



Katholieke
Universiteit
Leuven

Department of
Computer Science

Shared Internet Of Things Infrastructure Platform: The Complete Architecture Software Architecture (H09B5a and H07Z9a) – Part 2b

FILIPCIKOVA-HALILOVIC

Monika Filipcikova (r0683254)
Armin Halilovic(r0679689)

Academic year 2016–2017

Contents

1	Introduction	2
2	Architectural Decisions	3
2.1	ReqX: Requirement Name	3
2.2	Other decisions	3
2.2.1	Decision 1	4
2.3	Discussion	4
3	Client-server view (UML Component diagram)	5
4	Decomposition view (UML Component diagram)	7
5	Deployment view (UML Deployment diagram)	9
6	Scenarios	11
7	Element Catalog and Datatypes	12
7.1	Element catalog	12
7.1.1	ComponentA	12
7.1.2	ComponentZ	12
7.2	Common interfaces	13
7.3	Defined Exceptions	13
7.4	Defined data types	13
8	Attribute-driven design documentation	14
8.1	Introduction	14
8.2	Decomposition 1: SIoTIP System (Av3, UC14, UC15, UC18)	14
8.2.1	Module to decompose	14
8.2.2	Selected architectural drivers	14
8.2.3	Architectural design	14
8.2.4	Instantiation and allocation of functionality	16
8.2.5	Interfaces for child modules	17
8.2.6	Data type definitions	20
8.2.7	Verify and refine	20
8.3	Decomposition 2: OtherFunctionality1 (M1, P2, UC11)	23
8.3.1	Module to decompose	23
8.3.2	Selected architectural drivers	23
8.3.3	Architectural design	23
8.3.4	Instantiation and allocation of functionality	24
8.3.5	Interfaces for child modules	24
8.3.6	Data type definitions	28
8.3.7	Verify and refine	28
8.4	Decomposition X: DRIVERS (Elements/Subsystem to decompose/expand)	31
8.4.1	Elements/Subsystem to decompose/expand	31
8.4.2	Selected architectural drivers	31
8.4.3	Architectural design	31
8.4.4	Instantiation and allocation of functionality	31
8.4.5	Interfaces for child modules	32
8.4.6	Data type definitions	32

1. Introduction

”Since you are a two-student group, you can omit/disregard the low-priority non-functional requirements (i.e., P2, U1, M2). Please make sure all other elements of the assignment are provided.”

Changes to decompositions 1 and 2.

2. Architectural Decisions

Note: This section discusses *all* your architectural decisions *in-depth*. First, *all* decisions related to the non-functionals are discussed in detail. Next, *all* other decisions are listed and discussed.

Hint: Don't just say *what* you have done. Explain *why* you have done it.

2.1 ReqX: Requirement Name

TODO: Use this section structure for each requirement

Key Decisions

TODO: Briefly list your key architectural decisions. Pay attention to the solutions that you employed (in your own terms or using tactics and/or patterns).

- decision 1
- ...

Employed tactics and patterns: ...

Rationale

TODO: Describe the design choices related to *ReqX* together with the rationale of why these choices were made.

Considered Alternatives

Alternative(s) for choice 1 Explain what alternative(s) you considered for this design choice and why they were not selected.

Deployment Decisions

...

Considered Deployment Alternatives

...

2.2 Other decisions

TODO: *Optional* If you have made any other important architectural decisions that do not directly fit in the sections of the other qualities you can mention them here. Follow the same structure as above.

2.2.1 Decision 1

KeyDecisions

...

Rationale

...

Considered Alternatives

...


Deployment Decisions

...

Considered Deployment Alternatives

...

2.3 Discussion

 **TODO:** Use this section to discuss your architecture in retrospect. For example, what are the strong points of your architecture? What are the weak points? Is there anything you would have done otherwise with your current experience? Are there any remarks about the architecture that you would give to your customers? Etc.

3. Client-server view (UML Component diagram)

Figures

3.1	Context diagram for the client-server view.	5
3.2	Primary diagram of the client-server view.	6

✓ **Hint:** No need to just repeat what we can see on the diagram.

Don't do this: *As you can see on fig. x: comp A consists of B and C, and C connects to D.*

But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

✓ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.

If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

✓ **Hint:** If you have any doubts about the size of your figures, it is better to make your figure too large than too small. Alternatively, you can test the readability by printing it.

⚠ **Attention:** With regard to the context diagram, recall the lectures on what it means and should contain. Be sure not to miss any elements here. This is a frequent source of errors.

⚠ **Attention:** Make sure your main component-and-connector and context diagrams are consistent.

📄 **TODO:** The context diagram of the client-server view: Discuss which components communicate with external components and what these external components represent.

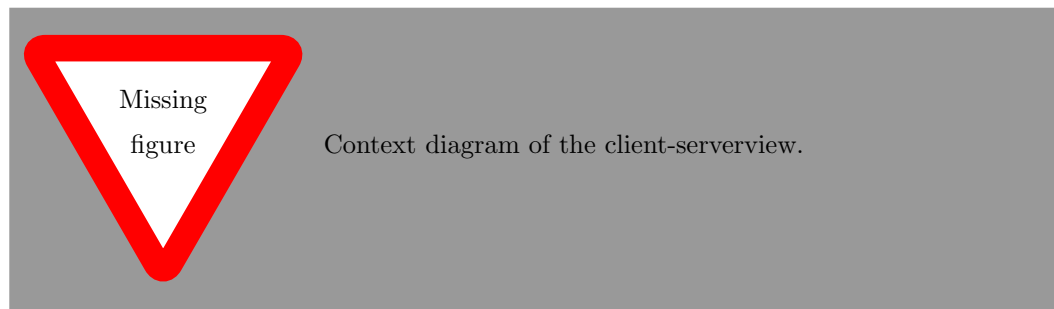


Figure 3.1: Context diagram for the client-server view.

📄 **TODO:** The primary diagram and accompanying explanation.

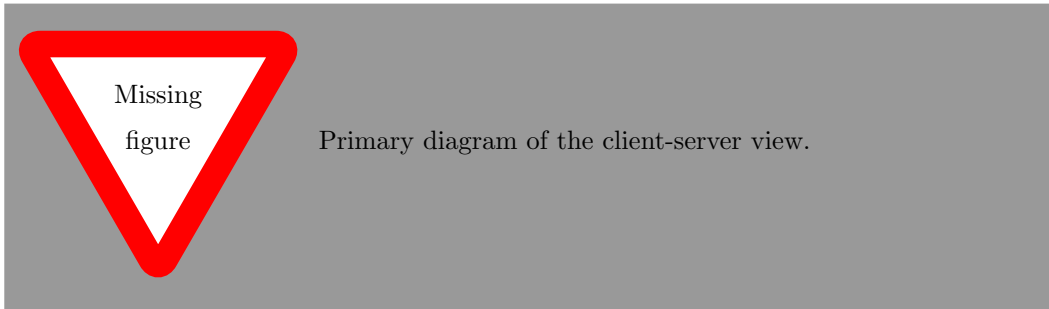


Figure 3.2: Primary diagram of the client-server view.

4. Decomposition view (UML Component diagram)

Figures

4.1	Decomposition of <code>ComponentX</code>	7
4.2	Decomposition of <code>ComponentY</code>	8

✓ **Hint:** No need to just repeat what we can see on the diagram.
Don't do this: *As you can see on fig. x: comp A consists of B and C, and C connects to D.*
But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

✓ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.
If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

▲ **Attention:** *Consistency between views!* Be sure to check for consistency between the client-server view and your decompositions.

▲ **Attention:** *Consistency of a single decomposition!* Make sure that every interface provided or required by the decomposed component, is provided or required by a subcomponent in the decomposition.

