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 Architectural Decisions **5**
- 1.1 Av1: Communication between SIoTIP gateway and Online Service . . . 5
- 1.2 Av2: Application failure . . . 7
- 1.3 Av3: Pluggable device or mote failure . . . 8
- 1.4 M1: Integrate new sensor or actuator manufacturer . . . 9
- 1.5 P1: Large number of users . . . 10
- 1.6 P2: Requests to the pluggable data database . . . 11
- 1.7 U2: Easy installation . . . 12
- 1.8 Other decisions . . . 13
  - 1.8.1 Authentication . . . 13
  - 1.8.2 Notifications . . . 13
  - 1.8.3 Application execution subsystem . . . 14
- 1.9 Discussion . . . 14

2 Client-server view (UML Component diagram) **16**
- 2.1 Context diagram . . . 17
- 2.2 Primary diagram . . . 18

3 Decomposition view (UML Component diagram) **20**

4 Deployment view (UML Deployment diagram) **25**
- 4.1 Context diagram . . . 26
- 4.2 Primary diagram . . . 27

5 Scenarios **28**
- 5.1 Scenarios . . . 28

6 Element Catalog and Datatypes **38**

7 Catalog **39**
- 7.1 Components . . . 39
  - 7.1.1 AccessRightsManager . . . 39
  - 7.1.2 ApplicationContainer . . . 39
  - 7.1.3 ApplicationContainerManager . . . 39
  - 7.1.4 ApplicationContainerMonitor . . . 39
  - 7.1.5 ApplicationExecutionSubsystemMonitor . . . 39
  - 7.1.6 ApplicationFrontEndClient . . . 40
  - 7.1.7 ApplicationFrontEndFacade . . . 40
  - 7.1.8 ApplicationInstance . . . 40
  - 7.1.9 ApplicationManagementLogic . . . 40
  - 7.1.10 ApplicationManager . . . 40
  - 7.1.11 ApplicationProviderClient . . . 40
  - 7.1.12 ApplicationProviderFacade . . . 41
  - 7.1.13 AuthenticationManager . . . 41
  - 7.1.14 ContainerLogic . . . 41
  - 7.1.15 CustomerOgranisationClient . . . 41
  - 7.1.16 CustomerOrganisationFacade . . . 41
  - 7.1.17 DeviceCommandConstructor . . . 41
  - 7.1.18 DeviceDataConverter . . . 42
  - 7.1.19 DeviceDataScheduler . . . 42

7.1.20 DeviceDB . . . 42
7.1.21 DeviceManager . . . 42
7.1.22 Gateway . . . 42
7.1.23 GatewayCommunicationHandler . . . 43
7.1.24 GatewayCommunicationMonitor . . . 43
7.1.25 GatewayMonitor . . . 43
7.1.26 GWCommunicationLogic . . . 43
7.1.27 InfrastructreOwnerClient . . . 44
7.1.28 InfrastructureOwnerFacade . . . 44
7.1.29 InfrastructureOwnerManager . . . 44
7.1.30 InvoiceManager . . . 44
7.1.31 Mote . . . 44
7.1.32 NotificationHandler . . . 44
7.1.33 OnlineServiceCommunicationHandler . . . 45
7.1.34 OnlineServiceCommunicationMonitor . . . 45
7.1.35 OnlineServiceMonitor . . . 45
7.1.36 OSCommunicationLogic . . . 45
7.1.37 OtherDataDB . . . 46
7.1.38 PluggableDevice . . . 46
7.1.39 PluggableDeviceDataDB . . . 46
7.1.40 RegisteredUserFacade . . . 46
7.1.41 RequestStore . . . 46
7.1.42 SessionDB . . . 47
7.1.43 SubscriptionManager . . . 47
7.1.44 SystemAdministratorFacade . . . 47
7.1.45 ThirdPartyInvoicingService . . . 47
7.1.46 ThirdPartyNotificationDeliveryService . . . 47
7.1.47 TopologyManager . . . 47
7.1.48 UnregisteredUserClient . . . 48
7.1.49 UnregisteredUserFacade . . . 48
7.1.50 UserManager . . . 48
7.2 Interfaces . . . 48
7.2.1 AccessRights . . . 48
7.2.2 AccessRightsMgmt . . . 48
7.2.3 Actuate . . . 49
7.2.4 Actuate . . . 49
7.2.5 AppDeviceData . . . 49
7.2.6 AppDeviceMgmt . . . 49
7.2.7 AppInstanceMgmt . . . 50
7.2.8 Applications . . . 51
7.2.9 ApplicationTesting . . . 51
7.2.10 AppMessages . . . 51
7.2.11 AppMonitoring . . . 52
7.2.12 AppStatus . . . 52
7.2.13 Authentication . . . 52
7.2.14 Authentication . . . 53
7.2.15 Commands . . . 53
7.2.16 Config . . . 53
7.2.17 Config . . . 53
7.2.18 DataConversion . . . 54
7.2.19 DBAccessRightsMgmt . . . 54
7.2.20 DBAppDeviceMgmt . . . 54
7.2.21 DBAppMgmt . . . 54
7.2.22 DBDeviceData . . . 56

