are depicted.

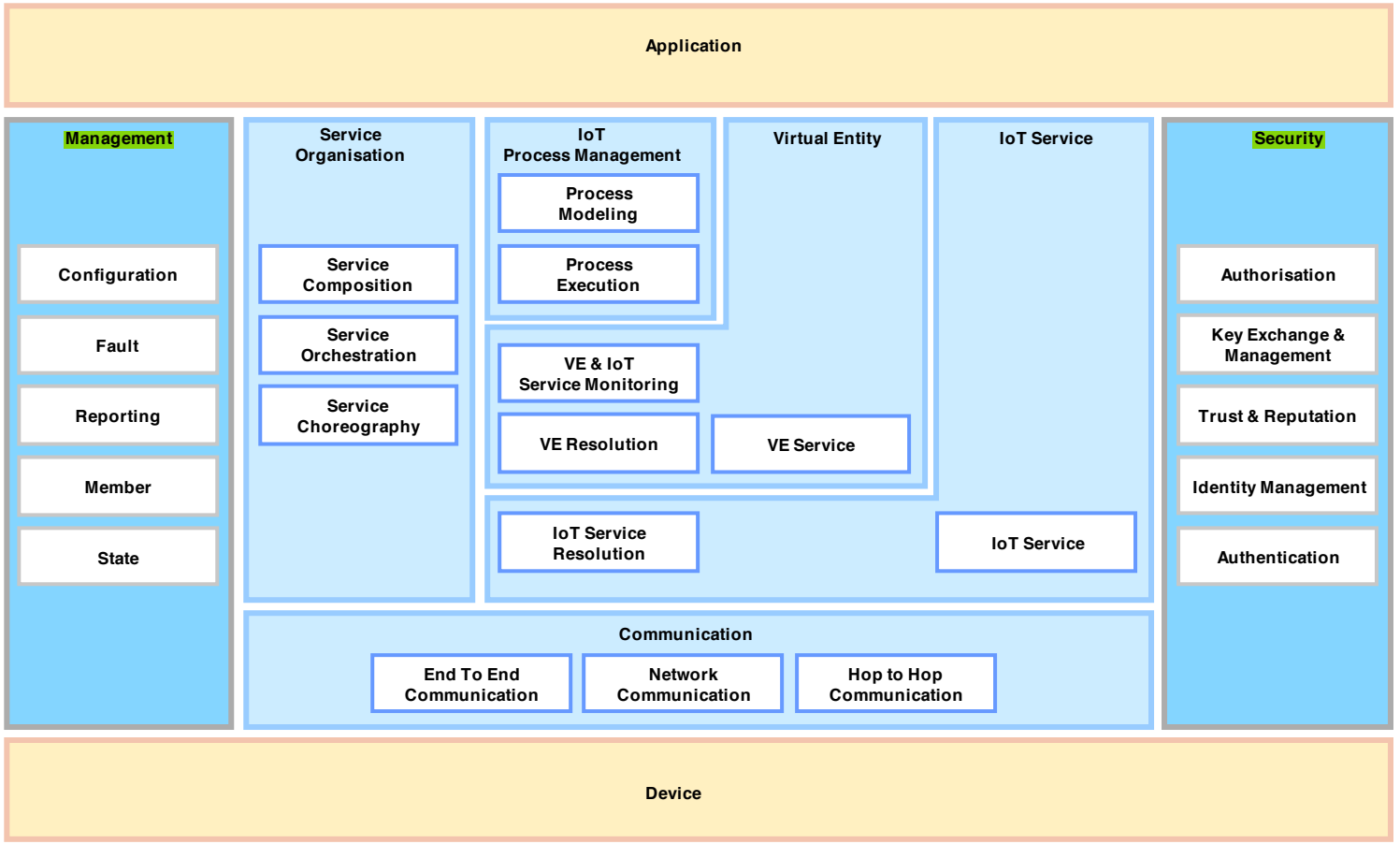

Figure 32: Functional-decomposition viewpoint of the IoT Reference Architecture. .

In the following sub-sections, the FCs of each FG will be described in more detail.

The Functional View presented in this chapter will give a description of the Functional Components, but will not describe the interactions taking place between the Functional Components.

The reason is that these interactions are typically depending on Design Choices which are not made at this level of abstraction.

Section 5.5 of the “Guidance” chapter will go more in detail and depict some typical interaction scenarios.

In addition to the description in this chapter, more detailed information such as requirement mapping, system use cases, interaction diagrams and interface definitions can be found in Appendix C.

4.2.2.2 IoT Process Management

The IoT Process Management FG relates to the integration of traditional process management systems with the IoT ARM. The overall aim of the FG is to provide the functional concepts and interfaces necessary to augment traditional (business) processes with the idiosyncrasies of the IoT world.

The IoT Process Management FG consists of two Functional Components (see Figure 33 below):

- Process Modelling;
- Process Execution.

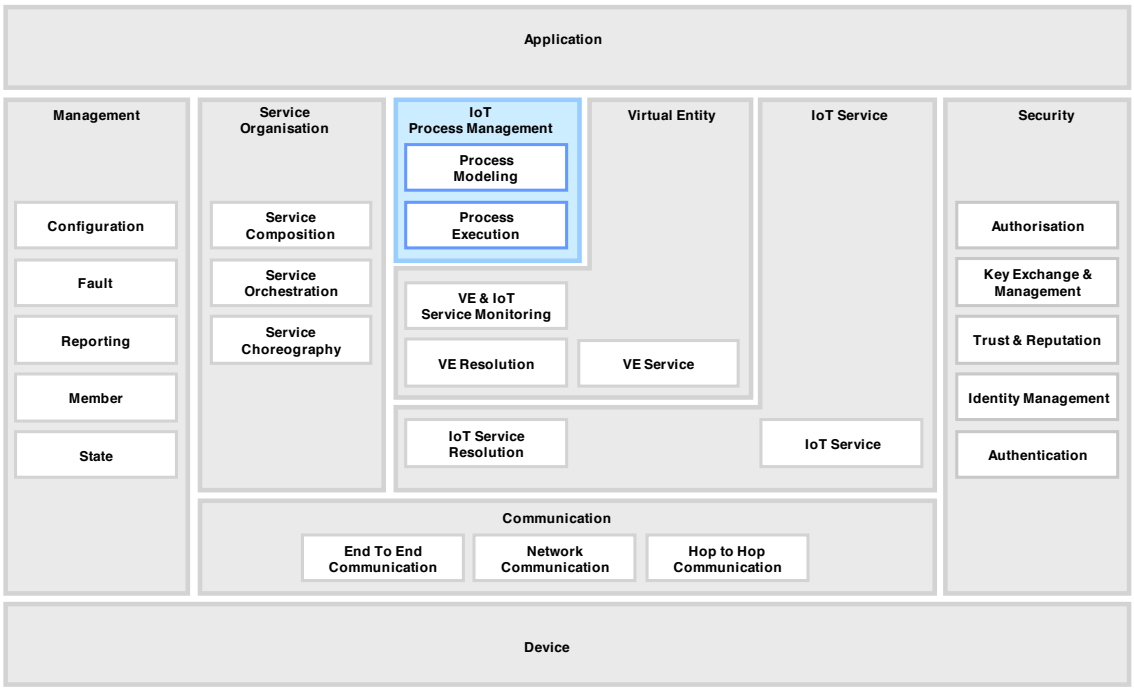

Figure 33: IoT Process Management FG

The **Process Modelling FC** provides an environment for the modelling of IoT-aware business processes that will be serialised and executed in the Process Execution FC.

The main function of the Process Modelling FC is to provide the tools necessary for modelling processes using the standardised notation,[4] i.e. using novel modelling concepts specifically addressing the idiosyncrasies of the IoT ecosystem.

The **Process Execution FC** executes IoT-aware processes that have been modelled in the Process Modelling FC described above. This execution is achieved by utilising IoT Services that are orchestrated in the Service Organisation layer.

The Process Execution FC is responsible for deploying process models to the execution environments: activities of IoT-aware process models are applied to appropriate execution environments, which perform the actual process execution by finding and invoking appropriate IoT Services.

The Process Execution FC also aligns application requirements with service capabilities. For the execution of applications, IoT Service requirements must

[4] Such a notation is currently been developed as part of the IoT-A project.

be resolved before specific IoT Services can be invoked. For this step, the Process Execution FC utilises components of the Service Organization FG.

Finally, the Process Execution FC can run applications: after resolving IoT Services, the respective services are invoked. The invocation of a service leads to a progressive step forward in the process execution. Thus, the next adequate process based on the outcome of a service invocation will be executed.

4.2.2.3 Service Organisation

The Service Organisation FG (see Figure 34) is the central Functional Group that acts as a communication hub between several other Functional Groups. Since the primary concept of communication within the IoT ARM is the notion of a Service, the Service Organisation is used for composing and orchestrating Services of different levels of abstraction.

The Service Organisation FG consists of three Functional Components:

- Service Orchestration;
- Service Composition;
- Service Choreography.

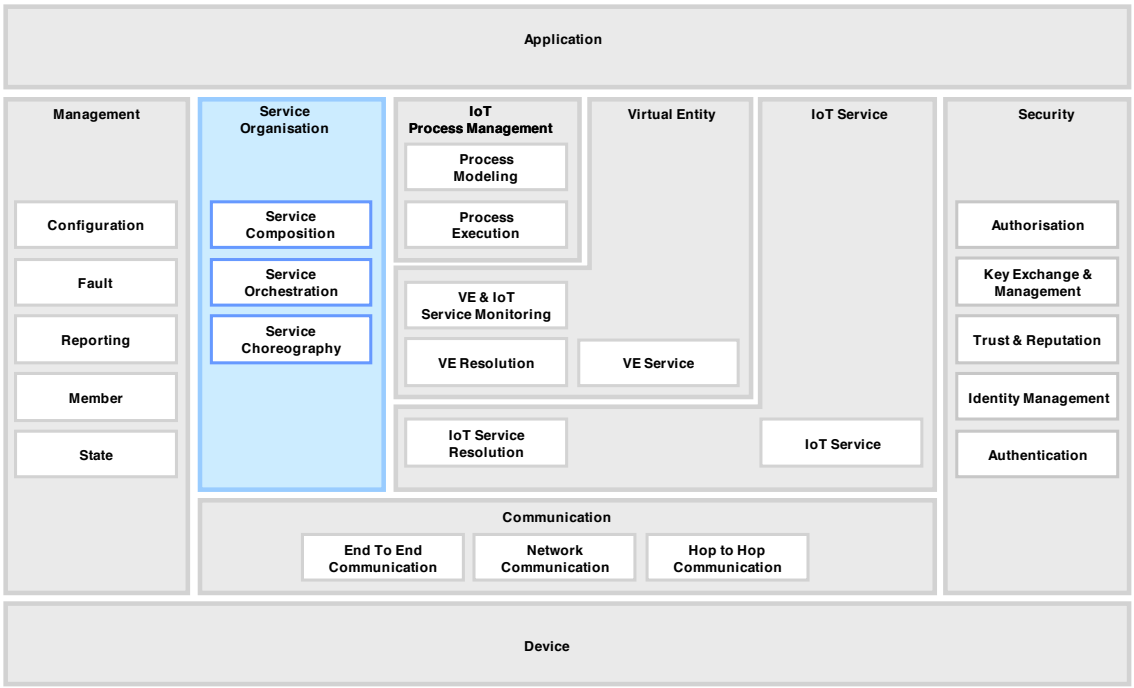

Figure 34: Service Organisation FG

The **Service Orchestration FC** resolves the IoT Services that are suitable to fulfil service requests coming from the Process Execution FC or from Users.

Its only function is to orchestrate IoT Services: resolve the appropriate services that are capable of handling the IoT User's request. If needed, temporary

resources will be set up to store intermediate results that feed into Service Composition or complex event processing.

The **Service Composition FC** resolves services that are composed of IoT Services and other services in order to create services with extended functionality. The Functional Component has two main functions: (1) support flexible service compositions and (2) increase quality of information.

To support flexible service compositions, the Service Composition FC must provide dynamic resolution of complex services, composed of other services. These combinable services are chosen based on their availability and the access rights of the requesting user.

Quality of information can be increased by combining information from several sources. For example, an average value –with an intrinsically lower uncertainty- can be calculated based on the information accessed through several resources.

The **Service Choreography FC** offers a broker that handles Publish/Subscribe communication between services. One service can offer it's capabilities at the FC and the broker function makes sure a client interested in the offer will find the service with the desired capabilities.

Also service consumers can put service requests onto the Choreography FC while a suitable service is not available at the time when the request was issued. The service consumer will get notified as soon as a service became available that fulfils the service request issued before.

4.2.2.4 Virtual Entity

The Virtual Entity FG (see Figure 35) contains functions for interacting with the IoT System on the basis of VEs, as well as functionalities for discovering and looking up services that can provide information about VEs, or which allow the interaction with VEs. Furthermore, it contains all the functionality needed for managing associations, as well as dynamically finding new associations and monitoring their validity.

The Virtual Entity FG consists of three Functional Components:

- VE Resolution;
- VE & IoT Service Monitoring;
- VE Service.

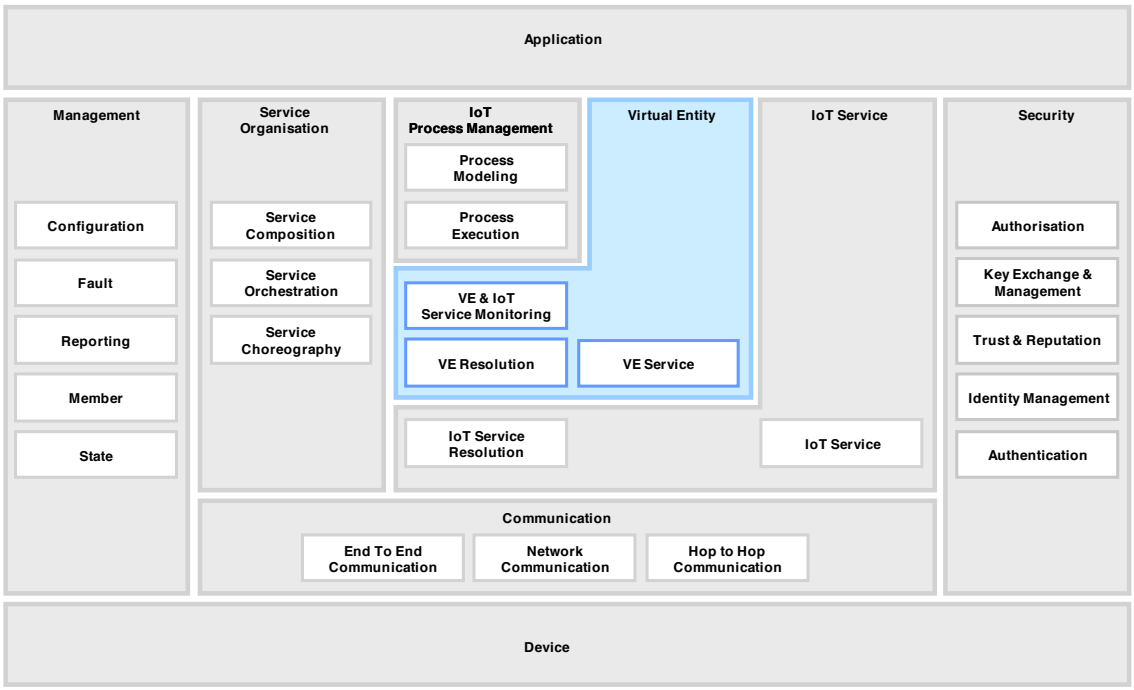

Figure 35: Virtual Entity FG

The **VE Resolution FC** is the Functional Component that provides the functionalities to the IoT User to retrieve associations between VE's and IoT Services.

This includes the discovery of new and mostly dynamic associations between VE and associated services. For the discovery qualifiers, location, proximity, and other context information can be considered. If no association exists, the association can be created.

The User can also subscribe or unsubscribe to continuous notifications about association discovery that fit a provided specification of the VE or of the Service. In case of a notification, a callback function will be called.

Similar, the User can subscribe or unsubscribe to notifications about association lookup.

The VE Resolution FC also allows to lookup VE-related services, i.e. search for services exposing resources related to a VE.

Finally, the VE Resolution FC allows managing associations: insert, delete and update associations between a VE and the IoT Services that are associated to the VE.