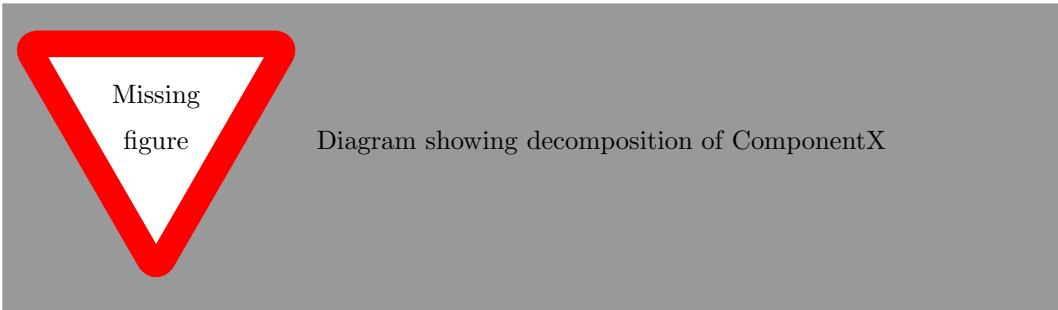


Figure 4.1: Decomposition of `ComponentX`

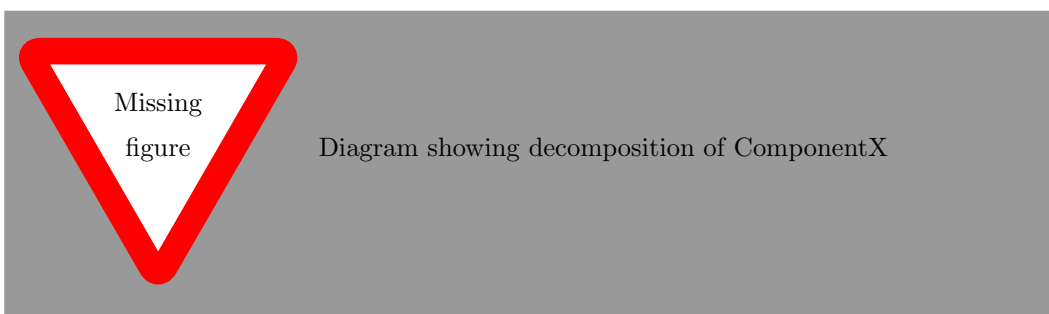


Figure 4.2: Decomposition of **ComponentY**.

This caption contains a longer explanation over multiple lines. This additional explanation is not shown in the list of figures.

5. Deployment view (UML Deployment diagram)

Figures

5.1	Context diagram for the deployment view.	9
5.2	Primary diagram for the deployment view.	10

✓ **Hint:** No need to just repeat what we can see on the diagram.

Don't do this: *As you can see on fig. x: components A and B are deployed on node C.*

But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

✓ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.

If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

⚠ **Attention:** Connect nodes on the deployment diagram, *not* components.

⚠ **Attention:** *Consistency between views!* Be sure to check for consistency between the client-server/decomposition view and your deployment view.

📖 **TODO:** Describe the context diagram for the deployment view. For example, which protocols are used for communication with external systems and why?

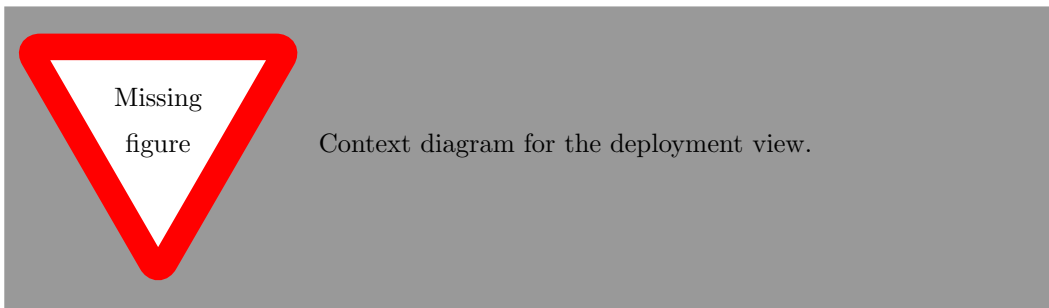


Figure 5.1: Context diagram for the deployment view.

📖 **TODO:** The primary deployment diagram itself. This discussion on the parts of the deployment diagram which are crucial for achieving certain non-functional requirements, and any alternative deployments that you considered, should be in the architectural decisions chapter.

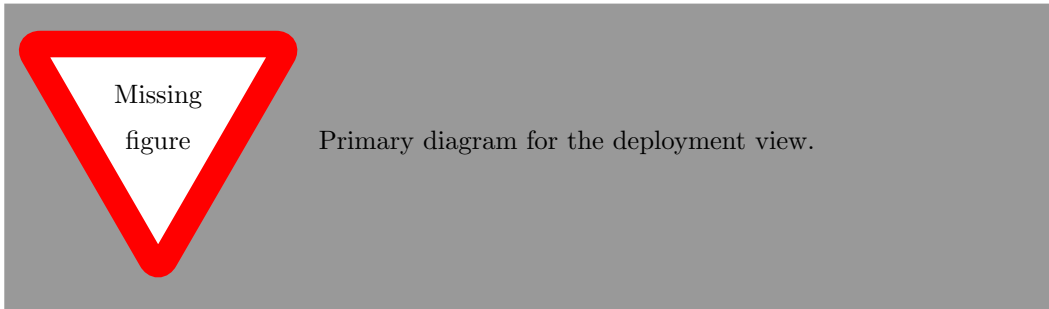


Figure 5.2: Primary diagram for the deployment view.

6. Scenarios

Figures

6.1 Scenario 1	11
--------------------------	----

✓ **Hint:** No need to just repeat what we can see on the diagram.

Don't do this: *As you can see on fig. x: component A calls operation b, next component C calls operation d.* But, please do explain if there is anything non-trivial (e.g., a custom mapping from actors to external components on the context diagram).

✓ **Hint:** Add any essential information, necessary for interpreting the figure, in the caption. Be sure to add a separate short title for inclusion in the list of figures: `\caption[shorttitle]{longtitle}`.

If your explanation becomes too long for the caption, you can create a separate subsection. Don't forget to refer to the figure and vice versa.

⚠ **Attention:** Do include a list of which sequence diagrams together illustrate a which scenario from the assignment.

✓ **Hint:** Don't only model the 'happy path' in your sequence diagrams. Take into account the quality attributes. For example, what happens when a certain component fails (Av) or overloads (P)? Use the sequence diagrams to illustrate how you have achieved the qualities in your architecture.

📌 **TODO:** Illustrate how your architecture fulfills the most important data flows. As a rule of thumb, focus on the scenario of the assignment. Describe the scenario in terms of architectural components using UML Sequence diagrams and further explain the most important interactions in text. Illustrating the scenarios serves as a quick validation of the completeness of your architecture. If you notice at this point that for some reason, certain functionality or qualities are not addressed sufficiently in your architecture, it suffices to document this, together with a rationale of why this is the case according to you. You do not have to further refine your architecture at this point.

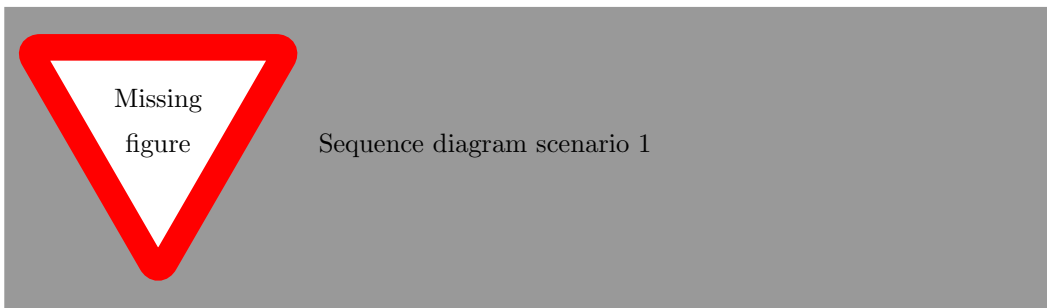


Figure 6.1: The system behavior for the first scenario.