7.2.23 DBDeviceMgmt . . . 56
7.2.24 DBInvoiceMgmt . . . 57
7.2.25 DBIODeviceMgmt . . . 58
7.2.26 DBIOMgmt . . . 58
7.2.27 DBNotificationMgmt . . . 58
7.2.28 DBSubscriptionMgmt . . . 59
7.2.29 DBTopologyMgmt . . . 59
7.2.30 DBUserMgmt . . . 60
7.2.31 DeliveryMgmt . . . 61
7.2.32 DeliveryMgmt . . . 61
7.2.33 DeviceCommands . . . 62
7.2.34 DeviceCommands . . . 62
7.2.35 DeviceData . . . 62
7.2.36 DeviceData . . . 62
7.2.37 DeviceMgmt . . . 63
7.2.38 ForwardData . . . 63
7.2.39 FrontEndAppRequests . . . 63
7.2.40 GatewayUpdates . . . 64
7.2.41 GWAppInstanceMgmt . . . 65
7.2.42 GWCommunicationMonitoring . . . 65
7.2.43 Heartbeat . . . 65
7.2.44 InvoiceDeliveryMgmt . . . 66
7.2.45 InvoiceMgmt . . . 66
7.2.46 IOMgmt . . . 66
7.2.47 Monitoring . . . 66
7.2.48 NotificationDeliveryMgmt . . . 66
7.2.49 Notifications . . . 67
7.2.50 Notifications . . . 67
7.2.51 Notifiy . . . 67
7.2.52 OSCommunicationMonitoring . . . 67
7.2.53 OSMonitoring . . . 68
7.2.54 OSUpdates . . . 68
7.2.55 Registration . . . 68
7.2.56 RequestData . . . 69
7.2.57 RequestData . . . 69
7.2.58 RequestData . . . 69
7.2.59 Requests . . . 69
7.2.60 RoleMgmt . . . 70
7.2.61 Sessions . . . 70
7.2.62 SubscriptionMgmt . . . 70
7.2.63 SubscriptionMgmt . . . 71
7.2.64 TopologyMgmt . . . 72
7.2.65 TopologyMgmt . . . 72
7.2.66 UserMgmt . . . 73
7.2.67 UserMgmt . . . 74
7.3 Exceptions . . . 74
7.4 Data types . . . 75

A Attribute-driven design documentation **77**
A.1 Introduction . . . 77
A.2 Adapted ADD process . . . 77
A.3 Decomposition 1: Av3, UC14, UC15, UC18 (SIoTIP System) . . . 78
A.4 Decomposition 2: M1, P2, UC11 (OtherFunctionality1) . . . 84
A.5 Decomposition 3: U2, UC4, UC6, UC9, UC10, UC17, UC19 . . . 90