The **VE & IoT Service Monitoring FC** is responsible for automatically finding new associations, which are then inserted into the VE Resolution FC. New associations can be derived based on existing associations, Service Descriptions and information about VE's.

The functions of the VE & IoT Service Monitoring FC are to assert static associations, i.e. create a new static association between VE's and services described by the provided association, discover dynamic associations, i.e. create a new dynamic or monitored association between VE's and Services, update the association and delete the association from the VE Resolution framework.

Finally, the **VE Service FC** handles with entity services. An entity service represents an overall access point to a particular entity, offering means to learn and manipulate the status of the entity. Entity services provide access to an entity via operations that enable reading and/or updating the value(s) of the entities' attributes. The type of access to a particular attribute depends on the specifics of that attribute (read only / write only or both).

A specific VE service can provide VE history storage functionality, to publish integrated context information (VE context information - dynamic and static), VE state information, VE capabilities.

The two functions currently defined for the VE Service FC are to read and set an attribute value for the entity.

It is not required to have an explicit register for Virtual Entities, but the VE Resolution FC could be extended in order to be used in that way. The important aspect is to agree on how to assign identifiers to Virtual Entities.For modelling any other aspect of the Virtual Entity, a Virtual Entity service can be used that gives you access to all information about a Virtual Entity. This can be current sensor information, as well as historic information. Historic information would typically be stored in a database, which can be modelled as a Network Resource (see Section 3.3.3).

4.2.2.5 IoT Service

The IoT Service FG (see Figure 36) contains IoT services as well as functionalities for discovery, look-up, and name resolution of IoT Services. It consists of two Functional Components:

- IoT Service ;
- IoT Service Resolution.

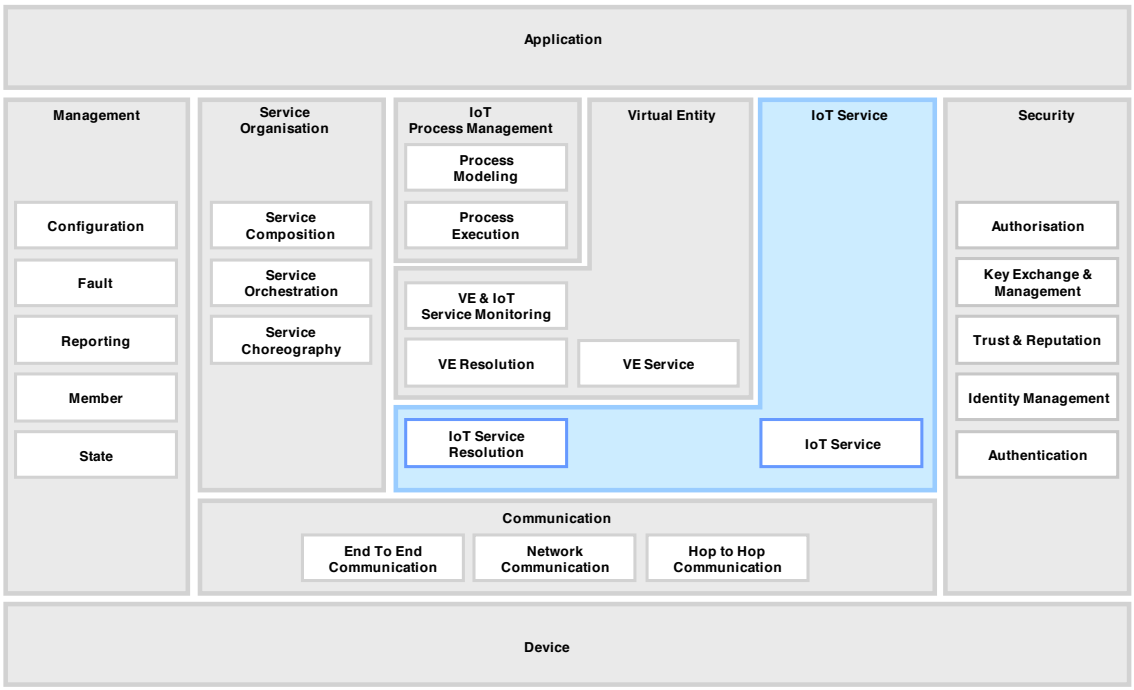

Figure 36: IoT Service FG

An IoT Service exposes one Resource to make it accessible to other parts of the IoT system. Typically, IoT Services can be used to get information provided by a resource retrieved from a sensor device or from a storage resource connected through a network. An IoT Service can also be used to deliver information to a resource in order to control actuator devices or to configure a resource. Resources can be configurable in non-functional aspects, such as dependability security (e.g. access control), resilience (e.g. availability) and performance (e.g. scalability, timeliness).
IoT Services can be invoked either in a synchronous way by responding to service requests or in an asynchronous way by sending notifications according to subscribtions previously made through the service.

A particular type of IoT Service can be the Resource history storage that provides storage capabilities for the measurements generated by resources.

The main functions of the **IoT Service FC** are to (1) return information provided by a resource in a synchronous way, (2) accept information sent to a resource in order to store the information or to configure the resource or to control an actuator device and (3) subscribe to information, i.e. return information provided by a resource in an asynchronous way.

The **IoT Service Resolution FC** provides all the functionalities needed by the user in order to find and be able to contact IoT Services. The IoT Service Resolution also gives services the capability to manage their service descriptions (typically stored in a database as one entry), so they can be looked up and discovered by the user. The user can be either a Human User or a software component.

Service Descriptions are identified by a service identifier and contain a service locator that enables accessing the service. Typically they contain further information like the service output, the type of service or the geographic area for which the service is provided. The exact contents, structure and representation depend on design choices taken, which is left open at the Reference Architecture level. Some examples for service models (structure) and representations of a service description can be found in [Mar t í n 2012] .

The functionalities offered by the IoT Service Resolution FC in brief are:

- **Discovery functionality** finds the IoT Service without any prior knowledge such as a service identifier. The functionality is used by providing a service specification as part of a query. What can be queried based on a service specification depends on what is included in the service description. As described above, this may include the service output, the service type and the geographic area for which the service is provided. The representation of the service specification will also be linked to the service description, e.g. if the service description is represented in RDF, a service specification based on SPARQL would be appropriate;
- **Lookup** is a functionality which enables the User to access the service description having prior knowledge regarding the service identifier;
- **Resolution** function resolves the service identifiers to locators through which the User can contact the Service. A service locators are typically also included in the service description, the resolution function can be seen as a convenience function that reduces the amount of information that has to be communicated, especially if the service description is large and the contained information is not needed;
- **Other functionalities** provided by the IoT Service Resolution FC are the management of the service descriptions. IoT Services can update, insert or simply delete the service descriptions from the IoT Service Resolution FC. It is also possible that these functions are called by the functional components of the Management FG and not by the IoT Services themselves.

4.2.2.6 Communication

The Communication FG (see Figure 37 below) is an abstraction, modelling the variety of interaction schemes derived from the many technologies belonging to IoT systems and providing a common interface to the IoT Service FG.

The Communication FG consists of three functional components:

- Hop To Hop Communication;
- Network Communication;
- End To End Communication.

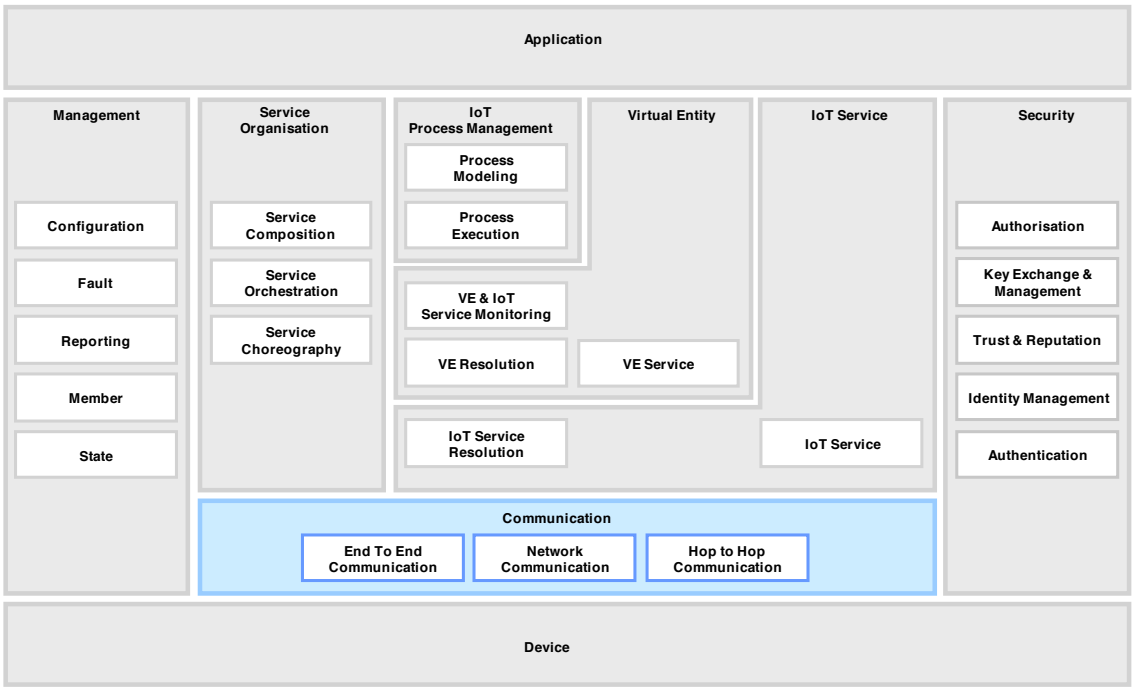

Figure 37: Communication FG

The **Hop To Hop Communication FC** provides the first layer of abstraction from the device's physical communication technology. The functional component is an abstraction to enable the usage and the configuration of any different link layer technology.

Its main functions are to transmit a frame from the Network Communication FC to the Hop To Hop Communication FC and from a Device to the Hop To Hop Communication FC. The arguments for the frame transmission can be set and example of arguments include: reliability, integrity, encryption and access control.

The Hop To Hop Communication FC is also responsible for routing a frame. This function allows routing a packet inside a mesh network such as for instance 802.15.4 (mesh-under routing). Note that this function is not mandatory for all implementations of the Hop To Hop Communication FC. It is required only for meshed link layer technologies.

Finally, the Hop To Hop Communication FC allows to manage the frame queue and set the size and priorities of the input and output frame queues. This function can be leveraged in order to achieve QoS.

The **Network Communication FC** takes care of enabling communication between networks through Locators (addressing) and ID Resolution. The FC includes routing, which enables linking different network address spaces. Moreover different network technologies can be converged through network protocol translations.

The functions of the Network Communication FC are to transmit a packet from the Hop To Hop Communication FC to the Network Communication FC and from the End To End Communication FC to the Network Communication FC. The arguments for the packet transmission can be configured and examples of arguments include: reliability, integrity, encryption, unicast/multicast addressing and access control.

The Network Communication FC enables as well network protocol translation where it allows translating between different network protocols. Examples would be to translate IPv4 to IPv6 and ID to IPv4. Note that this function is necessary to implement a Gateway.

In case a packet needs to be routed, the Network Communication FC allows finding the next hop in a network. It also allows dealing with multiple network interfaces. The function is not mandatory for all implementations of the Network Communication FC. It is required only on devices with multiple network interfaces.

Another function of the Network Communication FC is to resolve the locator-to-ID where it allows getting a locator from a given ID. The resolution can be internal based on a lookup table or external via a resolution framework.

Finally, the Network Communication FC can manage the packet queue and setup the size and priorities of the input and output packet queues. This function can be leveraged in order to achieve QoS.

The **End To End Communication FC** takes care of the whole end-to-end communication abstraction, meaning that it takes care of reliable transfer, transport and, translation functionalities, proxies/gateways support and of tuning configuration parameters when the communication crosses different networking environments.

The End To End Communication FC is responsible to transmit a message from the Network Communication FC to the End To End Communication FC and from (IoT) Service to the End To End Communication FC. The arguments for the message can be configured and examples include: reliability, integrity, encryption, access control and multiplexing.

A second function of the End To End Communication FC is to cache and proxy. The Cache and Proxy function allows to buffer messages in the End To End Communication FC.

Another function of the FC is to translate end-to-end protocol. The Translate End To End Protocol function allows to translate between different End To End Protocols. An example would be to translate HTTP/TCP to COAP/UDP. Note that this function is necessary to implement a Gateway.

A last function of the FC is to pass the context of protocol translation between gateways. The context could be related to addressing, methods specific for a RESTful protocol, keying material and security credentials.

4.2.2.7 Security

The Security FG (see Figure 38) is responsible for ensuring the security and privacy of IoT-A-compliant systems.

It consists of five functional components:

- Authorisation;
- Key Exchange & Management;
- Trust & Reputation;
- Identity Management;
- Authentication.

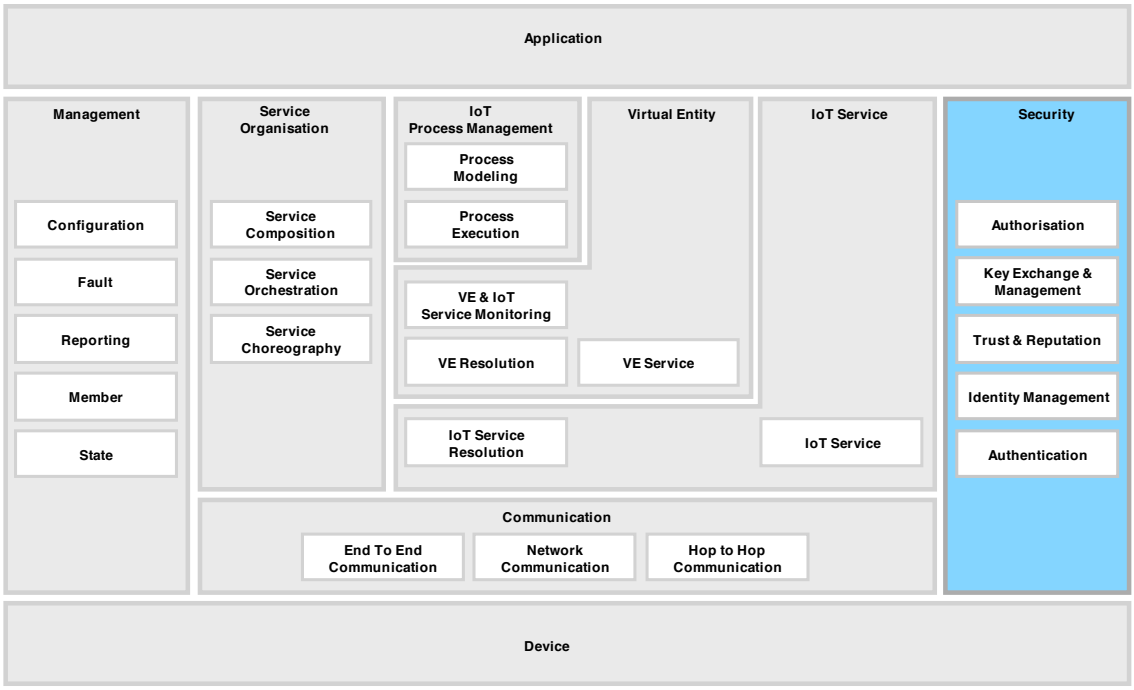

Figure 38: Security FG

The **Authorization FC** is a front end for managing policies and performing access control decisions based on access control policies. This access control decision can be called whenever access to a restricted resource is requested. For example, this function is called inside the IoT Service Resolution FC, to check if a user is allowed to perform a lookup on the requested resource. This is an important part of the privacy protection mechanisms

The two default functionalities offered by the Authorization FC are 1/ to determine whether an action is authorized or not -the decision is made based on the information provided from the assertion, service description and action type- and 2/ to manage policies. This refers to adding, updating or deleting an access policy.

The **Authentication FC** is involved in user and service authentication. It checks the credentials provided by a user, and, if valid, it returns an assertion as result, which is required to use the IoT Service Client. Upon checking the correctness of the credentials supplied by a newly joining node, it establishes secured contexts between this node and various entities in its local environment.

The two functionalities provided by the Authentication FC are 1/ to authenticate a user based on provided credential and 2/ to verify whether an assertion provided by a user is valid or invalid.