7. Element Catalog and Datatypes

✓ **Hint:** Make sure the elements are sorted alphabetically. You can use the `\componentItem{name}{contents}` command for this in your report. Note that you cannot use newlines in the `componentItem` content, but you can use `\\`.

✓ **Hint:** Common interfaces such as, for example, `ping` can be described separately so you don't have to repeat them for each component that provides it.

✓ **Hint:** Similarly, you can describe the exceptions separately as well, so you don't have to repeat what they mean for each operation that can throw them.

⚠ **Attention:** Don't forget to include the exceptions in the method signature in the element catalog!

⚠ **Attention:** Interfaces are uniquely identified by their name, regardless of the context (e.g., the component that provides it). In other words, two interfaces with the same name are considered identical.

⚠ **Attention:** Don't forget to document the external interfaces!

7.1 Element catalog

📌 **TODO:** List all components and describe their responsibilities and provided interfaces. Per interface, list all methods using a Java-like syntax and describe their effect and exceptions if any. List all elements and interfaces alphabetically for ease of navigation.

7.1.1 ComponentA

- **Responsibility:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.

Provided interfaces

- InterfaceC
 - `returnType1 operation1(ParamType param) throws SomeException`
 - * Effect: Describe the effect of the operation
 - `void operation2(ParamType2 param)`
 - * Effect: Describe the effect of the operation
- InterfaceD
 - `returnType2 operation3()`
 - * Effect: Describe the effect of the operation


7.1.2 ComponentZ

- **Responsibility:** Responsibilities of the component.
- **Super-component:** The direct super-component, if any.
- **Sub-components:** the direct sub-components, if any.


Provided interfaces

- InterfaceA
 - `returnType1 operation1(ParamType param) throws SomeException`
 - * Effect: Describe the effect of the operation
 - `void operation2(ParamType2 param)`
 - * Effect: Describe the effect of the operation
 - * Exceptions: None
- InterfaceB
 - `returnType2 operation3()`
 - * Effect: Describe the effect of the operation


7.2 Common interfaces

 **TODO:** If you have any common interfaces used by multiple components you may define them here and refer to them.

7.3 Defined Exceptions

 **TODO:** Instead of describing the exceptions with each operation, you may define common exceptions here and refer to them from the operation definition.

7.4 Defined data types

 **TODO:** List and describe all data types defined in your interface specifications. List them alphabetically for ease of navigation.

- Paramtype1: Description of data type.
- Paramtype2: Description of data type.
- returnType1: Description of data type.

8. Attribute-driven design documentation

8.1 Introduction

Explain we changed ADD. List of our changes here.
Changed decomposition 1 and 2 compared to part 2a.

8.2 Decomposition 1: SIIoTIP System (Av3, UC14, UC15, UC18)

8.2.1 Module to decompose

In this run we decompose the SIIoTIP System.

8.2.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av3*: Pluggable device or mote failure

The related functional drivers are:

- *UC14*: Send heartbeat (Av3)
This use case checks whether or not motes and pluggable devices are still operational.
- *UC15*: Send notification (Av3)
This use case sends a notification to a registered user.
- *UC18*: Check and deactivate applications (Av3)
This use case deactivates any application that requires deactivation, because of unavailability of essential pluggable devices or unassigned mandatory roles.

Rationale Av3 was chosen first since it has high priority and it is more relevant to the core of the system than the other quality requirements with high priority (M1 and U2). We believe that handling pluggable device failure/connectivity is more important to the whole of the system than M1 and U2, and that handling this first would give a stronger starting point for later ADD iterations than M1 or U2.

8.2.3 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

Av3: Failure detection Gateway need to be able to autonomously detect failure of one of its connected motes and pluggable devices. This is achieved by making motes send heartbeats to their connected gateways. The gateways can then monitor their connected devices. The heartbeats contain a list of devices that are connected/operational at the moment the mote sends the heartbeat. Each gateway makes use of a **PluggableDeviceManager** component to monitor the devices. This component uses timers to keep track of how long it has been since a device has sent a heartbeat or occurred in a list of connected devices. Once a timer expires, this is treated as a failure.

A mote has failed when 3 consecutive heartbeats do not arrive within 1 second of their expected arrival time. A pluggable device has failed when it does not occur in a heartbeat of the mote in which it is expected to be in. This is detected within 2 seconds after the arrival of the heartbeat.

Av3: Automatic application deactivation and redundancy settings Applications should be automatically suspended when they can no longer operate due to failure of a pluggable device or mote and reactivated once the failure is resolved. Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices.

This problem is tackled by the `PluggableDeviceManager`. It stores the requirements for pluggable devices set by applications for all applications that use the gateway that the `PluggableDeviceManager` runs on. When it detects that an application can no longer operate due to failures, it will send a command to the `ApplicationManager` (via the `GatewayFacade`) to suspend that application. When the required devices are operational again, the `PluggableDeviceManager` detects this and sends a command to reactivate the application.

Applications are suspended within 1 minute after detecting the failure of an essential pluggable device.

Application are reactivated within 1 minute after the failure is resolved.

Av3: Notifications The infrastructure owner should be notified of any persistent pluggable device or mote failures. Customer organisations should be notified if one or more of their applications is suspended or reactivated. Applications using a failed pluggable device or any device on a failed mote should be notified.

The `NotificationHandler` was put in place to deal with notifications. Other components can use it to generate notifications for certain users in the system. The `NotificationHandler` will then insert information relevant to the notification in the database (message, status, date and time, source, ...), and use an external delivery service to deliver the notification to users. The used delivery medium is based on the user's preferences.

Since they are stored in the database, users can always view their notifications via their dashboard. However, this functionality is not expanded on in this decomposition yet.

Infrastructure owners are notified within 1 minute after detecting a mote outage lasting at least 10 seconds.

Infrastructure owners are notified within 1 minute after the detection of the unavailability of a pluggable device for 30 seconds.

Applications are notified of the failure of relevant pluggable devices within 10 seconds.

Alternatives considered

Av3: Failure detection An alternative would have been to move the `PluggableDeviceManager` component from gateways to the Online Service. This solution would make the gateways do less work, but would be very unscalable. The reason is that as the customer base (and thus the amount of devices) increases, the Online Service would need to keep track of huge amounts of devices. This would also flood the network to the Online Service with heartbeats.

Av3: Failure detection Another alternative for failure detection could have been the use of a Ping/Echo mechanism instead of Heartbeats. Pings could then be used to check if a device is currently operational. However, as a device could not be operational for a moment because of e.g. interference, timers would still be necessary to keep track of operational devices. We opted to use heartbeats, as this would reduce the amount of data sent over the network used by the motes, and as motes would have to do slightly more work to process each Ping request in order to generate a reply.

Av3: Notifications Reliable and quick delivery of notifications is crucial to the system in order to solve problems should things go wrong. Currently, the solution is to use a third party service for delivery of notifications. In the case that no external services are found satisfactory, or if this dependency on an external service is unwanted, it is possible to build an internal solution for this. For example, a `NotificationSender` component could make use of the `Factory pattern` for different message channels for different delivery methods (each with their own `sendNotification` method). This solution allows us to easily add new message channels in the future with little effort. The disadvantage of this is that an internal solution takes a lot more time to implement.

8.2.4 Instantiation and allocation of functionality

This section describes the components which instantiate our solutions described in the section above and how the components are deployed on physical nodes.

Decomposition Figure 8.1 shows the components resulting from the decomposition in this run.