A.6 Decomposition 4: Av2, UC12, UC25, UC26, UC27 (application execution subsystem) 94
A.7 Decomposition 5: Av1 (Gateway - Online Service communication subsystem) 97
A.8 Decomposition 6: P1, UC1, UC2, UC3, UC5, UC7, UC8, UC16, UC20 100
A.9 Decomposition 7: UC28, UC29 (authentication subsystem) 102
A.10 Decomposition 8: UC22, UC23 (application upload and statistics) 104
A.11 Decomposition 9: UC21 (invoicing subsystem) 105

1. Architectural Decisions

1.1 Av1: Communication between SIoTIP gateway and Online Service

Key Decisions

- `OSCommunicationLogic` handles communication from Gateways to the Online Service by remote method invocation.
- `OnlineServiceMonitor` monitors the Gateway's connectivity to the Online Service.
- `RequestStore` stores all pluggable device data and application commands on Gateways.
- `OnlineServiceCommunicationMonitor` monitors the internal communication components of Gateways.
- `GWCommunicationLogic` handles communication from the Online Service to Gateways by remote method invocation.
- `GatewayMonitor` monitors the connectivity of the Online Service to gateways.
- `GatewayCommunicationMonitor` monitors the internal communication components of Gateways.

Employed tactics and patterns: heartbeats, ping/echo, buffering

Rationale

To handle all communication going from Gateways to the Online Service, the `OnlineServiceCommunicationHandler` was introduced. It isolates all communication-related concerns along with `GatewayCommunicationHandler` on the Online Service. It forwards requests from one party to the other in a Remote Procedure Call-like manner and transmits results and possible exceptions.
Av1 requres autonomous detection of failures of individual internal communication components in Gateways. This is why we added the `OnlineServiceCommunicationMonitor` which monitors the communication handler. If the component is to fail, the monitor will detect it using a ping/echo mechanism and try to restart the component. If the failure persists, the monitor makes the gateway reboot itself entirely.
To handle more of Av1's constrains, the communication handler is decomposed into:

- `OSCommunicationLogic`
- `OnlineServiceMonitor`
- `RequestStore`

The `OnlineServiceMonitor` monitors the Gateway's connectivity to the Online Service. Messages sent by the Online Service are used as heartbeats. An update is sent to the monitor for each message. If no message has been received by the Online Service in somewhat less a minute, the monitor makes the gateway ping the Online Service in order to check if it is reachable.
The `RequestStore` is used to satisfy the constraint that all incoming pluggable data and issued application commands are stored internally when the Online Service is unreachable. This is done by sending those kinds of requests through the `RequestStore` to store them there. Then a unique requestID is added to the request before it is passed to the `OSCommunicationLogic`. When the Online Service receives a request, it sends an acknowledgement for the requestID back to the gateway it came from. Then, the request is deleted from the `RequestStore`. This way, if the Online Service becomes unreachable at any moment, all data that should be stored in this situration already is stored in the `RequestStore`.
After the monitor detects that a connection to the Online Service is possible again, it makes the Gateway start synchronising again. In this case, the `OSCommunicationLogic` requests the data from the `RequestStore` as new requests and tries again.
When the Online Service is unreachable, application parts running locally on the SIoTIP Gateway continue to operate normally. Nothing is done to influence the way the application parts work in that situation.

On the Online Service, a very similar approach is taken. There, the `GatewayCommunicationHandler` exists out of:

- `GWCommunicationLogic`
- `GatewayMonitor`

which have responsibilities analogue to the Gateway's version. Also, the `GatewayCommunicationMonitor` here monitors the Online Service's internal communication components, but does not restart them, as this was not necessary for the quality attribute.
A difference between the `GatewayMonitor` and `OnlineServiceMonitor` is that the `GatewayMonitor` has to keep track of the connectivity of *all* SIoTIP Gateways. The `GatewayMonitor` detects failures of gateways when 3 consecutive expected synchronisations do not arrive within 1 minute of their expected arrival time, as Av1 states. When a status change of a Gateway is detected, this is reflected in the `DeviceDB`. This is done to be able to notify a SIoTIP system administrator after the detection of a simultaneous outage of more than 1% of the registered gateways.

Considered Alternatives

Alternative for monitoring of Gateways Instead of making the GatewayMonitor check how many synchronization periods Gateways have missed, we could make it ping all of them and detect outages faster. However, this would massively increase the amount of messages sent by the `GatewayCommunicationHandler`, and thus is not a good option.