The **Identity Management FC** addresses privacy questions by issuing and managing pseudonyms and accessory information to trusted subjects so that they can operate (use or provide services) anonymously.

Only one default function is attributed to this FC: to create a fictional identity (root identity, secondary identity, pseudonym or group identity) along with the related security credentials for users and services to use during the authentication process.

The ***Key Exchange and Management*** (KEM) **FC** is involved to enable secure communications between two or more IoT-A peers that do not have initial knowledge of each other or whose interoperability is not guaranteed, ensuring integrity and confidentiality.

Two functions are attributed to this FC:

- **Distribute keys** in a secure way: upon request, this function finds out a common security framework supported by the issuing node and a remote target, creates a key (or key pair) in this framework and then distributes it (them) securely. Security parameters, including the type of secure communications enablement, are provided;

- **Register security capabilities**: nodes and gateways that want to benefit from the mediation of the KEM in the process of establishing secure connections can make use of the register security capabilities function. In this way the KEM registers their capabilities and then can provide keys in the right framework.

The **Trust and Reputation Architecture FC** collects user reputation scores and calculates service trust levels.

Again, two default functions are attributed to the FC:

- **Request reputation information**: this function is invocated at a given remote entity to request reputation information about another entity. As input parameters, a unique identifier for the remote entity (subject), as well as the concrete context (what kind of service) is given. As a result a reputation bundle is provided;

- **Provide reputation information**: this function is invocated at a given remote entity to provide reputation information (recommendations or

feedback) about another entity. As input parameters, a unique identifier for the entity to be assessed (subject), as well as the concrete context, the given score and a timestamp are given. As a result, the corresponding reputation is provided.

4.2.2.8 Management

Section 3.5.2.5 provides a high-level discussion for the role and the goals of the Management FG, but it does not specify how to functionally parse this group. For guidance on this question we turned to FCAPS, which offers a comprehensive high-level framework for network management [Fleischmann 2005]. It was, among others, incorporated into an ITU-T recommendation [ITU-T 1997] and it has already been considered for Smart-Grid applications, which are just one example for IoT [Greenfield 2009]. The letters F C A P S stand for the functionalities

- Fault;
- Configuration;
- Accounting (Administration);
- Performance;
- Security.

Of these functionalities, Fault, Configuration, and Performance cover all the important goals of the Management FG. In this document we choose to make Security a separate functionality group in order to emphasise its importance for IoT. FCAPS was designed with telecommunication applications in mind, while subscriber-based services will be just one of many business models for the IoT. Therefore accounting functionalities will be covered by primary services. However, for administration purposes we introduce the functional components State FC and Member FC. Performance functionality is related to the monitoring of the state of the system and to the adaptation of its configuration, and is therefore incarnated into the Fault, State and Configuration Functional Components. (see Figure 39) illustrates how the high-level goals motivating the creation of a Management FG (see Section 3.5.2.5) map onto the chosen functional components.

	Management FCs				
High-level goals	**Fault**	**Configuration**	**Reporting**	**Member**	**State**
Cost reduction	X	X	X	X	
Attending unforeseeable usage issues	X	X	X	X	X
Fault handling	X	X	X	X	X

Flexibility		X	X	X	X

Table 4: Mapping of the high-level roles of the Management FG onto FCs.

IoT systems differ from pure networking solutions in that they also offer low-level services and support for business administration. An IoT system is thus much more complex than a communication system, and we chose to make the management of FG-specific FCs part of that very FG, while the Management FG is responsible for cross-functionality-group task. In other words, it is responsible for the composition and tracking of actions that involve several of the "core FGs" (i.e. not including Device and Application FG). The requirement grounding for the Management FG is based on the extrapolation of a number of communications requirements to system-wide management and behaviours (these requirements can be found in the description of the individual functional components). In addition, if the interaction of the Application and/or Device FG necessitates the composition and tracking of at least two core FGs, such actions are also candidates for the sphere of responsibility of the Management FG.

By exclusion, the following management activities are thus out of the scope of the Management FG:

1. Activities that only pertain to a single functionality group: an example for this is the management of authorisations in the Security FG;
2. The management of interactions between functionality groups that do not require "external" intervention. An example for the latter are requests between two FGs that can be managed by the requesting FG itself.

The Management FG (see Figure 39) consists of five Functional Components:

- Configuration;
- Fault;
- Reporting;
- Member;
- State.

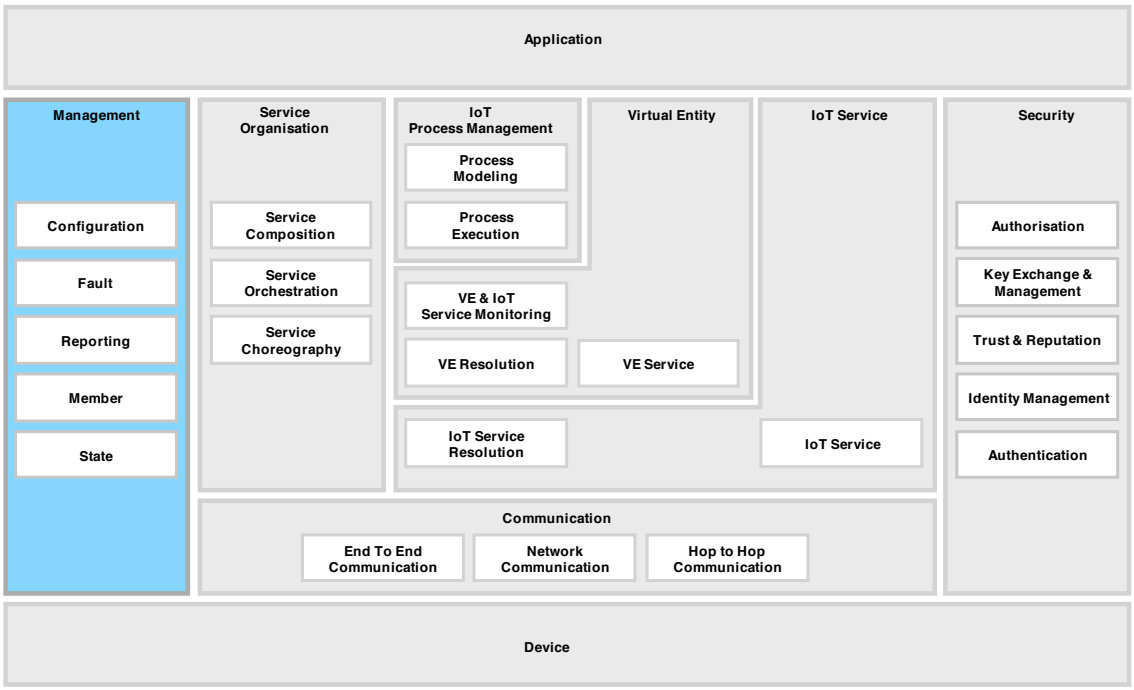

Figure 39: Management

The **Configuration FC** is responsible for initialising the system configuration such as gathering and storing configuration from FC's and Devices. It is also responsible for tracking configuration changes and planning for future extension of the system.

As such, the main functions of the Configuration FC are to retrieve a configuration and to set the configuration:

- The **retrieve configuration** function allows to retrieve the configuration of a system, either from history (latest known configuration) or from the system (current configuration, including retrieval of the configuration of one or a group of Devices), enabling tracking of configuration changes. The function can also generate a configuration log including descriptions of Devices and FCs. A filter can be applied to the query;

- The **set configuration** function is mainly used to initialise or change the system configuration.

The goal of the **Fault FC** is to identify, isolate, correct and log faults that occur in the IoT system. When a fault occurs, the respective functional component notifies the Fault FC. Such notification triggers, for instance, are the gathering of more data in order to identify the nature and severity of the problem. Another action can encompass bringing backup equipment on-line.

Fault logs are one input used for compiling error statistics. Such statistics can be used for identifying fragile functional components and/or devices. Also,

“performance thresholds can be set in order to trigger an alarm.” [Wikipedia 2012] . Performance data is provided by the State FC.

The Fault FC contains functions to handle a fault, to monitor a fault and to retrieve a fault.

The role of the function that handles a fault is to react to fault detection by generating alarms, logging faults, or applying corrective behaviours. Generated alarms can be disseminated to other FCs. This function can also analyse faults and, if requested, start an action sequence that tackles the fault, possibly interfacing with the changeState() function of the State FC. This usually includes command messages sent to other FCs. This function can also set the system back to a previous state by calling the setConfiguration() function in the Configuration FC. One of the actions this might entail is setting back the system to a previous configuration.

Faults can also be monitored by the Fault FC. This function is mainly used in subscription mode where it monitors the errors of the system and notifies subscribers of matchingevents.

Finally, the Fault FC provides access to the Fault History. For this access, a filter function can be applied.

The **Member FC** is responsible for the management of the membership and associated information of any relevant entity (FG, FC, VE, IoT Service, Device, Application, User) to an IoT system.

It is typically articulated around a database storing information about entities belonging to the system, including their ownership, capabilities, rules, and rights.

This FC works in tight cooperation with FCs of the Security FG, namely the Authorisation and Identity Management FCs.

The Member FC has three default functions: 1/ the continous monitoring of members, 2/ the retrieve member function which allows retrieving members of the system complying with a given filter and also allows to subscribe to updates of the membership table fitting a specified filter (e.g. to be notified of all updates to entities belonging to a given owner) and finally 3/ the update member function which allows to update member metadata in the membership database and to register or unregister member metadata in the membership database.

The **Reporting FC** can be seen as an overlay for the other Management FCs. It distils information provided by them. One of many conceivable reporting goals is to determine the efficiency of the current system. This is important since by “collecting and analysing performance data, the [system] health can be monitored” [Wikipedia 2012] . Establishing trends enables the prediction of future issues. This FC can also be utilised for billing tasks.

There is only one default function for the FC: retrieve a report. This function generates reports about the system. Can either return an existing report from the report history, or generate a new on through calls on the other Management FCs.

The **State FC** monitors and predicts state of the IoT system. For a ready diagnostic of the system, as required by Fault FC, the past, current and predicted (future) state of the system are provided. This functionality can also support billing. The rationale is that Functions/Services such as Reporting need to know the current and future state of the system. For a ready diagnostic of the system one also needs to know its current performance.

This FC also encompasses a behaviour functionality, which forces the system into a particular state or series of states. An example for an action for which such functionality is needed is an emergency override and the related kill of run-time processes throughout the system. Since such functionality easily can disrupt the system in an unforeseen manner this FC also offers a consistency checks of the commands issued by the changeSt at e functionality in the State FC.

The functions of the State FC are to change or enforce a particular state on the system. This function generates sequence of commands to be sent to other FCs. This function also offers the opportunity to check the consistency of the commands provided to this function, as well as to check predictable outcomes (through the pr edi ct St at e function).

A second function is to monitor the state. This function is mainly used in subscription mode, where it monitors the state of the system and notifies subscribers of relevant changes in state.

Other functions of the FC are to predict the state for a given time, to retrieve the state of the system through access to the state history and to update the state by changing or creating a state entry.

4.2.2.9 Mapping of Functional View to the Red Thread example

In this section, the “Red Thread” example will be mapped on the Functional View and the main Functional Components used for the example are highlighted as can be seen in Figure 40:

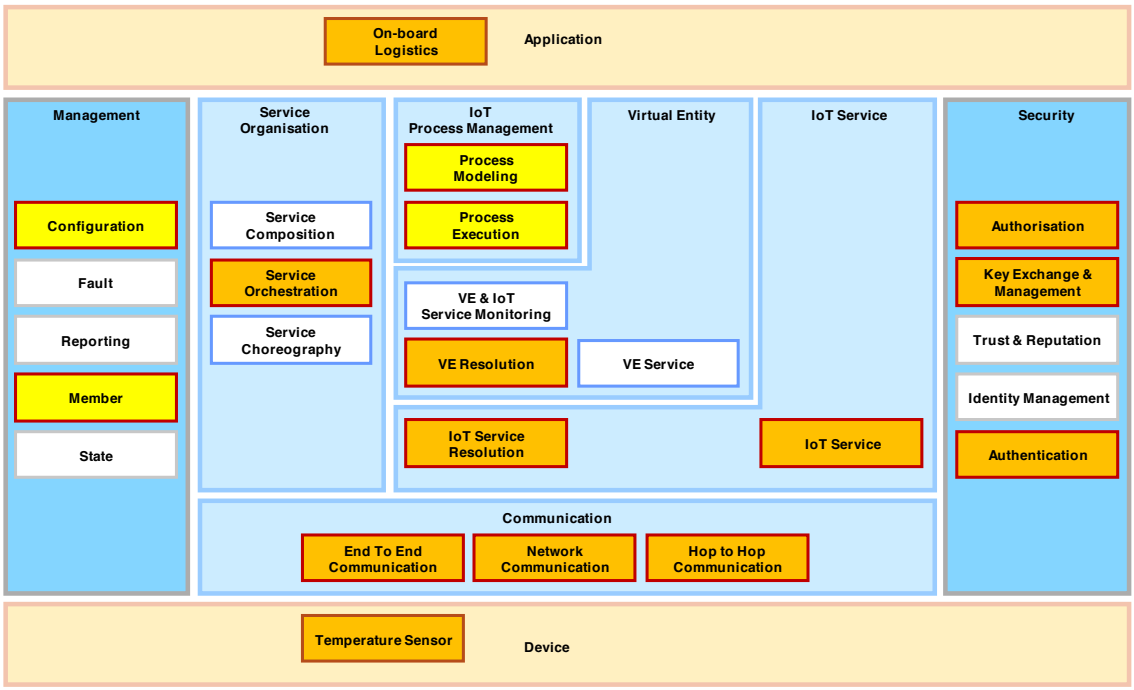

Figure 40: “Red Thread” example

In that figure, Functional Components which are used only once, such as during the instantation of the process model or configuration of devices are indicated in light yellow.

Functional Components which are used at runtime of the use case are indicated in orange.

The example of this section can be described only at a high level, since a concrete architecture and implementation are needed to go into further detail. Also the design choices of the concrete architecture need to be considered.

In this example, the embedded sensors (Temperature Sensor) continuously measure the environmental conditions within the truck. The measurement data is available to “Ted” ’s IoT-Phone (On-board Logistics Application) since the IoT-Phone is subscribed to the service exposing the measurement data (IoT Service). In order to subscribe to the data, the association between the service exposing the data and the Load carrier needs to be resolved (VE Resolution and IoT Service Resolution). The communication from sensor to IoT-Phone makes use of the network protocol stack of the IoT Communication Model (End To End Communication, Network Communication, Hop to Hop Communication, Key Exchange & Mangement). All transactions take place in a secure way, meaning that no operations are allowed unless authentication (Authentication) took place and explicit authorisation is obtained for the particular operation (Authorisation).

It is beyond the scope of this section but an illustration of the adaption of the ARM to a specific case and implementation can be found in [Meyer 2013] .

4.2.3 Information view

One of the main purposes of connected and smart objects in the IoT is the exchange of information between each other and also with external systems. Therefore the way to define, structure, store, process, manage and exchange information is very important. The information view helps to generate an overview about static information structure and dynamic information flow.

Based on the IoT Information Model, this view gives more details about how the relevant information is to be represented in an IoT system. As we are describing a reference architecture as opposed to a specific system architecture, concrete representation alternatives are not part of this view.

Going beyond the IoT Information Model, the information view also describes the components that handle the information, the flow of information through the system and the life cycle of information in the system.

The current version of the Information View focuses on the description, the handling and the life cycle of the information and the flow of information through the system and the components involved. Given the current level of detail, we will provide a viewpoint only for modelling the type system of Virtual Entities.

4.2.3.1 Information Description

Description of Virtual Entities

The Virtual Entity is the key concept of any IoT system as it models the Physical Entity or the Thing that is the real element of interest. As specified in the IoT IM, Virtual Entities have an identifier (ID), an entityType and a number of attributes that provide information about the entity or can be used for changing the state of the Virtual Entity, triggering an actuation on the modelled Physical Entity. The modelling of the entityType is of special importance. The entityType can be used to determine what attributes a Virtual Entity instance can have, defining its semantics. The entityType can be modelled based on a flat type system or as a type hierarchy, enabling sub-type matching. Figure 41 shows a flat entityType model for aspects of the red thread scenario with boxes and pallets as concrete load carriers. Figure 42 shows a hierarchical entityType model for the same scenario. Here more abstract entityTypes have been introduced like Human and LoadCarrier. The entityType Human has an attribute *name*, which is inherited by all sub-types, i.e. by Driver, Worker and Manager.