Visual Paradigm Standard Edition(K.U.Leuven)

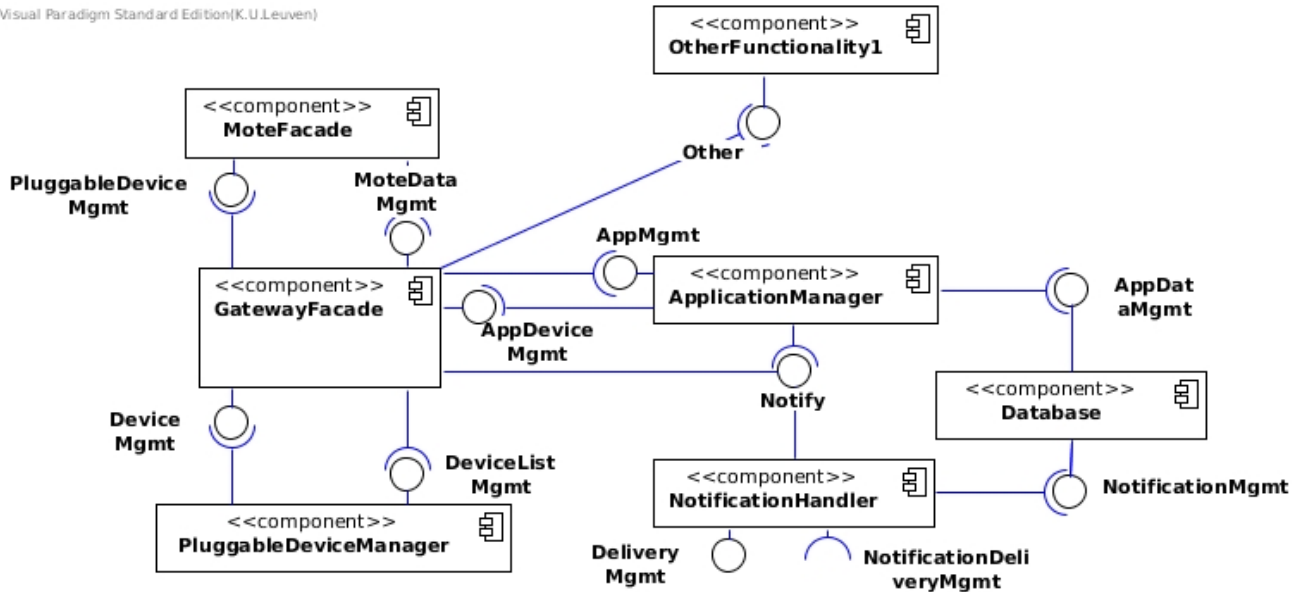


Figure 8.1: Component-and-connector diagram of this decomposition.

The responsibilities of the components are as follows:

ApplicationManager Responsible for activating/deactivating applications, setting pluggable device redundancy requirements on **PluggableDeviceManager** components, and using the **NotificationHandler** to send notifications to customer organisations. (Av3)

Database General database for data. For example, storage of data about notifications.

GatewayFacade Main component on the gateway that allows different components to work with each other. E.g. transmits heartbeats from motes to **PluggableDeviceManager**, transmits commands to shut down applications, triggers notifications to be generated, ...

MoteFacade Sends heartbeats to the **GatewayFacade**. Includes a list of connected pluggable devices in the heartbeats. (UC14)

NotificationHandler Responsible for generation, storage, and delivery of notifications based on users' preferred communication channel. (UC15)

PluggableDeviceManager Monitors connected/operational devices on a gateway. Sends notifications in case of hardware failure. Can send a command to disable or reactivate applications when necessary. (Av3)

Deployment Figure 8.2 shows the allocation of components to physical nodes.

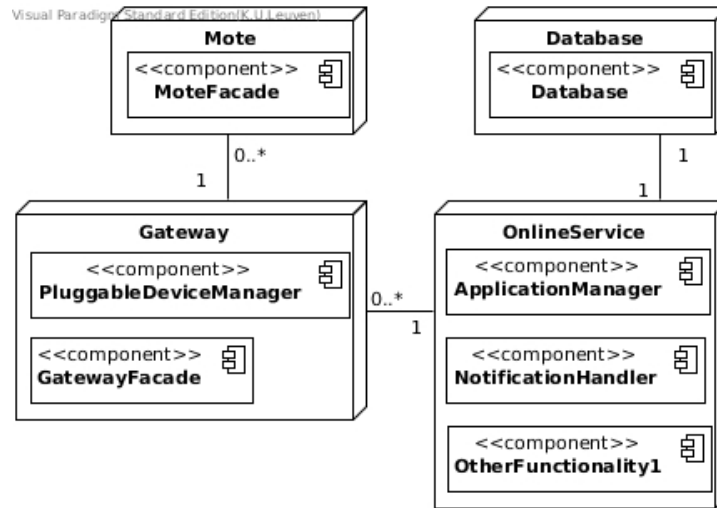


Figure 8.2: Deployment diagram of this decomposition.

8.2.5 Interfaces for child modules

This section describes the interfaces assigned to the components defined in the section above. Per interface, we list its methods by means of its syntax. The data types used in these interfaces are defined in the following section.

Each method shows which (part of a) quality attribute or use case caused a need for the method. However, this does not mean that a method is only to be used to satisfy that quality attribute or use case, it could be used for other causes not yet mentioned here.

ApplicationManager

- AppMgmt
 - `void deactivateApplicationInstance(int applicationInstanceID)`
 - * Effect: Deactivates a running instance of an application.
 - * Created for: UC18
 - Av3: automatic suspension/reactivation of applications.
 - `void activateApplicationInstance(int applicationInstanceID)`
 - * Effect: Activates a new instance of an application.
 - * Created for: UC18
 - Av3: automatic suspension/reactivation of applications.

Database

- NotificationMgmt
 - `int storeNotification(NotificationData data)`
 - * Effect: Stores a new notification entry in the database. Returns the id of the new notification.
 - * Created for: UC15
 - Av3: notifications
 - `void updateNotification(NotificationData data)`
 - * Effect: Updates an existing notification (e.g. change status to "sent").
 - * Created for: UC15
 - `int lookupNotificationChannelForUser(int userID)`
 - * Effect: Returns the type of communication channel a user prefers. Different communication channels are mapped to integers.

- * Created for: UC15
- AppMgmt
 - void updateApplication(ApplicationData data)
 - * Effect: Updates an application in the database (e.g. change state to 'inactive').
 - * Created for: UC18
 - Av3: automatic suspension/reactivation of applications.
 - void updateSubscription(SubscriptionData data)
 - * Effect: Updates a subscription in the database (e.g. change state to 'disabled').
 - * Created for: UC18

GatewayFacade

- Heartbeat
 - void heartbeat(MoteInfo moteInfo, List<Tuple<PluggableDeviceID, PluggableDeviceType>> pds)
 - * Effect: Sends a heartbeat from the mote to the gateway, including a list of the PluggableDevices and their DeviceTypes (i.e. those currently plugged into the mote)
 - * Created for: Given constraint
 - UC14
 - Av3: failure detection
- DeviceData
 - void rcvData(PluggableDeviceID pID, DeviceData data)
 - * Effect: Provides pluggable device data to the gateway (Initiated by the device).
 - * Created for: Given constraint, unused at the moment.
 - void rcvDataCallback(PluggableDeviceID pID, DeviceData data, int requestID)
 - * Effect: Provides device data to the gateway (Callback of getDataAsync).
 - * Created for: Given constraint, unused at the moment.
- DeviceManagement
 - void pluggableDevicePluggedIn(MoteInfo mInfo, PluggableDeviceID pID, PluggableDeviceType type)
 - * Effect: Notifies the gateway that a new PluggableDevice of the given type is connected to the mote.
 - * Created for: Given constraint, unused at the moment.
 - void pluggableDeviceRemoved(PluggableDeviceID pID)
 - * Effect: Notifies the gateway that a PluggableDevice is removed.
 - * Created for: Given constraint, unused at the moment.
 - void pluggableDevicePersistentFailure(int deviceID)
 - * Effect: Lets the gateway know that a timer for pluggable device or mote has expired. This will generate a notification for an infrastructure owner.
 - * Created for: Av3: The infrastructure owner should be notified of any persistent pluggable device or mote failures.
 - List<DeviceInfo> getConnectedDevices()
 - * Effect: Returns a list of information about devices that are connected to the gateway.
 - * Created for: UC18
 - * Tradeoff: send PluggableDeviceID instead of DeviceInfo. If you send DeviceInfo, then Application-Manager does not have to fetch this info. If you send PluggableDeviceID's, then less bandwidth is used and the Gateways do less work.
 - void deactivateApplicationInstance(int applicationInstanceID)
 - * Effect: Deactivates a certain application. This could happen when mandatory pluggable devices for the application are missing.
 - * Created for: Av3: automatic suspension/reactivation of applications.