Alternative for communication We have chosen that communication between Gateways and the Online Service is to be done in a "Remote Procedure Call-like manner". Another option would have been to use some pattern like Message Channel or Publisher-Subscriber. We believe these other ways to be too slow because of the way the messages would need to be stored somewhere and then some time later consumed by the Gateways. This would thus also take more resources than Remote Procedure Calls. Also, it might be more intuitive to develop this system using Remote Method Invocation, where for example a part of the `ApplicationManager` could just invoke the "sendActuationCommand" method on a Gateway object, rather than send a message through some channel, and parse that message when developing the Gateway communication component.

Deployment Decisions

For both the Online Service and Gateways, the components used for communication (`OnlineServiceCommunicationHandler`, `GatewayCommunicationHandler`) are to be deployed on different nodes than their monitoring components (`OnlineServiceCommu` `GatewayCommunicationMonitor`).
This is done so that if the node of a communication component fails, its monitoring component would not also fail and thus detect nothing because it was running on the same node.

1.2 Av2: Application failure

Key Decisions

- ApplicationInstances are executed within ApplicationContainers.
- ApplicationContainerManager creates/destroys/handles communication for ApplicationContainers.
- ApplicationContainerMonitor monitors ApplicationContainers and ApplicationInstances.
- ApplicationExecutionSubsystemMonitor monitors the application execution subsystem.

Employed tactics and patterns: container

Rationale

To handle Av2, we developed the whole application execution subsystem in a decomposition with Av2 and important application instance related use cases. The application execution subsystem is composed of the following components:

- ApplicationContainer
- ApplicationContainerMonitor
- ApplicationContainerManager
- ApplicationExecutionSubsystemMonitor
- DeviceDataConverter
- DeviceCommandConstructor

Av2 states "The system is able to autonomously detect failures of its individual application execution components, failing applications, and failing application containers.".
These responsibilities are handled by the ApplicationContainer, ApplicationContainerMonitor, and ApplicationExecutio
The ApplicationContainer provides a sandbox environment for an ApplicationInstance to run in and has the ability to monitor the instance. When an application instance fails, the container notifies the ApplicationContainerMonitor of this.
Next to this, the ApplicationContainerMonitor pings ApplicationContainers regularly to check whether or not the container has failed. If a failure of an application instance/container is detected, the ApplicationContainerMonitor sends a command to the ApplicationContainerManager to restart the instance/container or to create a new one in case the first two restarts did not work. If the application instance then keeps failing, it is suspended and the application provider and affected customer organisation are notified. When an application fails, a message of this is sent to the other parts of the application, so that they can possible run in a degraded mode.
The ApplicationContainerMonitor also keeps track of how many times an ApplicationInstance has failed, so that when the container fails, there also is data about the instance.
Lastly, the ApplicationExecutionSubsystemMonitor monitors other parts of the application execution subsystem, and restarts them in case failure is detected.

Because of how different ApplicationInstances run each in their own ApplicationContainer, no other applications or availability of other functionality of the system is affected.

Deployment Decisions

For Av2, is important that ApplicationInstances are deployed on different node from the ApplicationContainerMonitor that is responsible for monitoring. If the node the ApplicationInstance fails, then the ApplicationContainerMonitor won't also automatically fail with the ApplicationInstance and interested parties can be informed.
In order to make sure that no applications or functionality of the system is affected when an application instance/container fails, each ApplicationContainer runs on its own node. The ApplicationContainerManager keeps track of all ApplicationContainers.

1.3 Av3: Pluggable device or mote failure

Key Decisions

- `DeviceManager` monitors connected/operational devices for a gateway.
- `DeviceManager` stores which application instances use which devices and stores the application instances' requirements.

Employed tactics and patterns: heartbeat

Rationale

Gateways 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 `DeviceManager` component to monitor devices. This component uses timers to keep track of how long it has been since a device has sent a heartbeat or occured 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 is detected within 2 seconds after the arrival of the heartbeat.