For modelling entityType hierarchies, ontologies or UML class diagrams can be used. Of course, this choice is related to the design choice on how the overall Virtual Entity information is represented.

Figure 41: Example for flat entityType model.

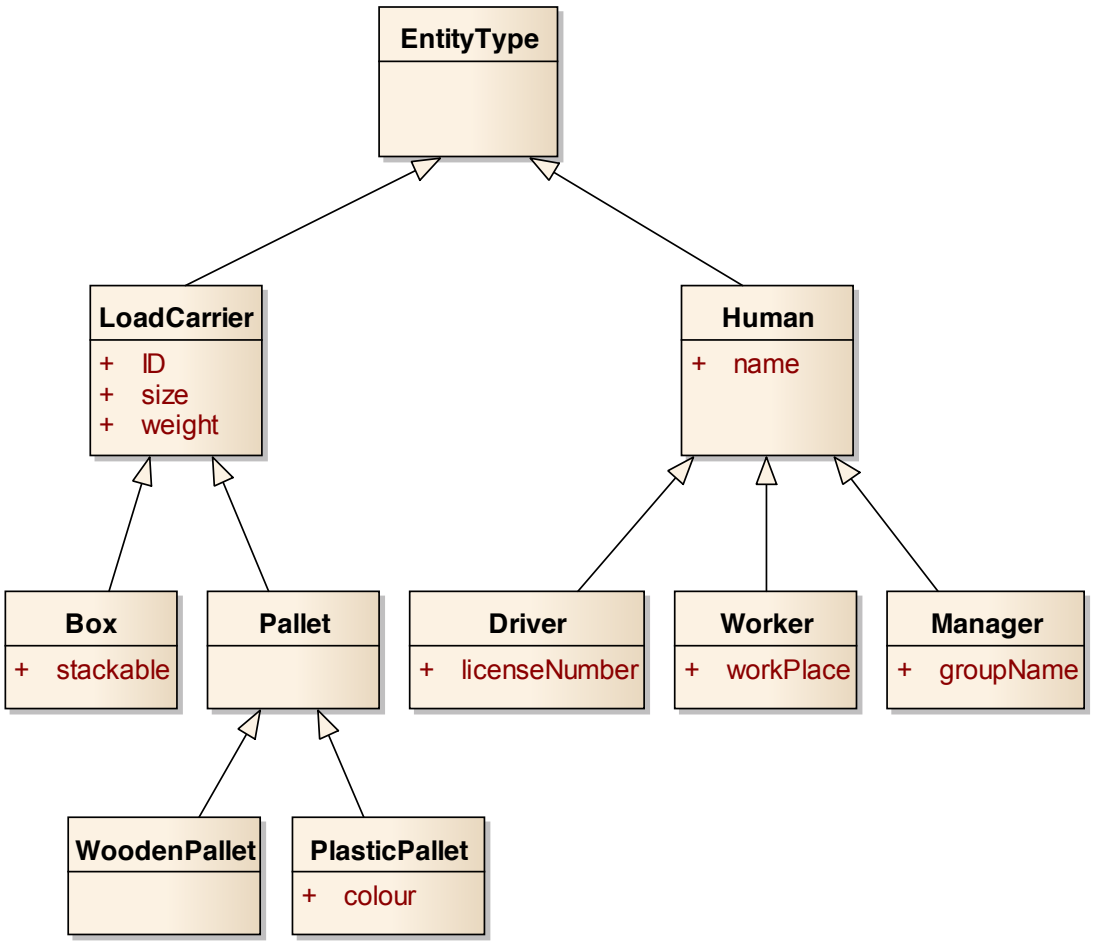

Figure 42: Example for hierarchical entityType model.

Viewpoint for modelling entityType hierarchies

EntityTypes are similar to classes in object-oriented programming, so UML class diagrams as shown above are suitable for modelling entityTypes. As shown in Figure 42: Example for hierarchical entityType model the generalization relation can be used for modelling sub-classes of entiyTypes, creating a hierarchy of several entityTypes inheriting attributes from its super-classes. Alternatively, ontology languages like OWL also provide the means for

modelling classes and sub-classes, so they can also be used for modelling type hierarchies. This is especially useful, if information in the IoT system is to be modelled using ontologies.

Service Descriptions

Services provide access to functions for retrieving information or executing actuation tasks on IoT Devices. As a basis for finding and interacting with services, services need to be appropriately described, which is done in the form of Service Descriptions. Service Descriptions contain information about the interface of the service, both on a syntactic as well as a semantic level, e.g. the required inputs, the provided outputs or the necessary pre-conditions as well as post-conditions. Furthermore, the Service Description may include information regarding the functionality of the resources, e.g. the type of resource, the processing method or algorithm etc., or information regarding the device on which the resource is running, e.g. it's hardware or its geographical location. Different specification languages for describing services are available, so again, there are different design choices.

Associations between Virtual Entities and Services

Services can provide information or enable actuation, but the services themselves may not be aware of e.g., which Virtual Entities can provide what information or can enable what kind of actuation. This information is captured by associations that relate to the Virtual Entity and the Service. The association includes the attribute of the Virtual Entity for which the Service provides the information or enables the actuation as a result of a change in its value.

4.2.3.2 Information Handling

Information in the system is handled by IoT Services. IoT Services may provide access to On-Device Resources, e.g. sensor resources, which make real-time information about the physical world accessible to the system. Other IoT Services may further process and aggregate the information provided by IoT Services/Resources, deriving additional higher-level information. Furthermore, information that has been gathered by the mentioned IoT Services or has been added directly by a user of the IoT system can be stored by a special class of IoT Service, the history storage. A history storage may exist on the level of data values directly gathered from sensor resources as a resource history storage or as a history storage providing information about a Virtual Entity as a Virtual Entity history storage.

IoT Services are registered to the IoT system using Service Descriptions. Service Descriptions can be provided by the services themselves, by users or by special management components that want to make the service visible and discoverable within the IoT system. The IoT Service Resolution is responsible for managing Service Descriptions and providing access to Service Descriptions. In detail, the IoT Service Resolution provides an interface for discovering Service Descriptions based on service specifications given by the requestor, for looking up a Service Description based on the identifier of a

service and for resolving a service identifier to a service locator. The latter can also be seen as a convenience function as the Service Description also contains the currently valid service locator.

Associations can be registered with the VE Resolution by services that know for what Virtual Entities they can provide information. The registration can be done by users, by special management components, or by the VE & IoT Service Monitoring component. The VE & IoT Service Monitoring component automatically derives the Associations based on information existing in the system, including Service Descriptions and other associations.

4.2.3.3 Information Handling by Functional Components

The following section describes how information is handled and exposed by the functional components in an IoT-system and shows the information flows between the functional components.

Before going into detail Figure 43 shows the information flow through the Functional Components based on the recurring example from Section 2.3. From the actuator on device level the temperature information is transferred to the IoT Service and afterwards to the VE Service. The VE Service itself is described in Section 3.4.2. From the VE Service the temperature value is transferred to the AndoidApp via the Subscribe/Notify-pattern.

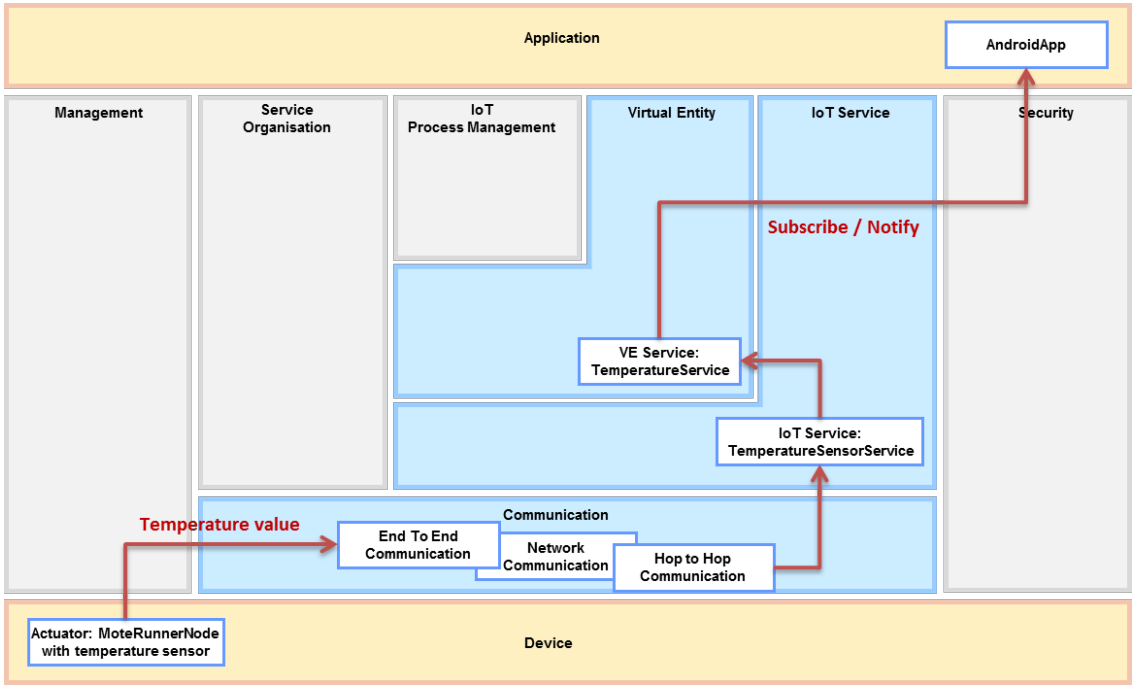

Figure 43: Information flow based on the “Read Thread”.

General information flow concepts

There are four message exchanges patterns considered for information exchange between IoT Functional Components. The first message exchange pattern is the Push-pattern, the second one is the Request/Response-pattern; the third one is the Subscribe/Notify-pattern, and the fourth one is the Publish/Subscribe-pattern. All patterns are explained in the following.

Push

The Push-pattern (see Figure 44) is a one-way communication between two parties in which a server sends data to a pre-defined client that receives the data. The server hereby knows the address of the client beforehand and the client is constantly awaiting messages from the server. The communication channel in this pattern is pre-defined and meant to be applied in scenarios in which the communication partners do not changed often. For example the server can be a constrained device that sends data to a gateway dedicated to this device. The gateway is listening constantly to the device and is consuming the data received from this device.

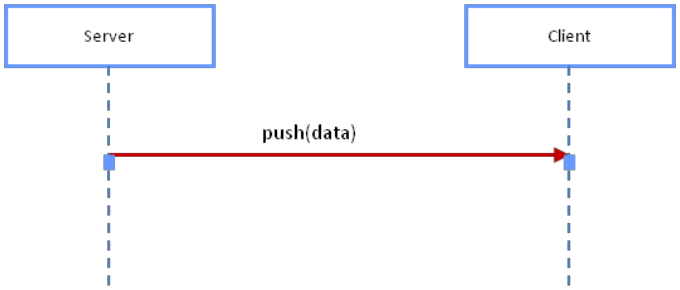

Figure 44: Push-pattern.

The Request/Response-pattern (see Figure 45 and Figure 46) is a synchronous way of communication between two parties. A client sends a request to a server. The server will receive the request and will send a response back to the client. The client is waiting for the response until the server has sent it.

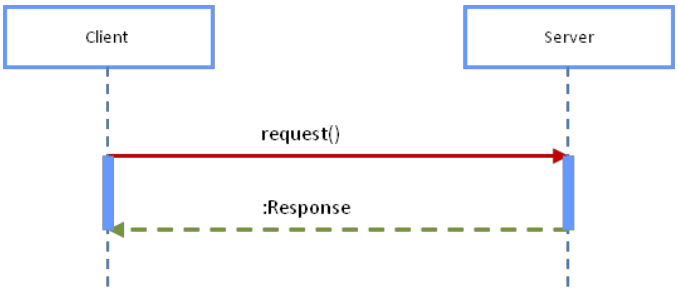

Figure 45: Request/Response-pattern for one client.

The server needs some time to prepare the response for the client. In the meanwhile another client might send a request. When the server is still busy with preparing the response for the first client it cannot produce the response for the second client. The second client will be placed into a queue until the server

is ready to prepare its response. Such scenario might lead to unacceptable response times.

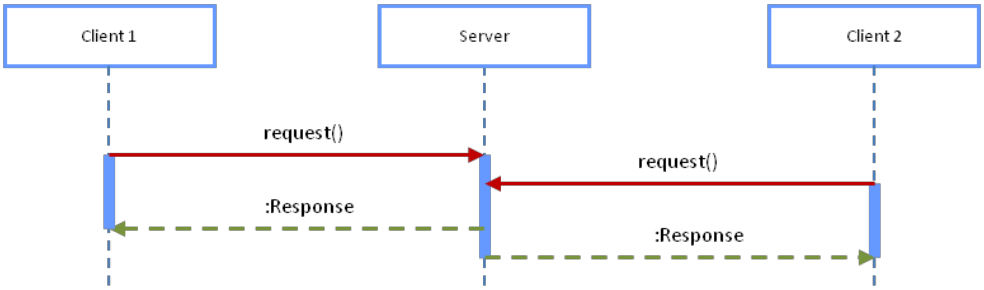

Figure 46: Request/Response-pattern for clients.

Subscribe/Notify

The Subscribe/Notify-pattern (see Figure 47 and Figure 48) allows an asynchronous way of communication between two parties without the client waiting for the server response. The client just indicates the interest in a service on the server by sending a subscribe-call to the server. The server stores the subscription together with the address of the client wants to get notified on and sends notifications to this address whenever they are ready to be sent.

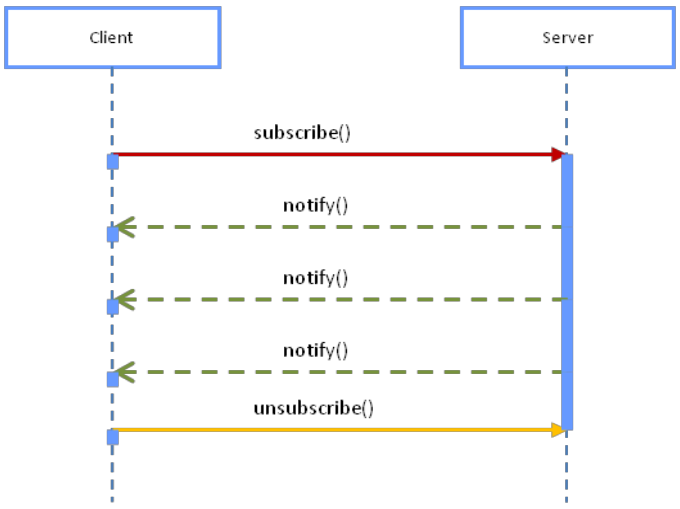

Figure 47: Subscribe/Notify-pattern for one client.

One advantage of the Subscribe/Notify-pattern over the Request/Response-pattern is the non-blocking behaviour of the subscribe method. The clients can continue with other task and need to process the notification only when it arrives. Another big advantage on the server side is that notifications can be multiplied and sent off to clients if the clients have subscribed to the same kind of notifications. To implement the Subscribe/Notify-pattern a server is required that is more powerful compared to the one required for the Request/Response-pattern. The server has to keep records about its subscribers and the kind of subscriptions if it allows several of them.

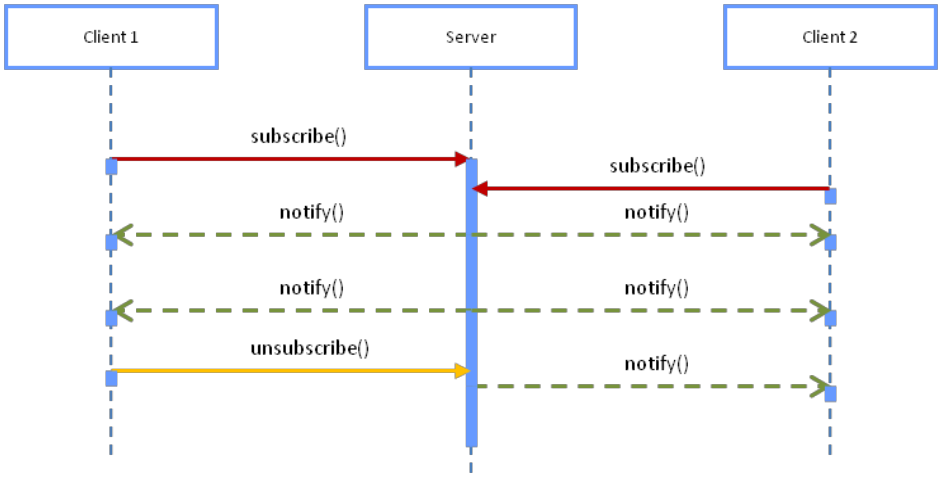

Figure 48: Subscribe/Notify-pattern for two clients.