- `void reactivateApplicationInstance(int applicationInstanceID)`
 - * Effect: Reactivate an application instance. This could happen automatically after a broken sensor has been replaced.
 - * Created for: Av3: automatic suspension/reactivation of applications.
- **AppDeviceMgmt**
 - `void setPluggableDevicesRequirements(int applicationID, List<PluggableDeviceInfo> devices)`
 - * Effect: Sets an application's requirements for pluggable devices.
 - * Created for: Av3: Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices.
 - `bool areEssentialDevicesOperational(int applicationID)`
 - * Effect: Returns true if all essential devices for the application with id "applicationID" are operational.
 - * Created for: UC18

MoteFacade

- **PluggableDeviceMgmt**
 - `List<DeviceInfo> getConnectedDevices()`
 - * Effect: Returns a list of information about devices that are connected to the mote.
 - * Created for: UC18
 - * Tradeoff: send PluggableDeviceID instead of DeviceInfo. If you send DeviceInfo, then Application-Manager does not have to fetch this info. If you send PluggableDeviceID's, then less bandwidth is used and the Gateways do less work.

NotificationHandler

- **Notify**
 - `void notify(int userID, String message)`
 - * Effect: Stores a new notification in the system and causes it to be sent to a user.
 - * Created for: UC14
 - Av3: notifications
- **DeliveryMgmt**
 - `void sendAcknowledgement(int notificationID)`
 - * Effect: Sends an acknowledgement to the system for a certain notification.
 - * Created for: UC15

External notification delivery service

- **NotificationDeliveryMgmt**
 - `void notify(JSONObject data)`
 - * Effect: Delivers a notification to an end user using a specific delivery service.
 - * Created for: UC15

PluggableDeviceManager

- **DeviceListMgmt**
 - `void sendHeartbeat(int moteID, List<PluggableDeviceInfo> devices)`
 - * Effect: Sends a heartbeat from a mote to check/update timers for operational devices.
 - * Created for: UC14
 - Av3: failure detection
 - `void setPluggableDevicesRequirements(int applicationID, List<PluggableDeviceInfo> devices)`

- * Effect: Sets an application's requirements for pluggable devices.
- * Created for: Av3: Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices.
- **bool areEssentialDevicesOperational(int applicationID)**
 - * Effect: Returns true if all essential devices for the application with id "applicationID" are operational.
 - * Created for: UC18

8.2.6 Data type definitions

This section defines the data types used in the interface descriptions above.

PluggableDeviceInfo contains information about a pluggable device (device id, power status, data update rate, ...)

NotificationData contains data about a notification (message text, recipient, communication channel, date, status, source, ...).

ApplicationData contains data about an application instance (instance id, running status, ...)

SubscriptionData contains data about a subscription (subscription id, subscription status, subscription period, ...).

8.2.7 Verify and refine

The selected architectural drivers have been handled completely in this decomposition. This section describes per component which (parts of) the remaining requirements it is responsible for. If requirements are split in multiple parts, this is indicated by the addition of a letter (or number, depending on the structure of the requirement) after their title.

ApplicationManager

- *Av2*: Application failure
Prevention: a, b
Detection: a, b, c
Resolution: a, b, c
- *P1*: Large number of users: c
- *M1*: Integrate new sensor or actuator manufacturer: 1.c, 2.a
- *M2*: Big data analytics on pluggable data and/or application usage data: d, e
- *U1*: Application updates: a, b, c, d
- *U2*: Easy Installation: e
- *UC12*: Perform actuation command
- *UC17*: Activate an application: 3, 4

Database

- None

GatewayFacade

- *Av1*: Communication between SIoTIP gateway and Online Service
Resolution: b, c, d
- *M1*: Integrate new sensor or actuator manufacturer: 1.a, 2.b
- *U2*: Easy Installation: a, c, d
- *UC11*: Send pluggable device data: 1

MoteFacade

- *M1*: Integrate new sensor or actuator manufacturer: 1.a, 2.b
- *U2*: Easy Installation: b, c, d
- *UC04*: Install mote: 1, 2
- *UC05*: Uninstall mote: 1
- *UC06*: Insert a pluggable device into a mote: 2
- *UC07*: Remove a pluggable device from its mote: 2
- *UC11*: Send pluggable device data: 1

NotificationHandler

- *UC16*: Consult notification message: 5
- *UC17*: Activate an application: 5, 6

OtherFunctionality1

- *Av1*: Communication between SIIoTIP gateway and Online Service
Detection: a, b, c, d Resolution: a
- *P1*: Large number of users: a
- *P2*: Requests to the pluggable data database
- *M1*: Integrate new sensor or actuator manufacturer: 1.d
- *M2*: Big data analytics on pluggable data and/or application usage data: a
- *U2*: Easy Installation: e
- *UC01*: Register a customer organisation
- *UC02*: Register an end-user
- *UC03*: Unregister an end user
- *UC04*: Install mote: 3
- *UC05*: Uninstall mote: 2.b
- *UC06*: Insert a pluggable device into a mote: 3: topology part; alternative 3a.1.b
- *UC07*: Remove a pluggable device from its mote: 3.b
- *UC08*: Initialise a pluggable device: 1, 2, 4
- *UC09*: Configure pluggable device access rights
- *UC10*: Consult and configure the topology
- *UC11*: Send pluggable device data: 3
- *UC13*: Configure pluggable device
- *UC16*: Consult notification message: 1, 2, 3, 4
- *UC17*: Activate an application: 1, 2
- *UC19*: Subscribe to application
- *UC20*: Unsubscribe from application
- *UC21*: Send invoice
- *UC22*: Upload an application
- *UC23*: Consult application statistics
- *UC24*: Consult historical data
- *UC25*: Access topology and available devices
- *UC26*: Send application command or message to external front-end
- *UC27*: Receive application command or message to external front-end
- *UC28*: Log in
- *UC29*: Log out

PluggableDeviceManager

- *U2*: Easy Installation: c, d
- *UC04*: Install mote: 4
- *UC05*: Uninstall mote: 2

- *UC06*: Insert a pluggable device into a mote: 3: uninitialised part; alternative 3a.1 3a.2 3a.4; 4
- *UC07*: Remove a pluggable device from its mote: 3.a, 3.c
- *UC08*: Initialise a pluggable device: 3
- *UC11*: Send pluggable device data: 2, 3a

8.3 Decomposition 2: OtherFunctionality1 (M1, P2, UC11)

8.3.1 Module to decompose

In this run we decompose OtherFunctionality1.

8.3.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *M1*: Integrate new sensor or actuator manufacturer
- *P2*: Requests to the pluggable data database

The related functional drivers are:

- *UC11*: Send pluggable device data (P2)
This use case stores pluggable device data in the pluggable device data storage. This could be a sensor reading or an actuator status.