Av3 states that 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.
This problem is also tackled by the `DeviceManager`. It stores the requirements (devices used and relationships between devices) for pluggable devices set by application instances for all instances that use the gateway that the `DeviceManager` runs on. When it detects that an application can no longer operate due to failures, it sends a command to the `ApplicationContainerManager` on the Gateway itself and on the Online Service to suspend that application. When the required devices are operational again, the `DeviceManager` detects this and sends a command to check if application instances can be reactivated.

Considered Alternatives

Alternative for failure detection An alternative would have been to move the `DeviceManager` 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.

Alternative for 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.

1.4 M1: Integrate new sensor or actuator manufacturer

Key Decisions

- Modifiability is maintained by splitting up functionality that would need to be updated when new pluggable devices are introduced to the system into different components.
- In interfaces, pluggable devices are only referred to by their unique PluggableDeviceID, types and other device info is left out of parameter lists.
- `PluggableDeviceDataDB` and `DeviceDB` store data about devices in a generic way.

Employed tactics and patterns: generalize modules

Rationale

M1 states that available application instances in the system can be updated to use any new pluggable device. This is made possible by not adding specific device type information in any method parameters. All available applications can request data (be it sensor data or just specific data about a device) from new types of devices simply by referring to them by their ID, if they have the rights to use that device. The new type of info will then be encapsulated by the `DeviceData` data type. This means that new information would have to be added in the `DeviceDB` about the new pluggable device (and possible its new type).
Thus, the new types of sensor or actuator data can be transmitted, processed and stored, and be made available to applications.

For the problem of data conversion, we have added the `DeviceDataConverter`. This component can be used by `ApplicationInstance`s to convert data into a desired format. This component simply contains methods to convert data. If a new type of data were to be added to the system, then some new methods could be added to this component to support converting to/from that new type of data. The component would then have to be re-deployed.

The `DeviceCommandConstructor` would need to be able to verify the correctness of actuation and configuration commands and generate the actual commands for the devices. This is handled by letting the `DeviceCommandConstructor` fetch the necessary data from the `DeviceDB`. Thus, this kind of information should also be added to the `DeviceDB` when adding a new device to the system.

M1 also states that 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. As all of the necessary data for these scenarios about the pluggable device comes from the `DeviceDB`, nothing else needs to be done for this point.

1.5 P1: Large number of users

Rationale

The use cases mentioned in P1 are mostly related to Infrastructure Owner requests and Customer Organisation requests. Our split of the databases into `PluggableDeviceDB`, `DeviceDB`, `OtherDataDB`, goes well with this statment, as most of the requests of Infrastructure Owners lead to requests in the `DeviceDB` and most of the requests of Customer Organisation lead to requests in the `OtherDataDB`. This means that as both of these groups of users grow, the growth of one group will not influence the performance noticed by the other group.

P1 also states that scaling up to service an increasing amount of infrastructure owners, customers organisations and applications should (in worst case) be linear. This is certainly possible to an extent, as all the components related to requests from infrastructure owners and customers organisations are stateless. They could be replicated and load balancing could be applied.
The story for applications is slightly more complex, as at a certain point more `ApplicationContainerManager` nodes would be necessary to keep track of all `ApplicationContainer`s. This would then also create a need for some kind of coordinator so that all `ApplicationContainerManager`s can work together correctly to keep track of and maintain all `ApplicationContainer`s.

A problem for linear scaling we did not handle is the performance of the databases. In case replication and load balancing is applied on other components to handle a growing number of users, the databases will become the bottleneck of the system. To handle this, replication for performance should be used on the databases.

Another problem that we didn't handle is the scalability of the databases themselves. Ideally, a database management system that supports linear scalability should be used.

1.6 P2: Requests to the pluggable data database

Key Decisions

- `PluggableDeviceDataDB` separates the requests concerning pluggable data so that those requests have no impact on requests concerning other data.
- `DeviceDataScheduler` handles all requests for the `PluggableDeviceDataDB`, recognizes when the data processing subsystem needs to run in normal or overload mode, and prevents starvation of requests.