Publish/Subscribe

The Publish/Subscribe-pattern (see Figure 49 and Figure 50) allows a loose coupling between communication partners. There are services offering information and advertise those offers on a broker component. When clients declare their interest in certain information on the broker the component will make sure the information flow between service and client will be established.

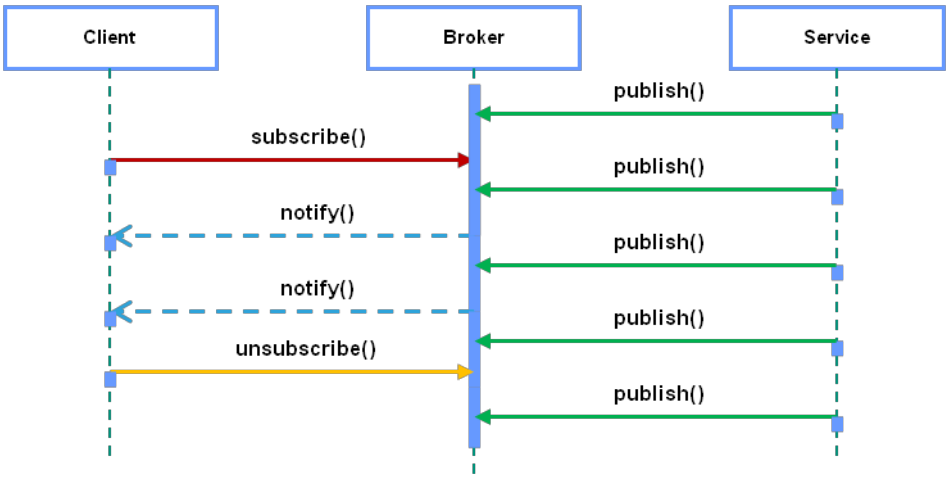

Figure 49 Publish/Subscribe-pattern

Services can publish information to the broker regardless how many clients are interested in this information; if no client has subscribed to it the broker does not forward the notification to any client, if more clients have subscribed to the same information the broker will multiply the information and send out notification to each subscriber.

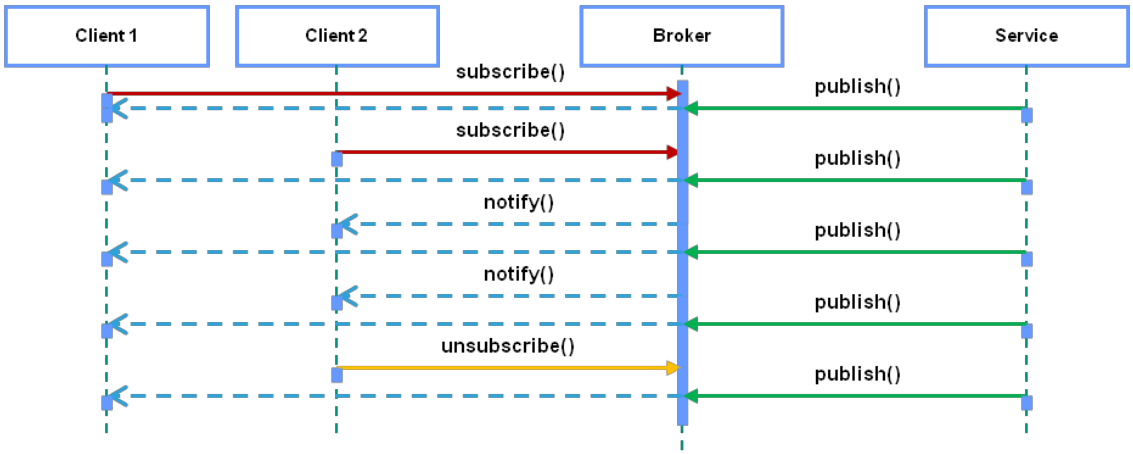

Figure 50 Publish/Subscribe-pattern 2 clients

Information flow through functional components

User requests information from IoT Service

Figure 51 shows the information request from a user to an IoT Service and the corresponding response.

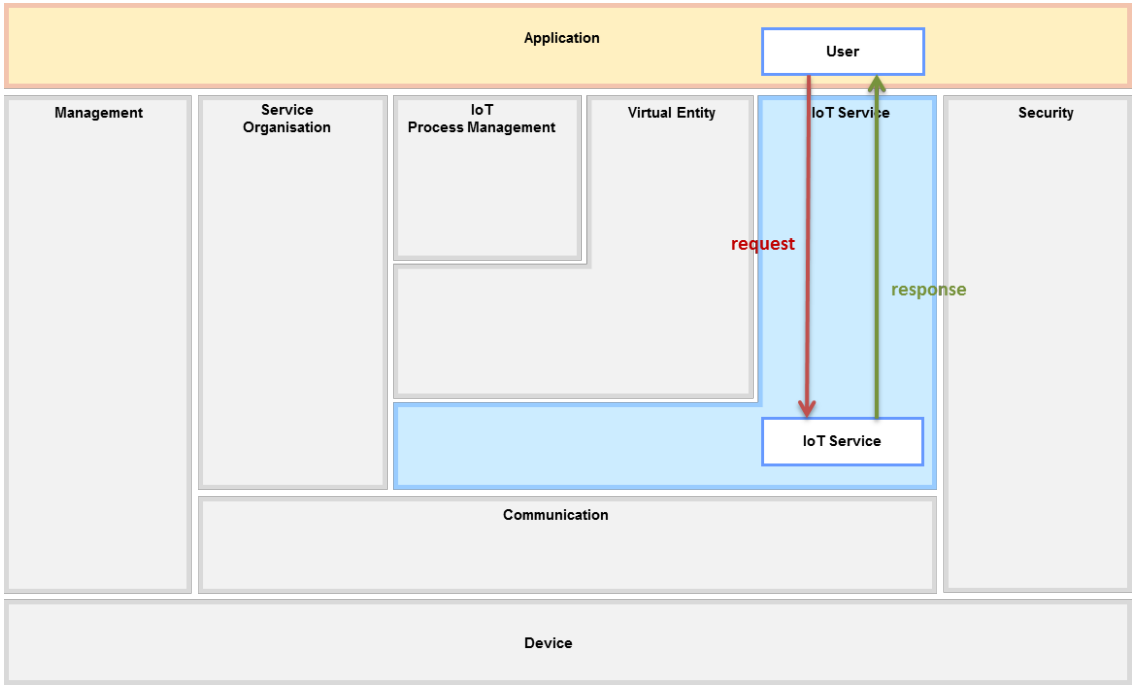

Figure 51: User requests IoT service.

User gets information from Virtual Entity-level service

Virtual Entity-level service provides access to Virtual Entity information, augmenting sensor information with entity information (entityId, entityType or several attributes), thus changing the abstraction level. Figure 52 shows the Subscribe/Notify-pattern, which can be used to get updates about an Attributes value.

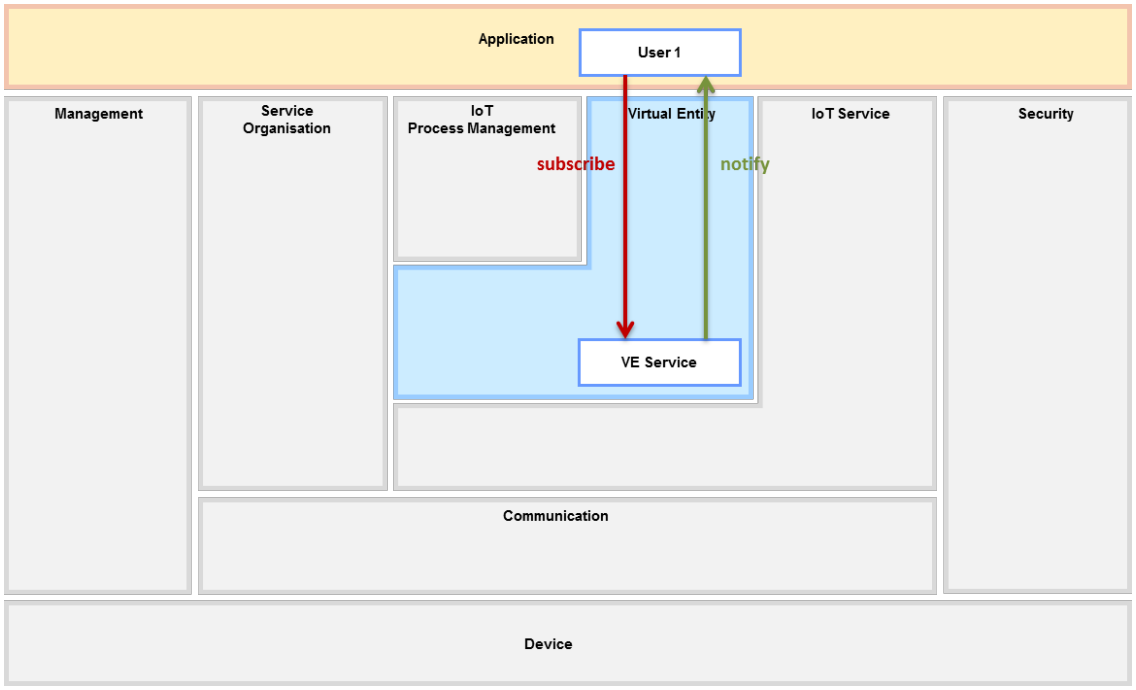

Figure 52: User subscribes for updates of VE-attribute.

Service gets sensor value from Device

The Sensor Device in Figure 53 pushes an updated sensor value using the Functional Component Flow Control & Reliability to an IoT Service. Besides the Push-pattern Request/Response and Subscribe/Notify-pattern are possible. Figure 54 shows a similar situation but the information is pushed up to the VE Service.

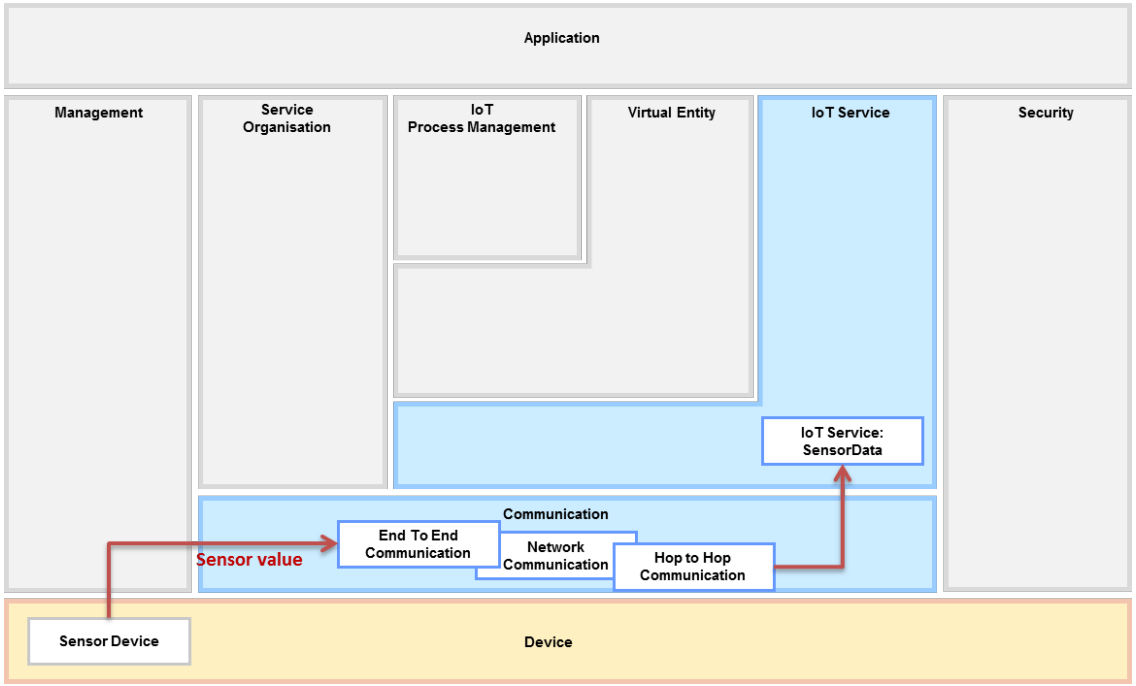

Figure 53: Information flow from Sensor Device to IoT Service using the Push-pattern.

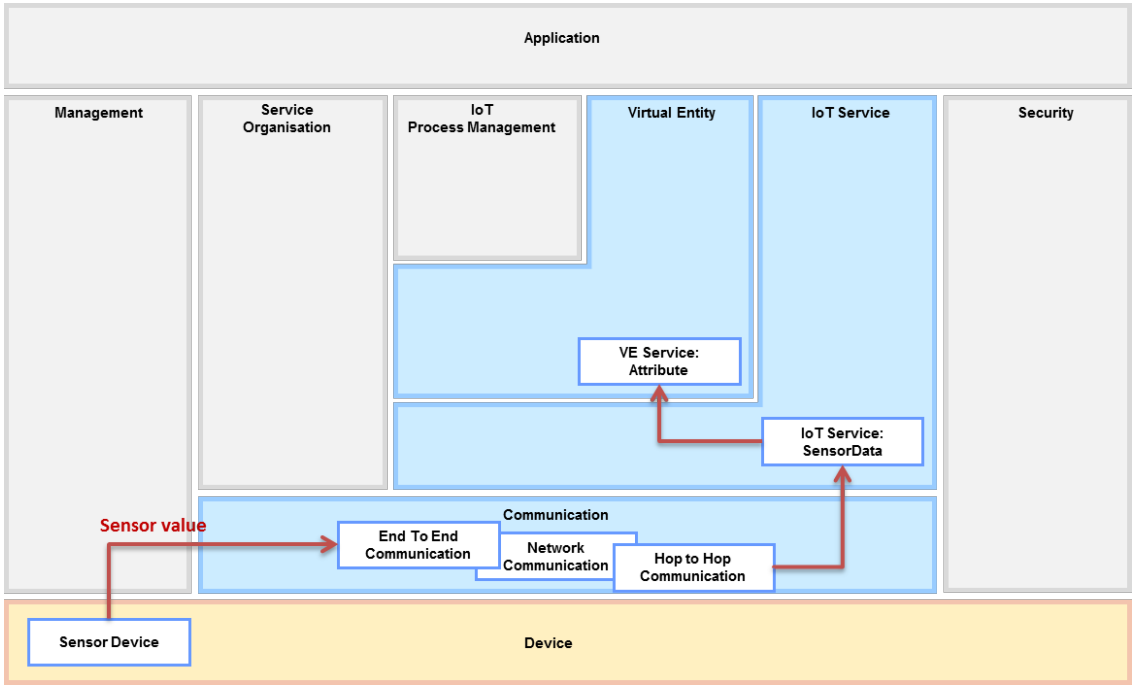

Figure 54: Information flow from Sensor Device to VE Service using the Push-pattern.

Sensor information storage

Figure 55 shows the special case of using an information storage device which stores additional, e.g. historic, values. The IoT Service DataStorage requests values and the StorageDevice sends the corresponding response. The storage policy of the Storage Device is application-specific, e.g. stores values only for certain duration, stores values with reduced granularity over time or in an averaged or aggregated form. Such a storage device can also be used from the VE Service level.