Rationale We chose M1 as it was one of the remaining quality attributes with high priority. M1's focus on easily introducing new types of devices to the system is very important because of the fast growing market for IoT and development of applications for IoT. Thus, we want to handle this quality attribute before U2 (the other remaining attribute with high priority), as we presume that customer organisations are more interested in using new devices than the effort it takes for infrastructure owners to install the devices.

We also chose P2 because it is strongly related to M1; the whole data flow from devices to storage/applications needs to exist before modifications can even be made. This combination of M1 and P2 would force us to handle processing and storage of data while making the involved components as simple as possible to modify.

8.3.3 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

M1: Data conversion With new types of devices, the pluggable data processing subsystem should be extended with relevant data conversions, e.g. converting temperature in degrees Fahrenheit to degrees Celsius.

The `DeviceDataConverter` is put in place to handle the task of converting pluggable device data to data of a different type in the system. This component can easily be modified for new types of data simply by adding a new conversion method for the new.

M1: Usage of new data by applications The available applications in the system can be updated to use any new pluggable devices.

This is made possible by the `RequestData` interface provided by `DeviceDataScheduler`. Data of the new type of device can be requested in the same way as for older devices: by using the device's unique id. The application manager can get pluggable device data from the `PluggableDeviceDB` and return this data to applications in the `DeviceData` datatype. This datatype can easily be updated for new types of pluggable devices.

P2: Scheduling The pluggable data processing subsystem needs to be able to run in normal or overload mode, depending on whether or not the system can process requests within the deadlines given in the quality requirement. Also, a mechanism should be in place to avoid starvation of any type of request.

The `DeviceDataScheduler` is used to deal with this problem. It is responsible for scheduling requests that wish to interact with the `PluggableDeviceDB`. In normal mode, the system processes incoming requests in a FIFO order. In overload mode, the requests are given a priority based on what the request is for and what the source of the request is. The requests are then not simply processed in an order based on their priorities, but an aging technique is to be used such that starvation will be avoided. Thus, in overload mode, requests are processed in an order based on a combination of the priorities of the requests and the age of the requests.

P2: Pluggable data separation The processing of (large amounts of) requests concerning pluggable data has no impact on requests concerning other data, e.g. available applications. In order to satisfy this constraint, all data directly related to pluggable data has been separated into the `PluggableDeviceDB`. All requests concerning pluggable data will be handled by this new component. `PluggableDeviceDB` will run on a node different from the node that the `Database` component runs on. This way requests concerning pluggable will have no impact on requests concerning other data.

M1: Handling new types of pluggable devices The new types of sensor or actuator data should be transmitted, processed and stored, and should be made available to applications. The infrastructure managers must be able to initialize the new type of pluggable device, configure access rights for these devices, and view detailed information about the new type of pluggable device.

The components created thus far have been created with high cohesion in mind so that updating them for new devices would be relatively straightforward. In order for this constraint to be satisfied, changes have to be made to the following elements:

- *PluggableDeviceFacade*: This component needs to be updated so that the new type of device can be initialised and configured, and thus so that the device's data can be sent to the system.
- *DeviceData*: Depending on how this data type is implemented, it might need an update in order for it to represent possible new data types (for example Temperature Filipcikova) and for the new data types to be serialized.
- *PluggableDeviceDB*: The database needs to be updated so that information can be retrieved about the new types of sensors and the new types of data. Data related to the displaying of sensor data will also need to be updated.
- *PluggableDeviceConverter*: see above.

8.3.4 Instantiation and allocation of functionality

This section describes the new components which instantiate our solutions described in the section above and how components are deployed on physical nodes.

Unless stated otherwise the responsibilities assigned in the first decomposition are unchanged.

Decomposition Figure 8.3 shows the components resulting from the decomposition in this run. The responsibilities of the components are as follows:

DeviceDataConverter The `DeviceDataConverter` is responsible for converting pluggable device data in the data processing subsystem. (M1)

DeviceDataScheduler Responsible for scheduling incoming read and write requests for pluggable device data. Monitors throughput of requests and switches between normal and overload mode when appropriate. Avoids starvation of any type of request. (P2)

PluggableDeviceDB Database dedicated to pluggable device data. (P2)

PluggableDeviceFacade Responsible for sending pluggable device data to `MoteFacade`. Needs to be initialised in order for the data to be used/stored. (UC11)

Deployment Figure 8.4 shows the allocation of components to physical nodes.

8.3.5 Interfaces for child modules

This section describes the interfaces assigned to the components defined in the section above. Per interface, we list its methods by means of its syntax. The data types used in these interfaces are defined in the following section.

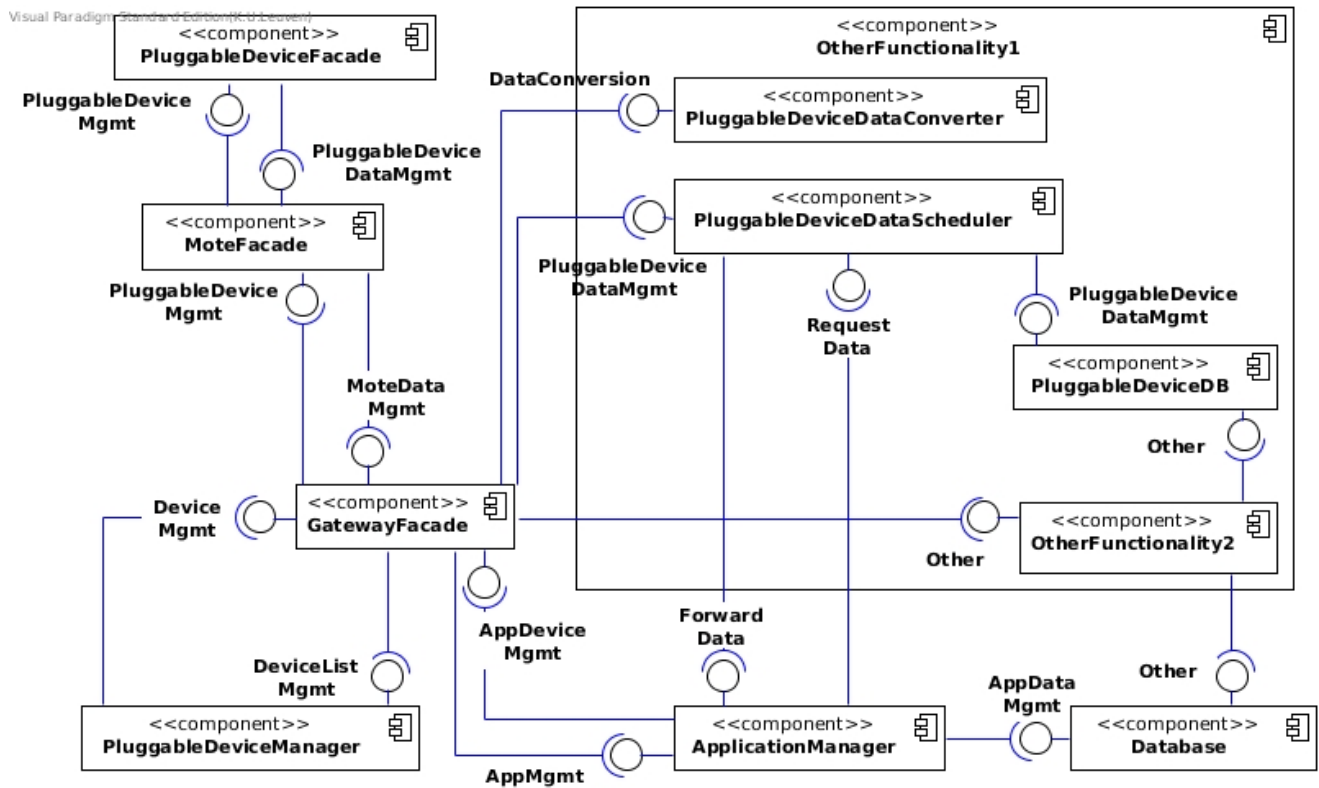


Figure 8.3: Component-and-connector diagram of this decomposition.