Employed tactics and patterns: scheduling, dynamic priority scheduling, logical separation, physical separation

Rationale

P2 required the pluggable data processing subsystem to be able to run in different modes, depending on whether or not certain request deadlines at met. Also, a mechanism should be in place to avoid starvation of any type of request. To handle this problem, the `PluggableDeviceDataScheduler` is introduced to the system. 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.

Furthermore, P2 requires that the processing of (large amounts of) requests concerning pluggable data has no impact on requests concerning other data. In order to statisfy this constraint, all data directly related to pluggable device data has been separated into the `PluggableDeviceDB`. All requests concerning pluggable data will be handled by this new component. By doing this, requests concerning pluggable data will have no impact on requests concerning other data.

Deployment Decisions

It is important that the `PluggableDeviceDB` is deployed on a separate node so that it does not impact any other database.

1.7 U2: Easy installation

Key Decisions

- Gateways automatically register themselves with the Online Service when a connection is possible.
- Motes and pluggable devices are automatically added in topologies and have their statuses updated in the database when they are plugged in and detected.
- When a status change is detected for pluggable devices, the `DeviceManager` automatically checks or makes the `ApplicationManagementLogic` check and activate
  deactivate specific applications.

Rationale

One of U2's constraints is that Gateway installation should be straightforward. It is a given that the SIoTIP Gateways will automatically connect to the internet after an infrastructure owner quickly set it up. After that, when a Gateway is not registered with the Online Service, if will automatically register. Then, this will be reflected in the SIoTIP system, the Gateway will show up as 'unplaced' in the infrastructure owner's Gateway, and the infrastructure owner will receive a notification of this.
When Gateways disconnect from the Online Service (due to e.g. communication channel failure), the `GatewayMonitor` on the Online Service notices this and updates the Gateway's status in the system. When a Gateway notices it can connect to the Online Service again (`OnlineServiceMonitor`), it will automatically connect again and be reactivated in the system (no topology update by infrastructure owner required).

Next, U2 says that mote installation, should not require more configuration than adding it in the topology. When the `DeviceManager` notices a new mote (heartbeat coming from a new device), it registers it in the system, links it to the gateway, and adds it as 'unplaced' in the topology of its infrastructure owner. If the mote disconnects and reconnects on the same gateway, it will be automatically reactivated and take it's last status/position in the topology.

For pluggable devices, the same holds as for motes, except that the new pluggable device is then linked to it's mote in the system. Also, pluggable devices get the location of their motes in the topology by default, but still start with status 'unplaced'.

For both motes and pluggable devices, if they are connected to a new device (gateway or mote) then the data concerning their last location and status is deleted.

Lastly, U2 states that applications should work out of the box if the required sensors and actuators are available. For this, each time the `DeviceManager` recognizes a new pluggable device, it starts a check for application instances (related to customer organisations associated with the gateway's infrastructure owner) that can be activated.

Considered Alternatives

Alternative for choice easy Gateway installation To avoid having the infrastructure owner to place a new Gateway in the topology (to be more in line with U2), we could let the Gateway be linked to a location in the topology at the moment infrastructure owner buys the Gateway. Then, after installing the Gateway, it will automatically register and show up as 'placed' in the topology. However, this would just move the problem of placing the Gateway in the topology instead of solving the problem. It is then better to make the infrastructure owner place the Gateway after physically installing it. This way the infrastructure owner kan keep some spare Gateways that are not linked to a location and install them when one of the Gateways break.

1.8 Other decisions

1.8.1 Authentication

KeyDecisions

A database is used to store login sessions.

Rationale

Authentication is done by verifying user credentials when a user wants to log in. Then a unique session is created and the sessionID of this session is returned to the client used by the user. In all subsequent requests, the user's client adds this sessionID with the request. If the sessionID exists in the database, then this means that the user is still logged in and the request can be handled. Every time the sessionID is checked in the database, a timestamp is reset. After a certain time without any requests for this sessionID, the session is deleted from the system.

The `SessionDB` is used to store all sessions.