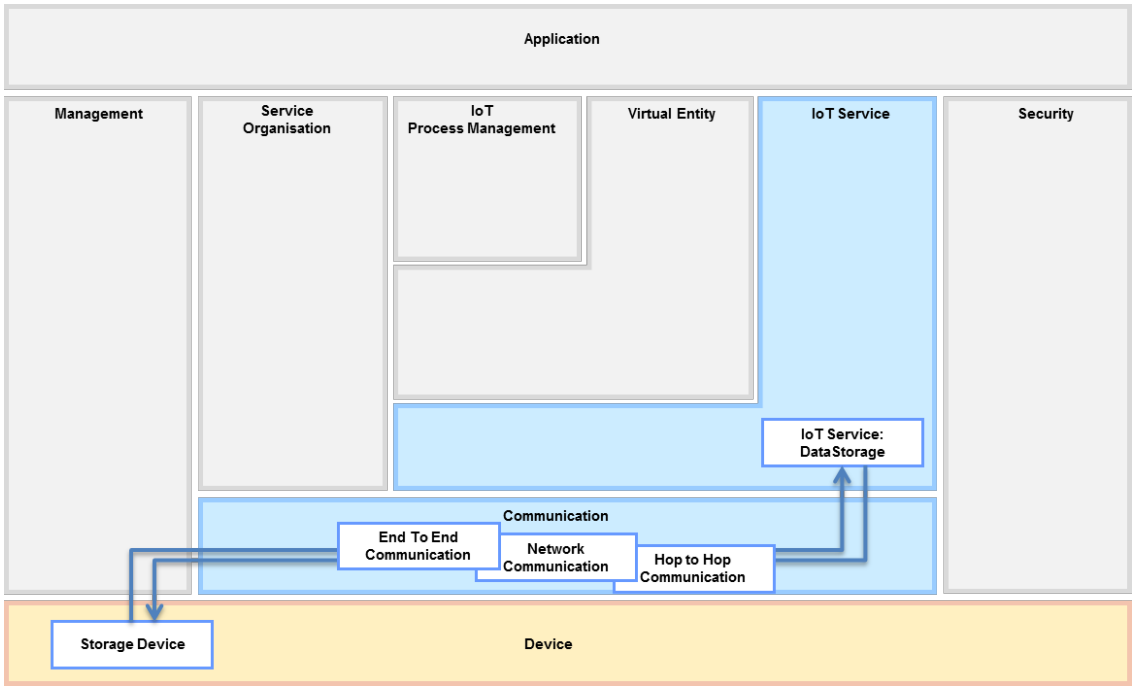

Figure 55: Usage of sensor information storage device.

IoT Service Resolution

The Functional Component IoT Service Resolution hosts the Service Descriptions that are needed for looking up and discovering IoT Services. Thus the resolution component offers methods to insert, update, and delete Service Descriptions (see Figure 56) according to the availability of IoT Services. The methods are meant to be invoked by the IoT Services itself, e.g. upon their deployment, dynamic change of location due to mobility or their undeployment from the system. It is also possible for the Service Management component to invoke these methods in order to maintain the system. For deleting a Service Description its Service ID needs to be given.

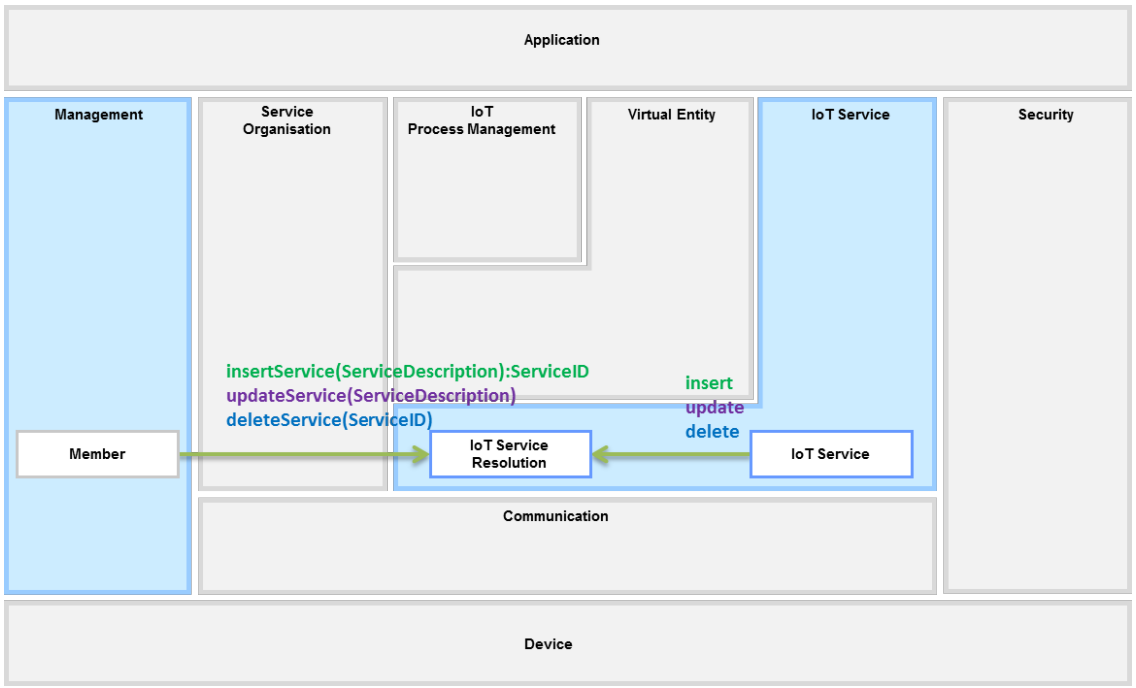

Figure 56: Insert, update, and delete Service Description.

The IoT Service Resolution component offers three methods to find IoT Services (see Figure 57 and Figure 58):

1. **look-up** of Service Description based on service identifier;
2. **discovery** of Service Descriptions based on service specification;
3. **resolution** of service identifier to service locator (contained in Service Description).

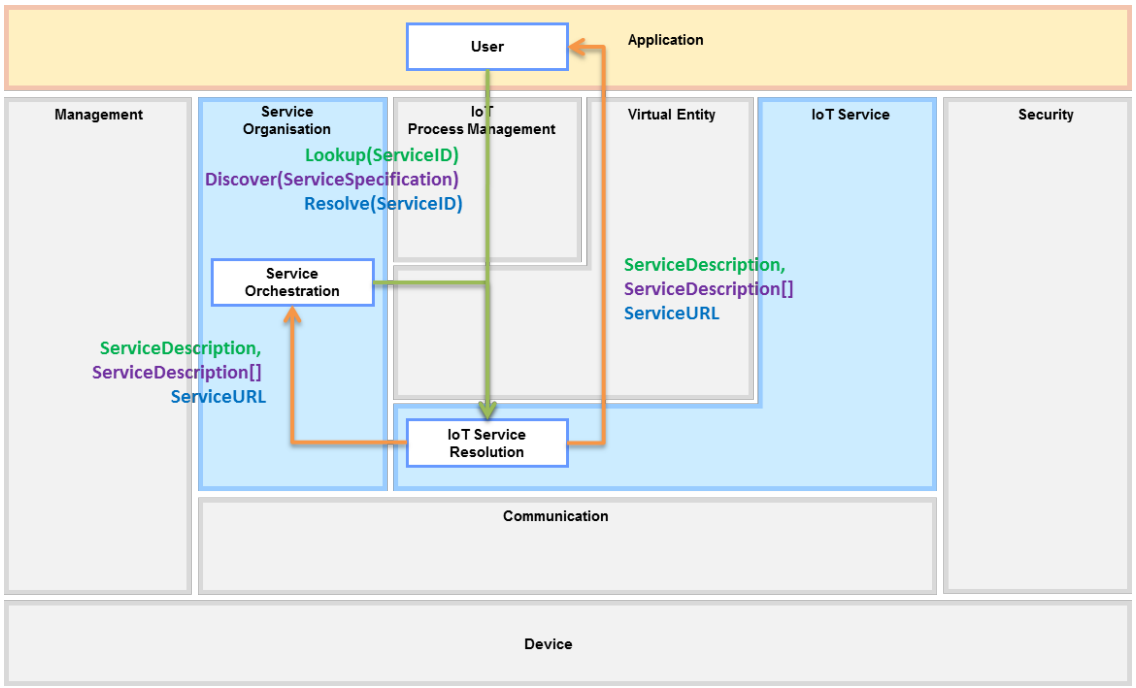

Figure 57: Request lookup, discover, and resolve IoT Services.

Figure 57 shows the different methods in a Request/Response manner, the component also offers similar functionality realised as Subscribe/Notify-pattern. The information flow is similar to the one according to Request/Response, but additionally identifiers for subscriptions and locators for call-back interfaces are exchanged as shown in Figure 58.

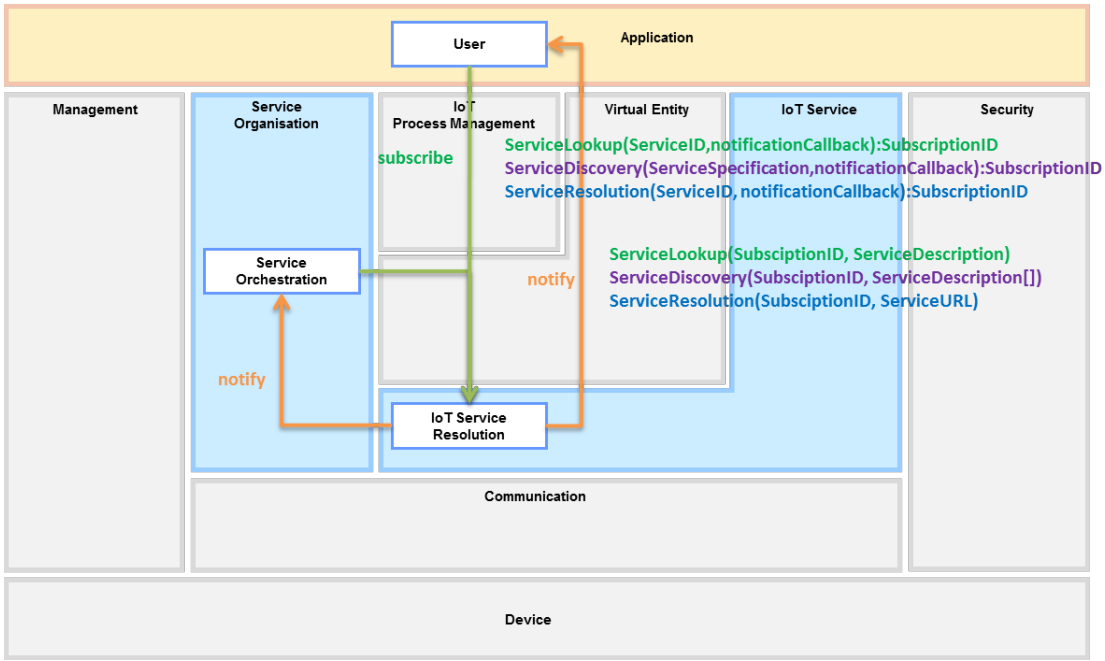

Figure 58: Subscribe to lookup, discover, and resolve IoT Services

VE Resolution

Associations between Virtual Entities and IoT Services are inserted into VE Resolution by IoT Services, the Service Management components or the VE & IoT Service Monitoring. They can later be updated and eventually deleted, e.g., when the IoT Service is undeployed. The message exchange is shown in Figure 59.

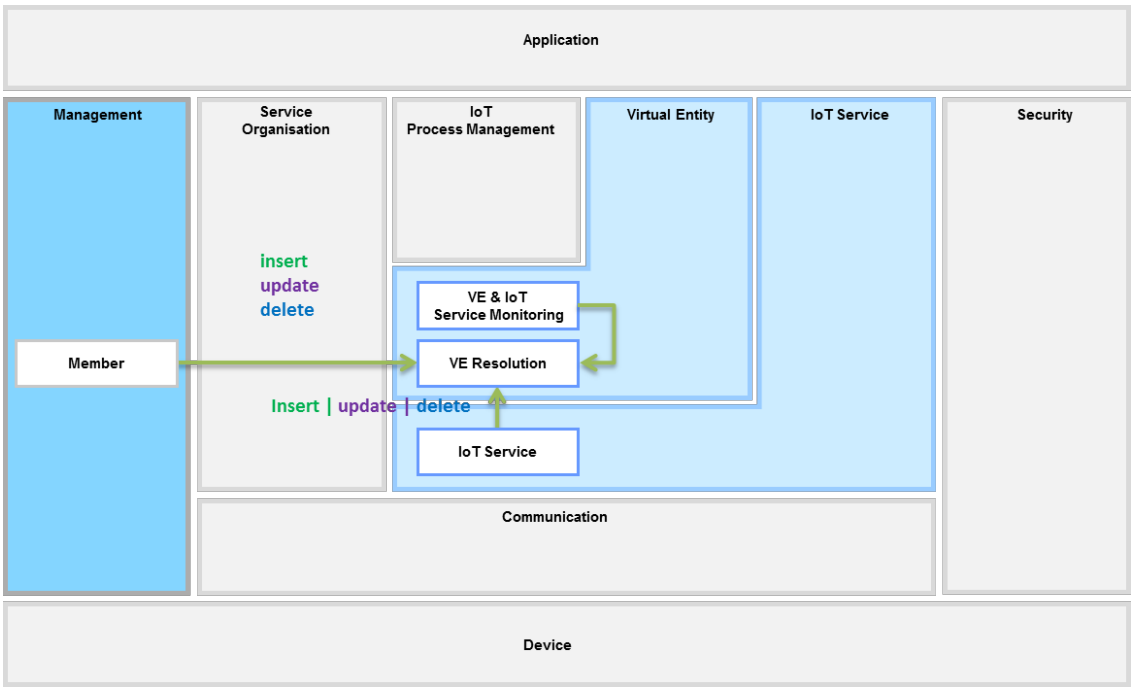

Figure 59: insert, update, and delete Association

The VE Resolution component allows retrieving of associations between Virtual Entities and IoT Services based on VE identifier and VE service specification through a lookup request as well as discovery of Associations based on VE specification and VE service specification as depicted in Figure 60.

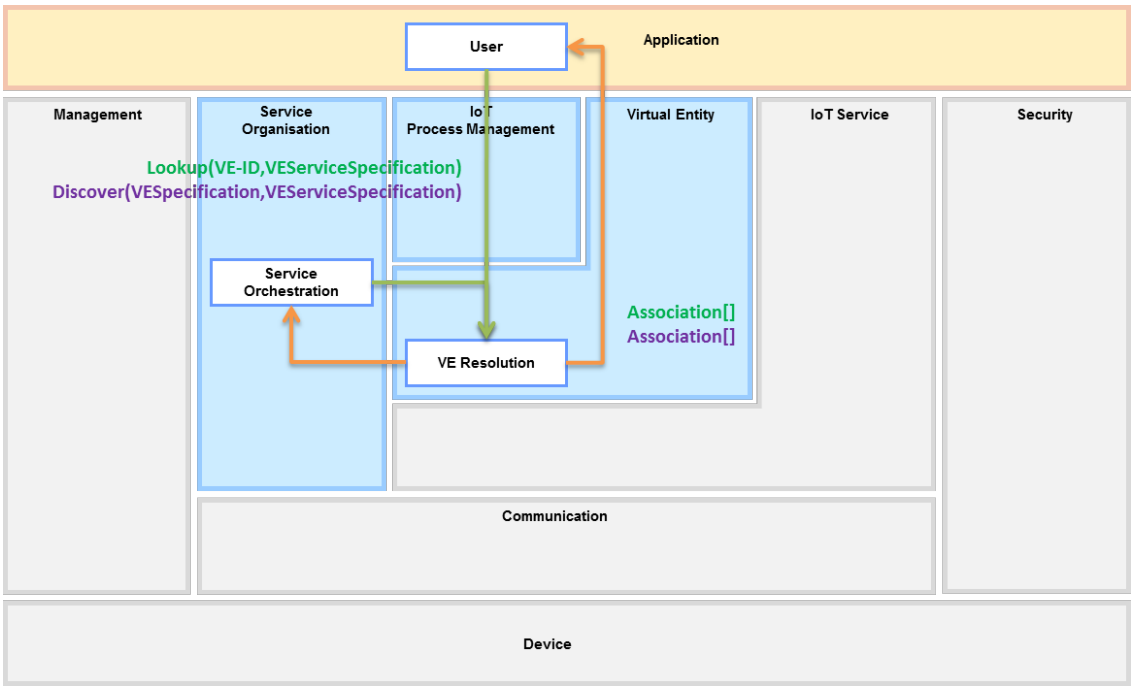

Figure 60: Request lookup and discover Associations

The VE Resolution component provides a information flow while applying the Subscribe/Notify-pattern. With this identifiers for subscriptions and locators for callback interfaces are exchanged additionally as shown in Figure 61.