Each method shows which (part of a) quality attribute or use case caused a need for the method. However, this does not mean that a method is only to be used to satisfy that quality attribute or use case, it could be used for other causes not yet mentioned here.

The interfaces and methods defined here are to be seen as an extension of the interfaces defined in previous sections, unless explicitly stated otherwise.

ApplicationManager

- ForwardData
 - void rcvData(PluggableDeviceID pID, DeviceData data)
 - * Effect: Sends pluggable device data to an application that wants to use it
 - * Created for: UC11: system relays data to applications

GatewayFacade

- DeviceData, last defined in section 8.2.5
 - void rcvData(PluggableDeviceID pID, DeviceData data)
 - * Now (also) used by: UC11
 - P2: storing new pluggable data
 - void rcvDataCallback(PluggableDeviceID pID, DeviceData data, int requestID)
 - * Now (also) used by: UC11
 - P2: storing new pluggable data
- DeviceManagement, last defined in section 8.2.5

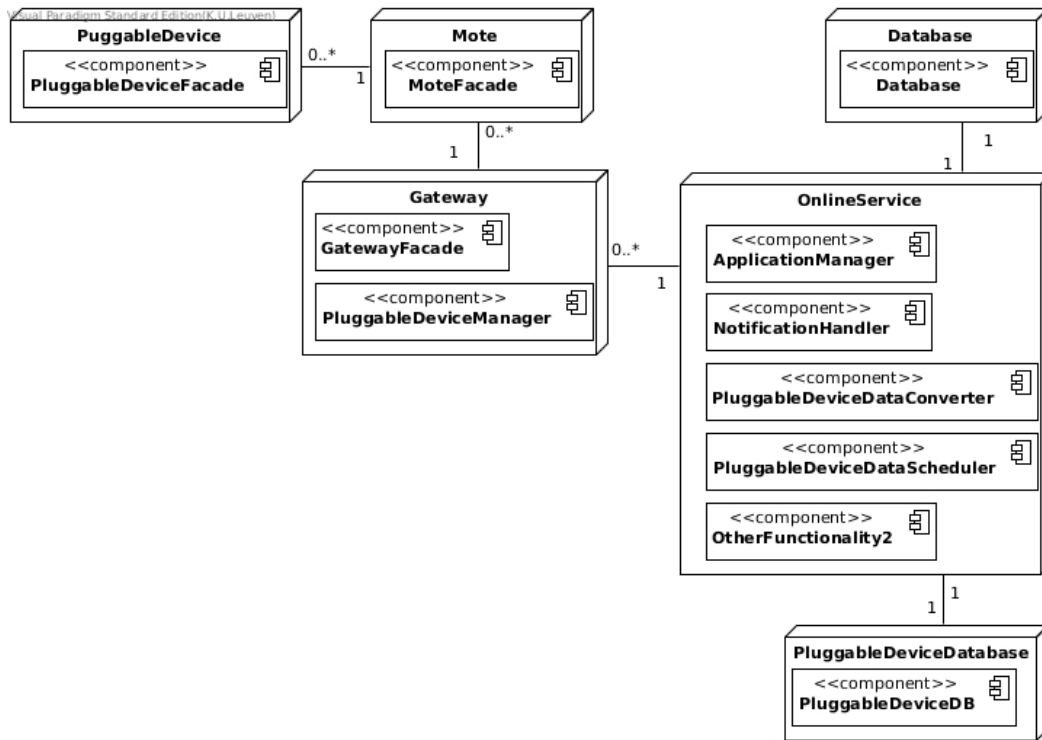


Figure 8.4: Deployment diagram of this decomposition.

- void setConfig(PluggableDeviceID pID, Map<String,String> config)
 - * Effect: Set the given configuration parameters of a PluggableDevice to the given values. Setting unknown parameters on a PluggableDevice has no effect.
 - * Created for: UC11: pluggable device needs to be initialised
 - M1: pluggable device must be able to be initialised

MoteFacade

- DeviceData
 - void rcvData(PluggableDeviceID pID, DeviceData data)
 - * Effect: Propagates pluggable device data to the connected gateway by call rcvData on the gateway. (Initiated by the device).
 - * Created for: UC11
 - P2: storing new pluggable data
 - void rcvDataCallback(PluggableDeviceID pID, DeviceData data, int requestID)
 - * Effect: Propagates pluggable device data to the connected gateway by calling rcvData on the gateway. (Callback of getDataAsync).
 - * Created for: UC11
 - P2: storing new pluggable data
- PluggableDeviceMgmt, last defined in section 8.2.5
 - void setConfig(PluggableDeviceID pID, Map<String,String> config)
 - * Effect: Set the given configuration parameters of a PluggableDevice to the given values. Setting unknown parameters on a PluggableDevice has no effect.
 - * Created for: UC11: pluggable device needs to be initialised
 - M1: pluggable device must be able to be initialised

PluggableDeviceFacade

- Config
 - `Map<String,String> getConfig()`
 - * Effect: Returns the current configuration of a PluggableDevice as a parameter-value map.
 - * Created for: Given constraint, unused at the moment.
 - `boolean setConfig(Map<String,String> config)`
 - * Effect: Set the given configuration parameters of the PluggableDevice to the given values. Setting unknown parameters on a PluggableDevice (e.g., noise threshold 3 on a light sensor) has no effect.
 - * Created for: Given constraint
 - UC11: pluggable device needs to be initialised
 - M1: pluggable device must be able to be initialised
- RequestData
 - `DeviceData getData()`
 - * Effect: Synchronously retrieve the device data of a device.
 - * Created for: Given constraint, unused at the moment.
 - `void getDataAsync(int requestID)`
 - * Effect: Asynchronously retrieve the device data of a device (by calling `rcvDataCallback`).
 - * Created for: Given constraint, unused at the moment.
- Actuate
 - `void sendActuationCommand(String commandName)`
 - * Effect:
 - * Created for: Given constraint, unused at the moment.

PluggableDeviceManager

- DeviceListMgmt, last defined in section 8.2.5
 - `bool isDeviceInitialised(PluggableDeviceID pID)`
 - * Effect: Returns true if the device with id "pID" has been initialized.
 - * Created for: UC11: pluggable device needs to be initialised
 - M1: pluggable device must be able to be initialised
 - * : TODO: need this check? is 'initialized' status stored in DB or on gateways? or both?

DeviceDataConverter

- DataConversion
 - `DeviceData convert(PluggableDeviceID pID, DeviceData data, string type)`
 - * Effect: Converts pluggable device data into other pluggable device data that contains the same information in a different type
 - * Created for: M1: data processing subsystem should be extended with relevant data conversions

DeviceDataScheduler

- RequestData
 - `List<DeviceData> getData(int applicationID, PluggableDeviceID pID, DateTime from, DateTime to)`
 - * Effect: Requests data from a specific device in a certain time period
 - * Created for: P2: requests from applications
- DeviceDataMgmt
 - `void rcvData(PluggableDeviceID pID, DeviceData data)`

- * Effect: Sends pluggable device data to the scheduler to be processed.
- * Created for: UC11
- P2: storing new pluggable data

PluggableDeviceDB

- DeviceDataMgmt
 - void rcvData(PluggableDeviceID pID, DeviceData data)
 - * Effect: Sends pluggable device data to the DB to be stored.
 - * Created for: UC11
 - P2: storing new pluggable data
 - List<DeviceData> getData(PluggableDeviceID pID, DateTime from, DateTime to)
 - * Effect: Returns data from a specific device in a certain time period.
 - * Created for: P2: lookup queries
 - List<int> getApplicationsForDevice(PluggableDeviceID pID)
 - * Effect: Returns a list of applications that can use the device with id "pID".
 - * Created for: UC11: looks up the list of applications that use the pluggable device

8.3.6 Data type definitions

This section defines new data types that are used in the interface descriptions above.