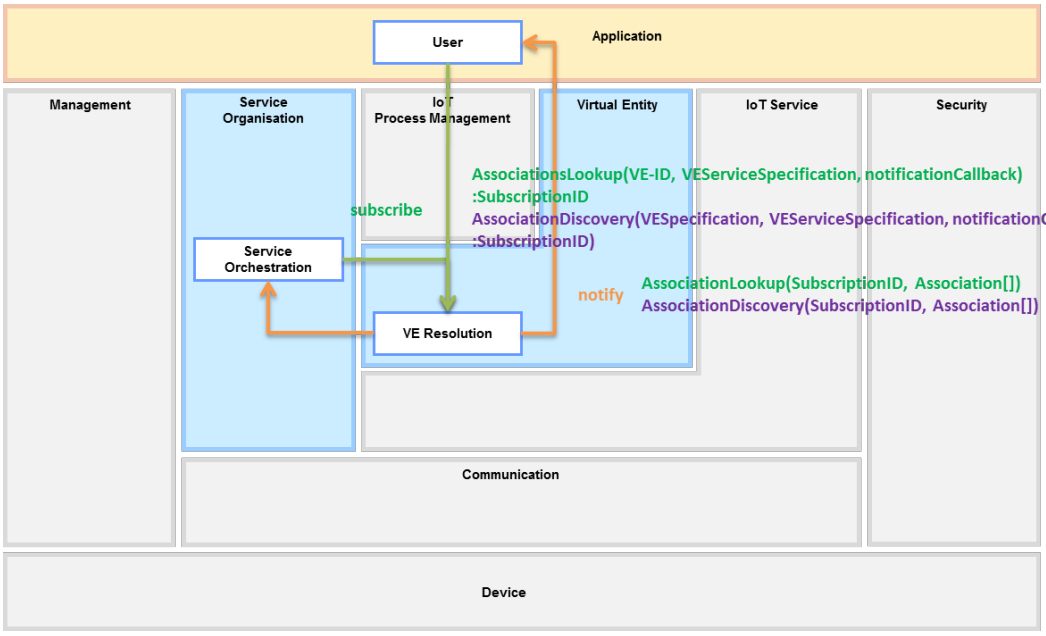

Figure 61: Subscribe to lookup and discover Associations

4.2.3.4 Information Life Cycle

Information provided by sensor resources is transient in nature and may not even be measured or observed without a specific request. Information stored by

a storage resource may be permanently stored there or have an expiry date after which the information is to be removed. For this purpose a storage resource may have to implement mechanisms that remove such information on a regular basis. It is also possible to adapt the granularity of information that is stored over time, i.e., for a certain time interval all the information is stored, for a further time interval only a fraction of the information is kept whereas the rest is discarded. Such a scheme may allow the definition of multiple such time intervals and also requires specific underlying mechanisms that can implement the scheme.

To avoid keeping Service Descriptions of services that no longer exist, a time-out mechanism needs to be implemented by the IoT Service Resolution. After the time-out has been reached without a renewal of the Service Description, the Service Description should automatically be removed. This in turn requires that the components originally providing the Service Description renew the registration of the Service Description before the time-out is reached. The same applies for associations stored by the VE Resolution.

4.2.4 Deployment & Operation view

Connected and smart objects in the IoT can be realized in many different ways and can communicate using many different technologies. Moreover, different systems may need to communicate the one to each other in a compliant way. Hence the Deployment and Operation view is very important to address how actual system can be realized by selecting technologies and making them communicate and operate in a comprehensive way.

The Deployment and Operation View aims at providing users of the IoT Reference Model with a set of guidelines to drive them through the different design choices that they have to face while designing the actual implementation of their services. To this extent this view will discuss how to move from the service description and the identification of the different functional elements to the selection among the many available technologies in the IoT to build up the overall networking behaviour for the deployment.

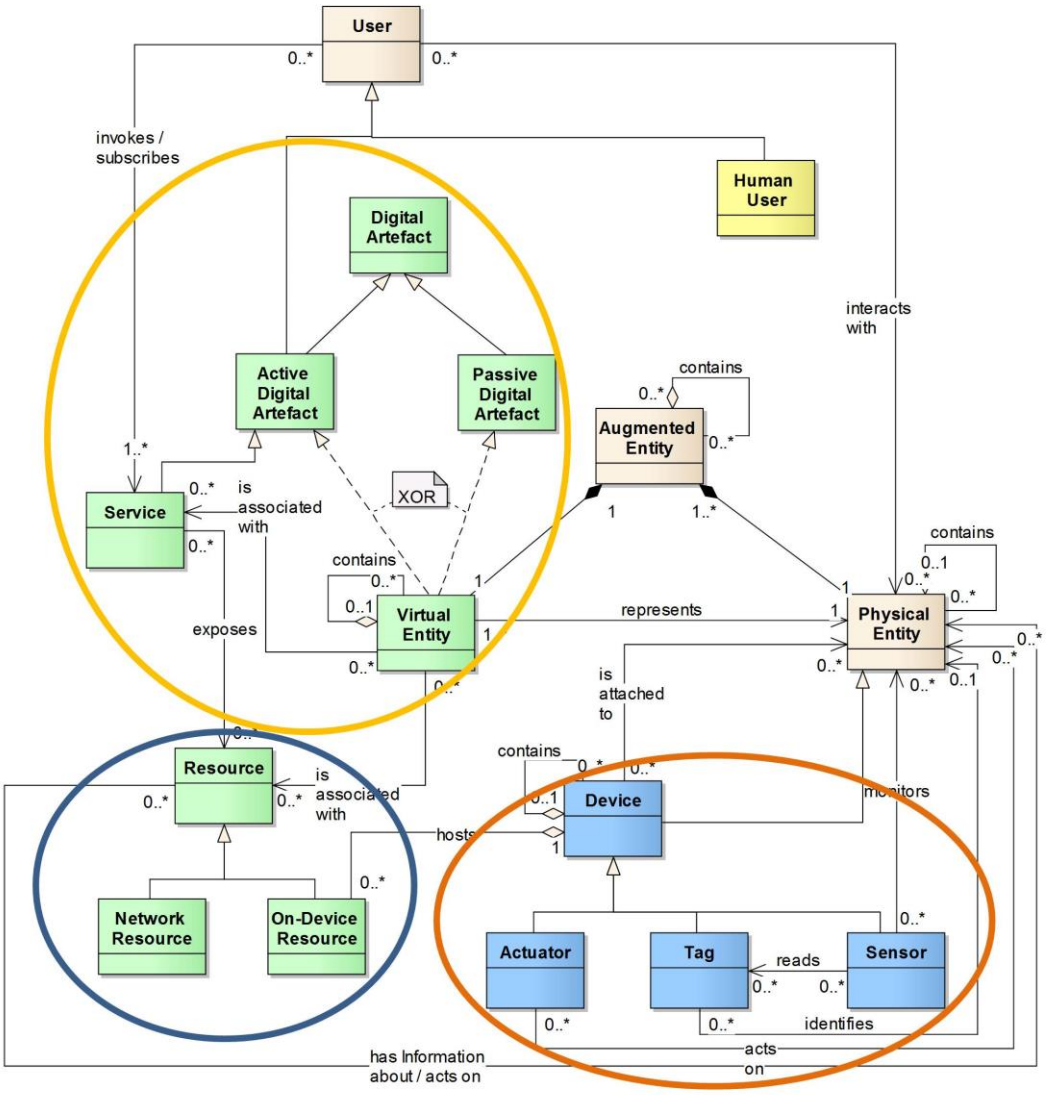

Figure 62: Domain Model elements grouped according to their common deployment aspects.

Since a complete analysis of all the technological possibilities and their combination falls beyond the scope of this view, this section will identify those categories that have the strongest impact on IoT systems realization. In particular, starting from the IoT Domain Model, we found three main element groups (see Figure 62): Devices, Resources, and Services highlighted in red, blue and yellow, respectively. Each of them poses a different deployment problem, which, in turn, reflects on the operational capabilities of the system.

In particular, the viewpoints used in the Deployment and Operation view are the following:

1. The IoT Domain Model diagram is used as a guideline to describe the specific application domain; to this extent UML diagrams can be used to further detail the interaction among the many elements composing the target application;
2. The Functional Model is used as a reference to the system definition; in particular it defines Functional Groups such as IoT Services and Connectivity groups which are fundamental for a correct definition of the system;
3. Network connectivity diagrams can be used to plan the connectivity topology to enable the desired networking capability of the target application; at the deployment level, the connectivity diagram will be used to define the hierarchies and the type of the sub-networks composing the complete system network;
4. Device Descriptions (such as datasheets and users manuals) can be used to map actual hardware on the service and resource requirements of the target system.

First of all, devices in IoT systems include the whole spectrum of technologies ranging from the simplest of the radiofrequency tags to the most complex servers. The unifying characteristics are mainly two-fold: on the one hand, every device is connected with one another forming a part of the IoT; and, on the other hand, every device is “smart”, even though with different degree of complexity, in that it provides computational capabilities. These two characteristics are the subject of the first choices a system designer has to make. Note that, for a given device to be fully interoperable in an IoT-A compliant system, it must respect the functionality definitions of the Functional Model. However, legacy systems that do not fully support the FM, may implement wrappers and adaptation software to comply to the model.

Selecting the computational complexity for a given device is somewhat intrinsic to the target application. However, choosing among the different connectivity types is not as straightforward as different choices may provide comparable advantages, but in different areas. For the same reason, it is possible to realize different systems implementing the same or similar application from the functional view, which are extremely different from the Deployment and Operation view. In this section, we will simply detail the main options for device connectivity, leaving their impact on the different perspective for Section 5.2.10 in which the design choices for the deployment view are discussed. The following list provides a few of the typical technologies that can be found in IoT systems:

- Sensor & Actuator Networks;
- RFIDs and smart tags;
- WiFi or other unconstrained technologies;

- Cellular networks.

As a consequence of the coexistence of different communication technologies in the same system, the second choice the system designer must account for is related to communication protocols. In particular, connectivity functionalities for IoT system are defined in this document in Communication FG of the FM; in addition, in order to better understand the application, it is important to describe it within the Functional View. Although, IoT-A and WP3 in particular suggest a communication protocol suite aimed at the interoperability among different technologies with IP as the common denominator, the system designer, may be forced to make suboptimal choices [Rossi 2012] and, [Rossi 2013]. In particular, we identified the following possibilities:

1) **IoT protocol suite**: This is the main direction supported by this project and providing the best solution for interoperability;

2) **Ad-hoc proprietary solutions**: Whenever the performance requirements of the target application are more important than the system versatility, ad hoc solutions may be the only way to go;

3) **Other standards**: Depending on the target application domain, regulations may exist forcing the system designer to adopt standards, different from those suggested by the IoT protocol suite, that solved a given past issue and have been maintained for continuity.

After having selected the devices and their communication methods, the system designer has to account for services and resources, as defined in the IoT Service FG section. These are pieces of software that range from simple binary application and increasing their complexity up to full-blown control software. Both in the case of resources and for services the key point here is to choose where to deploy the software related to a given device. The options are as follows:

1) **On smart objects**: This choice applies to simple resource definitions and lightweight services, such as web-services that may be realized in few tens or hundreds of bytes;

2) **On gateways**: Whenever the target devices are not powerful enough to run the needed software themselves, gateways or other more capable devices have to be deployed to assist the less capable ones;

3) **In the cloud**: Software can be also deployed on web-farms. This solution improves the availability of the services, but may decrease the performance in terms of latency and throughput.

Note that this choice has to be made per type of resource and service and depending on the related device. As an example, a temperature sensor can be deployed on a wireless constrained device, which is capable of hosting the temperature resource with a simple service for providing it, but, if a more complex service (for instance, when the Service Organisation FG is called in) is

needed, the software has to deployed on a more powerful device as per option 2) or 3).

On the same line, it is important to select where to store the information collected by the system, let their data be gathered by sensor networks or through additional information provided by users. In such a choice, a designer must take into consideration the sensitiveness (e.g.: is the device capable of running the security framework), the needed data availability and the degree of redundancy needed for data resiliency. The foreseen options are the following:

1. **Local only**: Data is stored on the device that produced it, only. In such a case, the locality of data is enforced and the system does not require complex distributed databases, but, depending on the location of a given request, the response might take longer time to be delivered and, in the worst case scenario, it may get lost;
2. **Web only**: No local copy is maintained by devices. As soon as data is sent to the aggregator, they are dispatched in databases;
3. **Local with web cache**: A hierarchical structure for storing data is maintained from devices up to database servers.

Finally, one of the core features of IoT systems is the resolution of services and entities, which is provided by the Entity and Service Resolution FCs, respectively and is in charge of semantically retrieving resources and services, discovering new elements and binding users with data, resources, and services. In particular, this is performed adopting the definitions of the Virtual Entity FG. This choice, while one of the most important for the designer, has only two options:

1. **Internal deployment**: The core engine is installed on servers belonging to the system and is dedicated to the target application or shared between different applications of the same provider;
2. **External usage**: The core engine is provided by a third party and the system designer has to drive the service development on the third party APIs.

Differently from the other choices, this is driven by the cost associated to the maintenance of the core engine software. In fact, since it is a critical component of the system, security, availability and robustness must be enforced. Hence, for small enterprises the most feasible solution is the external one.

4.2.4.1 Deployment example

Coming back to our “Red Thread” example, this section analyzes the system deployment for the “Transport monitoring with Smart Load Carriers” scenario.

First of all, we need to define the purpose of the application(s), the functionalities and their requirements for a correct operating behaviour and the data that needs to be treated.

Purpose: the application measures several environmental parameters of the load carrier such as the light, the temperature and the humidity of the truck and monitors the status of the several installed devices.

Functionalities:

- **Monitoring**: the application needs to provide the users with means to access information gathered by many sensors installed in the truck;
- **Controlling**: the application needs to provide users with means to modify the behaviour of the many installed devices;
- **Alarm**: the application needs to provide users with means to configure alarms to be triggered when a given condition is verified (e.g.: the temperature rises over a threshold value).

Requirements:

- **Lifetime**: all the installed devices must operate unassisted for more than two years;
- **Robustness**: a maximum data loss of 5% of the information is tolerated and no command nor alarm loss can be tolerated;
- **Responsiveness**: a maximum delay of 10 seconds is tolerated when issuing a command and for alarm reporting. A maximum delay of 15 minutes is tolerated for data reporting in steady state condition.

Data: all the information managed by the system is not sensitive and does not require for high security.

As a second step, the system integrator must define the Virtual Entities and the Services to be used in the application. To keep the example simple, we will define a single Service and a single Virtual Entity only. The service will be in charge of monitoring the sensing units and to provide users with interface to access the data. We will call this service “Monitoring service”. For what concerns the Virtual Entity we choose to represent a room in the house as a Virtual Entity, which is connected to the room Physical Entity and with the resources provided by the Sensors (Device) installed in the truck.

Basically, the application can be simply implemented by allowing the Service to query the Resources of the associated Virtual Entities periodically. However, many possibilities are left to the integrator for the actual deployment of the application.

Resources: it is clear that Resources must provide a connection between the sensing Devices and the Service, but the actual software harmonizing the Sensor behaviour with the service language can be run either on the sensing Device itself, in a gateway device connecting the house network with the external network, or directly in the cloud. The most versatile solution is to run the Resource software directly on the Device in order to enable any other

Service to query directly the Device for the needed information; however, depending on the actual hardware capabilities, the other two solutions can be considered.

Service: it must be possible to access the monitoring service from anywhere there is an Internet connection, and, in particular, from within the house. Note that, users using the service from within the house may be less tolerant to delays. A typical service deployment in this case is to have two paired services providing the same monitoring functionality: one is running in a local server and is able to directly query the devices in order to fetch up to date information, the second is running in the cloud and provides accessibility from the Internet. Note that the local service is also maintaining an information database of the data gathered in the house; database, which is only accessed by the service in the cloud.

Finally, the system integrator must make decisions about connectivity and data management: since the time requirements of the application are quite loose, low power devices can be chosen and low data rate connection can be selected for the sensing devices.

The first and foremost requirement is the addressability of every Service / Resource regardless of the Device hosting it. This can be achieved by supporting IP addressing and its compressed version defined by 6LoWPAN is currently the most feasible way to implement this in constrained devices. In addition, to make Resources and Services unambiguously addressable, unique identifier must be provided. To this extent many solutions have been proposed, but, in order to obtain the widest interoperability, it is preferable HTTP mappable solutions, such as CoAP. In such a way it is possible to implement very simple Services on the most constrained Device by providing web-service like interaction capabilities to every resource and functionality offered.

However, if the above baseline solution is not realizable, it is important to mimic its behaviour as close to the source device is located. To this extent Resources, Services or both can be deployed on other devices such as aggregator servers, gateways and proxies of the network. In such a way, it is the more powerful Device providing Resource and Service in the correct format that will interact with Services and Users on behalf of the final Device; also, this device must ensure the synchronization between the mimicked functionalities and their actual counterparts. This workaround allows for the integration of any possible technologies in the IoT, however it does not grant the full compliance to all the IoT-A unified requirement list.

However, in order to make the sensing devices interoperable with both the local and the cloud services, connectivity gateways or proxies must be considered. A few possible realizations are the following:

- Cabled sensors with Ethernet/xDSL gateway

- Pros: reliable, possibility to use the same cable for connectivity and power.
- Cons: high installation costs.

- Wireless sensors (802.15.4) with Ethernet/xDSL gateway
  - Pros: low cost, easy and cheap installation, moderate robustness, good lifetime.
  - Cons: may suffer from data losses.
- Low power WiFi sensors with WiFi/xDSL gateway
  - Pros: moderate costs, easy gateway implementation, easy and cheap installation, higher data rate is possible.
  - Cons: shorter lifetime than 802.15.4

The following figure (Figure 63) shows the deployment example above, highlighting the several physical devices involved (dark green), the different network type involved (solid horizontal lines) and the software installed per device (white/cyan rounded boxes, cyan is for mandatory parts while cyan is for optional elements).

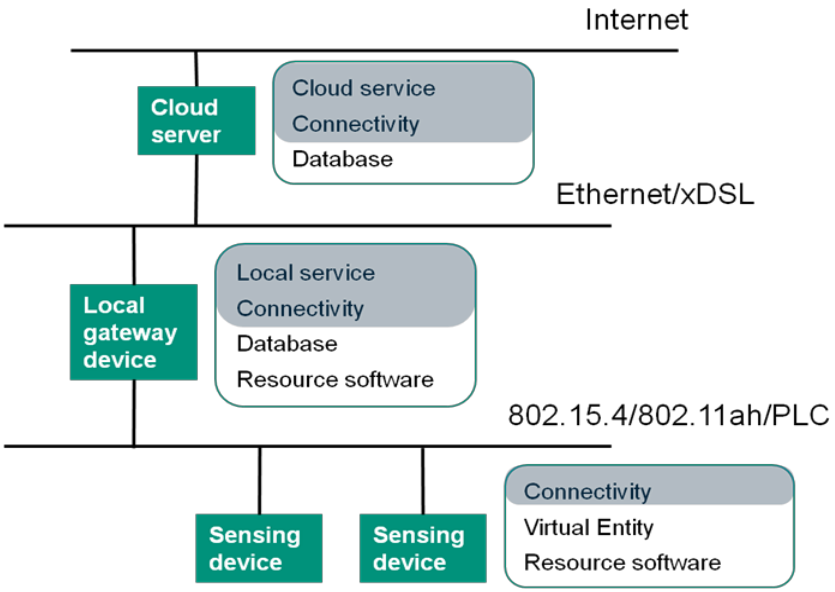

Figure 63: Transport monitoring example with possible deployment choices highlighted.

Although this example is quite simple, it can be used as a building block for more complex scenarios. In particular it is important here to understand how to separate the different networks in the system, where to deploy each functionality and which connectivity type to use per sub-network.

4.3 Perspectives

Architectural decisions often address concerns that are common to more than one view, or even all of them. These concerns are often related to non-functional or quality properties. We are following the approach of [Rozanski 2005], which suggests special perspectives to address these aspects of a concrete architecture. They emphasize the importance of stakeholder requirements just like we do within our project. Therefore we are adopting their definition of perspective for usage within IoT-A:

> An **architectural perspective** is a collection of activities, tactics, and guidelines that are used to ensure that a system exhibits a particular set of related quality properties that require consideration across a number of the system's architectural views. [Rozanski 2005]

where a quality property is defined as:

> A **quality property** is an externally visible, non-functional property of a system such as performance, security, or scalability [Rozanski 2005]

The stakeholder requirements clearly show a need of addressing non-functional requirements. Based on them, we identified the perspectives, which are most important for IoT-systems:

- Evolution and Interoperability;
- Availability and Resilience;
- Trust, Security and Privacy and
- Performance and Scalability.

As these requirements are requiring some kind of quality for a real system, the perspectives aim more at the concrete system architecture, than at a Reference Architecture.

We got 18 requirements concerning the Evolution and Interoperability perspective, 15 concerning Availability and Resilience, more than 20 related to Trust, Security and Privacy, and 17 more related to Performance and Scalability. As can be seen from the definition above there is a close relationship between Perspectives, Views and Guidance.

We will generally follow the structure as suggested by Rozanski/Woods, to describe the perspectives, but adjusted according to our needs. Each perspective contains the following information:

Desired Quality	The desired quality that the perspective is addressing
IoT-A Requirements	The IoT-A requirements this perspective addresses
Applicability	The Applicability of the perspective, e. g. the types of

	systems to which the perspective is applicable
Activities	A set of possible activities that are suggested to achieve the desired qualities. We are reusing existing literature, as well as, our own identified best practices here.
Tactics	Here we list Architectural Tactics, which an architect can use when designing the system.

An architectural tactic is defined as follows:

An architectural **tactic** is a design decision for realizing quality goals at the architectural level.

It can already be seen from the definition of tactic that there is a close relationship to the design decisions as outlined in Section 5.2.10. We therefore will list high-level design choices as architectural tactics whenever feasible.

We think that taking advantage of perspectives makes a lot of sense for every software architect, even more in the IoT-domain where a lot of Quality parameters have to be taken into account. Perspectives provide a framework for reusing knowledge: It is absolutely necessary to apply a systematic approach to ensure that a certain system fulfils the required quality properties. The use of Perspectives, combined with Views and Guidelines is a step towards that. In the “Guidance” Chapter in Section 5.4.4 we present an suggested usage of the perspectives in conjunction with Design Choices.

4.3.1 Evolution and interoperability

The Evolution and Interoperability perspective addresses the fact that requirements change and software evolves sometimes rapidly and need to interoperate not only with todays technologies, but also needs to be prepared to interoperate with later technologies. Interoperability therefore plays especially in IoT a crucial role. The vision of the Internet of Things is still evolving itself. There are, for example, not yet all used technologies mature enough, and there are for sure many more technologies to come in the future.

Desired Quality	The ability of the system to be flexible in the face of the inevitable change that all systems experience after deployment, balanced against the costs of providing such flexibility
IoT-A Requirements	UNI.003, UNI.010, UNI.012, UNI.023, UNI.042, UNI.047, UNI.048, UNI.071, UNI.093, UNI.094, UNI.096, UNI.417, UNI.422, UNI.432, UNI.509, UNI.701, UNI.712, UNI.720

Applicability	Important for all systems to some extent; more important for longer- lived and more widely used systems. IoT systems are expected, as an emerging technology, to be highly affected by evolution and interoperability issues.
Activities	Characterize the evolution needs Assess the current ease of evolution Consider the evolution trade-offs Rework the architecture
Tactics	Contain change Create extensible interfaces Apply design techniques that facilitate change Apply metamodel-based architectural styles Build variation points into the software Use standard extension points Achieve reliable change Preserve development environments

Table 5: Evolution and Interoperability (adopted from [Rozanski 2005]), extended with IoT specific aspects

4.3.2 Performance and scalability

This perspective addresses two quality properties that are closely related: Performance and Scalability. Both are, compared to traditional information systems, even harder to cope with in a highly distributed scenario as we have it in IoT.

Desired Quality	The ability of the system to predictably execute within its mandated performance profile and to handle increased processing volumes in the future if required
IoT-A Requirements	UNI.008, UNI.026, UNI.027, UNI.028, UNI.066, UNI.089, UNI.101, UNI.102, UNI.234, UNI.511, UNI.512, UNI.615, UNI.706, UNI.708, UNI.711, UNI.716, UNI.717
Applicability	Any system with complex, unclear, or ambitious performance requirements; systems whose architecture includes elements whose performance is unknown; and systems where future expansion is likely to be significant. IoT systems are very likely to have unclear performance characteristics, due to their heterogeneity and high connectivity of devices.
Activities	Capture the performance requirements Create the performance models Analyze the performance model Conduct practical testing

	Assess against the requirements Rework the architecture
Tactics	Optimize repeated processing Reduce contention via replication Prioritize processing Consolidate related workload Distribute processing over time Minimize the use of shared resources Reuse resources and results Partition and parallelize Scale up or scale out Degrade gracefully Use asynchronous processing Relax transactional consistency Make design compromises

Table 6: Performance and Scalability (adopted from [Rozanski 2005]), extended with IoT specific aspects

4.3.3 Trust, Security and Privacy

This chapter addresses fundamental properties of IoT systems for what concerns their relation to the user. They are inter-related and, often, the evaluation or the improvement of one of these qualities is necessarily related to the others.

4.3.3.1 Trust

Trust in the IoT environment is a fundamental yet difficult to obtain quality. As the security one, this quality is highly dependent on the computation and communication performance of the system. In the frame of IoT moreover, M2M subjects must be enabled to evaluate this quality in order to obtain autonomous systems.

Desired Quality	A complex quality related to the extent to which a subject expects (subjectively) an IoT system to be dependable regarding in all the aspects of its functional behaviour.
IoT-A Requirements	UNI.062, UNI.065, UNI.099, UNI.407, UNI.408, UNI.602, UNI.604, UNI.605, UNI.620, UNI.622
Applicability	Relevant to the systems that share the use of resources with subjects that are not a priori trusted.
Activities	Capture trust requirements Perform risk analysis Check interoperability requirements and their impact on trust between heterogeneous subjects Define trust model

	Consider risks derived from malicious or unintentional misuse of IoT systems[5]
Tactics	Harden root of trust Ensure physical security and implement tampering detection Ensure and check data freshness Consider the impact of security/performance tradeoffs on trust Use (trusted) infrastructural Trust and Reputation Agents for scalability Use security imprinting Check system integrity often Balance privacy vs. non-repudiation (accountability)

Table 7: Trust Perspective (extended from [Rozanski 2005])

4.3.3.2 Security

Security is an essential quality of an IoT system and it is tightly related to specific security features which are often a basic prerequisite for enabling Trust and Privacy qualities in a system.

[5] For example, simulating traffic by broadcasting car-to-infrastructure signals or inducing emergency maneuvers in ships or planes by simulating adverse environmental conditions. Generally, it is possible to make a fictional situation credible if the assumption that Physical and Virtual Entities are always and securely synchronized is overlooked.

Desired Quality	Ability of the system to enforce the intended confidentiality, integrity and service access policies and to detect and recover from failure in these security mechanisms.
IoT-A Requirements	UNI.062, UNI.407, UNI.408, UNI.410, UNI.412, UNI.413, UNI.424, UNI.502, UNI.507, UNI.604, UNI.609, UNI.611, UNI.612, UNI.617, UNI.618, UNI.624, UNI.719
Applicability	Relevant to all IoT systems.
Activities	Capture the security requirements Check interoperability requirements for impacts on security processes between heterogenous peers Conduct risk analysis Use infrastructural Authentication components that support more Identity Frameworks for scalability and interoperability Use infrastructural or federated Key Exchange Management to secure communication initiation and tunnelling between gateways for interoperability Use an Authorization component to enable interoperability with other systems Define security impact on interaction model Address all aspects of Service and Communication Security Integrate the trust model and support privacy features Identify security hardware requirements Consider performance/security tradeoffs Validate against requirements
Tactics	Use an extended Internet Threat Model for which takes into account specific IoT communication vulnerabilities Harden infrastructural functional components Authenticate subjects Define and enforce access policies Secure communication infrastructure (gateways, infrastructure services) Secure communication between subjects Secure peripheral networks (data link layer security, network entry, secure routing, mobility and handover) Avoid wherever possible wireless communication Physically protect peripheral devices or consider peripheral devices as available to malicious users in the attacker model

	Avoid Over-The-Air device management; if necessary secure properly

Table 8: Security perspective (adopted from [Rozanski 2005] , extended with IoT specific aspects)

4.3.3.3 Privacy

There are usually different concepts conveyed with the term privacy, some being more from the technical side and some more from the legal perspective, without forgetting ethical aspects.

Desired Quality	Ability of the system to ensure that the collection of personally identifying information be minimized and that collected data should be used locally wherever possible.
IoT-A Requirements	UNI.001, UNI.002, UNI.410, UNI.412, UNI.413, UNI.424, UNI.501, UNI.606, UNI.611, UNI.623, UNI.624
Applicability	Relevant to all IoT systems.
Activities	Capture the privacy requirements Conduct risk analysis Evaluate compliancy with existing privacy frameworks.
Tactics	Use an Identity Management component that supports Pseudonymization Avoid transmitting identifiers in clear especially over wireless connections Minimize unauthorized access to implicit information (e.g. deriving location information from service access requests) Validate against requirements Consider the impact of security/performance tradeoffs on privacy Enable the user to control the privacy (and thus security and trust) settings Balance privacy vs. non-repudiation (accountability)

Table 9: Privacy perspective (adopted from [Rozanski 2005] , extended with IoT specific aspects)

4.3.4 Availability and resilience

As we are dealing with distributed IoT systems, where a lot of things can go wrong, the ability of the system to stay operational and to effectively handle failures that could affect a system's availability is crucial.

Desired Quality	The ability of the system to be fully or partly operational as and when required and to effectively handle failures that could affect system availability
IoT-A Requirements	Uni.040, UNI.050, UNI.058, UNI.060, UNI.064, UNI.065, UNI.092, UNI.230, UNI.232, UNI.233, UNI.601, UNI.604, UNI.610, UNI.616, UNI.718
Applicability	Any system that has complex or extended availability requirements, complex recovery processes, or a high profile (e.g., is visible to the public)
Activities	Capture the availability requirements Produce the availability schedule Estimate platform availability Estimate functional availability Assess against the requirements Rework the architecture
Tactics	Select fault-tolerant hardware Use high-availability clustering and load balancing Log transactions Apply software availability solutions Select or create fault-tolerant software Design for failure Allow for component replication Relax transactional consistency Identify backup and disaster recovery solution

Table 10: Availability and resilience (adopted from [Rozanski 2005] , extended with IoT specific aspects)

4.4 Conclusion

The chapter has given an overview about the current state of the IoT Reference Architecture that is meant to be applied to any IoT architecture. The IoT Reference Architecture abstracts from domain specific use cases; it rather focuses on the domain agnostic aspects that IoT architectures may have in common. It does not mean that every IoTarchitecture has to implement every feature listed here, but in this report we have covered functional as well as non-functional aspects that are important to support in today's IoT-solutions on one hand and that are important to the stakeholders we have interviewed on the other hand. Following our architectural methodology we presented several views and perspectives of the IoT Reference Architecture.

The Functional View describes the functional building blocks of the architecture and the Deployment and Operation View explains the operational behaviour of the functional components and the interplay of them.

The Information View shows how the information flow is routed through the system and what requests are needed to query for or to subscribe to information offered by certain functional components.

The perspectives listed in this chapter tackle the non-functional requirements IoT architectures might have. The perspectives are categorised according to the non-functional requirements that have been extracted from the Unified Requirements (UNIs) gathered in WP6. As a result of the requirement analysis we have categorised the required system attributes into the four perspectives “Evolution and Interoperability”, “Performance and Scalability”, “Trust, Security and Privacy”, and “Availability and Resilience”.

For each of the perspectives we list a number of tactics to achieve the desired attribute of the system, e.g. anonymous usage. The tactics are state-of–the art methodologies commonly used in today’s systems architectures.

In Chapter 5 “Guidance” we present examples of Design Choices for the respective tactics listed in the perspectives section as example solutions for non-functional architectural requirements. The Design Choices will help the architect with selecting suitable solutions for non-functional architectural problems to focus on the domain-specific functional aspects.