DeviceData contains data from a pluggable device at a certain point in time (value, type, date) (e.g. a sensor reading, an actuator status)

PluggableDeviceSettings contains settings for a pluggable device (power status, data update rate, ...)

DateTime Represents an instant in time, typically expressed as a date and time of day.

8.3.7 Verify and refine

The selected architectural drivers have been handled completely in this decomposition. This section describes per component which (parts of) the remaining requirements it is responsible for. If requirements are split in multiple parts, this is indicated by the addition of a letter (or number, depending on the structure of the requirement) after their title.

ApplicationManager

- *Av2*: Application failure
 - Prevention: a, b
 - Detection: a, b, c
 - Resolution: a, b, c
- *P1*: Large number of users: c
- *M2*: Big data analytics on pluggable data and/or application usage data: d, e
- *U1*: Application updates: a, b, c, d
- *U2*: Easy Installation: e
- *UC12*: Perform actuation command
- *UC17*: Activate an application: 3, 4

Database

- None

GatewayFacade

- *Av1*: Communication between SIoTIP gateway and Online Service
Resolution: b, c, d
- *U2*: Easy Installation: a, c, d

MoteFacade

- *U2*: Easy Installation: b, c, d
- *UC04*: Install mote: 1, 2
- *UC05*: Uninstall mote: 1
- *UC06*: Insert a pluggable device into a mote: 2
- *UC07*: Remove a pluggable device from its mote: 2

NotificationHandler

- *UC16*: Consult notification message: 5
- *UC17*: Activate an application: 5, 6

OtherFunctionality2

- *Av1*: Communication between SIoTIP gateway and Online Service
Detection: a, b, c, d Resolution: a
- *P1*: Large number of users: a
- *M2*: Big data analytics on pluggable data and/or application usage data: a
- *U2*: Easy Installation: e
- *UC01*: Register a customer organisation
- *UC02*: Register an end-user
- *UC03*: Unregister an end user
- *UC04*: Install mote: 3
- *UC05*: Uninstall mote: 2.b
- *UC06*: Insert a pluggable device into a mote: 3: topology part; alternative 3a.1.b
- *UC07*: Remove a pluggable device from its mote: 3.b
- *UC08*: Initialise a pluggable device: 1, 2, 4
- *UC09*: Configure pluggable device access rights
- *UC10*: Consult and configure the topology
- *UC13*: Configure pluggable device
- *UC16*: Consult notification message: 1, 2, 3, 4
- *UC17*: Activate an application: 1, 2
- *UC19*: Subscribe to application
- *UC20*: Unsubscribe from application
- *UC21*: Send invoice
- *UC22*: Upload an application
- *UC23*: Consult application statistics
- *UC24*: Consult historical data
- *UC25*: Access topology and available devices
- *UC26*: Send application command or message to external front-end
- *UC27*: Receive application command or message to external front-end
- *UC28*: Log in
- *UC29*: Log out

PluggableDeviceDB

- *M2*: Big data analytics on pluggable data and/or application usage data: b

PluggableDeviceFacade

- *U2*: Easy Installation: d

PluggableDeviceManager

- *U2*: Easy Installation: c, d
- *UC04*: Install mote: 4
- *UC05*: Uninstall mote: 2
- *UC06*: Insert a pluggable device into a mote: 3: uninitialised part; alternative 3a.1 3a.2 3a.4; 4
- *UC07*: Remove a pluggable device from its mote: 3.a, 3.c
- *UC08*: Initialise a pluggable device: 3,

DeviceDataScheduler

- *P1*: Large number of users: b
- *M2*: Big data analytics on pluggable data and/or application usage data: b, c

8.4 Decomposition X: DRIVERS (Elements/Subsystem to decompose/expand)

8.4.1 Elements/Subsystem to decompose/expand

In this run we decompose/expand ...

8.4.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *QA*: Name of QA

The related functional drivers are:

- *UCX*: Name of UC
Short description of the UC.

8.4.3 Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.

QA: Problem title Short description of the problem.
Solution for the problem.

8.4.4 Instantiation and allocation of functionality

This section describes the new components which instantiate our solutions described in the section above and how components are deployed on physical nodes.

Unless stated otherwise the responsibilities assigned in the first decomposition are unchanged.

Decomposition Figure 8.6 shows the components resulting from the decomposition in this run.

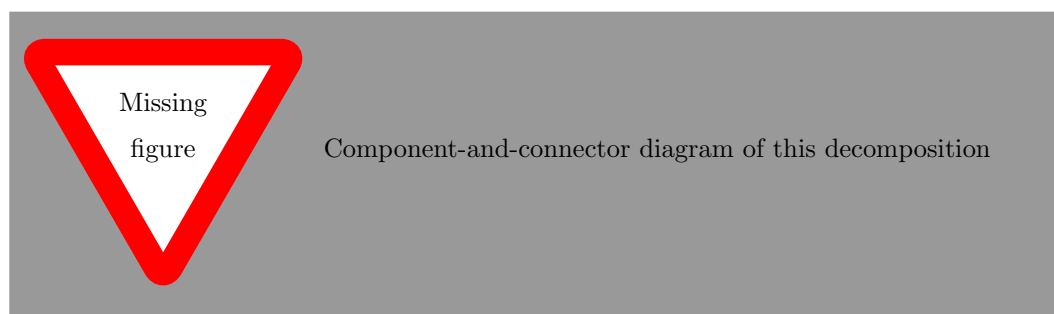


Figure 8.5: Component-and-connector diagram of this decomposition.

The responsibilities of the components are as follows:

Component Short description of its responsibilities. (Relevant QA or UC)

Deployment Figure 8.6 shows the allocation of components to physical nodes.

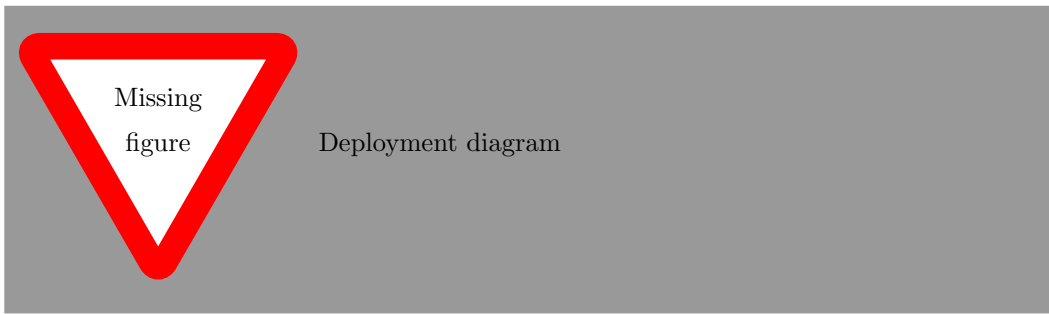


Figure 8.6: Deployment diagram of this decomposition.

8.4.5 Interfaces for child modules

This section describes the interfaces assigned to the components defined in the section above. Per interface, we list its methods by means of its syntax. The data types used in these interfaces are defined in the following section.

Each method shows which (part of a) quality attribute or use case caused a need for the method. However, this does not mean that a method is only to be used to satisfy that quality attribute or use case, it could be used for other causes not yet mentioned here.

The interfaces and methods defined here are to be seen as an extension of the interfaces defined in previous sections, unless explicitly stated otherwise.

Component

- InterfaceName
 - `void methodName(.. parameters ..)`
 - * Effect: Short description of the method
 - * Created for: Reason for the addition of this method.

8.4.6 Data type definitions

This section defines new data types that are used in the interface descriptions above.

DataType Description of data type