Considered Alternatives

Files for sessions The sessions could be stored as files on the UserFacade that handlers requests for a user. The facade could then check whether or not a certain file exists to check if a user is logged in. However, this would make the facades stateful and would add extra overhead in keeping the session files on different nodes consistent with each other in case that the user is redirected to another node for load balancing. Keeping different nodes for facades consistent is much easier if they are all stateless.

Deployment Decisions

The `SessionDB` is to be deployed on a separate node in order to avoid puttting extra strain on the other databases. We expect the SessionDB to receive many more requests than e.g. the `OtherDataDB`, since users need to send a sessionID with every request so that it can be checked whether or not the user is (still) logged in.

1.8.2 Notifications

KeyDecisions

The NotificationHandler stores and sends notifications.

Rationale

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. The `DeviceManager` sends request to the external delivery service to notifies the infrastructure owner, once mote or pluggable device failure occurs. Since they are stored in the database, users can always view their notifications via their dashboard.

Considered Alternatives

Custom delivery solution 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.

1.8.3 Application execution subsystem

The application execution subsystem is composed of the following components:

- `ApplicationContainer`
- `ApplicationContainerMonitor`
- `ApplicationContainerManager`
- `ApplicationExecutionSubsystemMonitor`
- `DeviceDataConverter`
- `DeviceCommandConstructor`

Rationale

We have designed the application execution subsystem to use the same components on the Online Service and on Gateways, because we wanted reuse a lot of functionality in order to make everyones' lives a little easier. This seemed possible to us as it seemed to us that in the requirements there was not much different between the application instances running on Gateways and on the Online Service.
The biggest difference is that because Gateways are weaker machines than the ones on the Online Service, the applications running on the gateways should be of smaller scales. This is perfectly possible by changing some configurations in the `ApplicationContainer`s so that they have stricter limits on the resources used of the node they are running on. If an application were to be too large or resource intensive, the `ApplicationContainerManager`'s 'testApplication' method could then generate error messages for the Gateway version of the application.
Furthermore, to make this work smoothly, some interfaces used by the application execution subsystem would need to be re-routed to use (the same methods on) different interfaces of different components. For example, the `DeviceCommandConstructor` on the gateway should use the `DeviceManager` to fetch the correct formatting syntax of a pluggable device, instead of the `DeviceDB`. These things could also be done by using some specific configurations when deploying the components.

1.9 Discussion

Overall, the architectural decisions lack alternative solutions and alternative deployments decisions. More research should be done about different ways to handle the non-functional requirements, in order to better judge and compare our solutions.

In retrospect, we should have handled the DeviceDataConverter (and all components which rely on specific pluggable device information) differently. If we let data conversion methods be stored on the `DeviceDB`, we could let the DeviceDataConverter fetch those methods when necessary and maybe keep them in some kind of local cache. That way, the new methods can just be added in the database and the component would not need to be re-deployed. This is already done with pluggable device information on the `DeviceManager`; when it detects a new device, it loads information about the device and keeps it on the gateway while the device is connected to it.

We should have decomposed the `DeviceManager` some more to show how device connectivity is being tracked with a `DeviceMonitor` and how applications' device usage/requirements data is stored in a `ApplicationDataStore`.

Too much communication happens between Gateways and the Online Service, we should have let fewer requests happen between Gateways and the Online Service, and do more of the work on a `GatewayFacade` on the Online Service by doing different method calls from the `GatewayFacade` instead of `Gateway`s.
In the rationale of P1, it is noticeable that the quality attribute has not been completely handled.

In our design, we have not worked with states of application instances. Thus, if an application were to crash at some point, or if it would need to be shut down temporarily, etc. it would not be possible to save the state of an

application and restore it at a later point. Maybe a simple solution for this would be to force application providers to implement a certain method so that the state of an application could be retrieved and stored.

The `NotificationHandler` should have different notify methods for different situations, so that more details can be saved such as the cause of the notification.

2. Client-server view (UML Component diagram)

Figures

2.1 Context diagram for the client-server view. 17
2.2 Primary diagram of the client-server view. 19

2.1 Context diagram

The context diagram of the client-server view is displayed in figure 2.1.

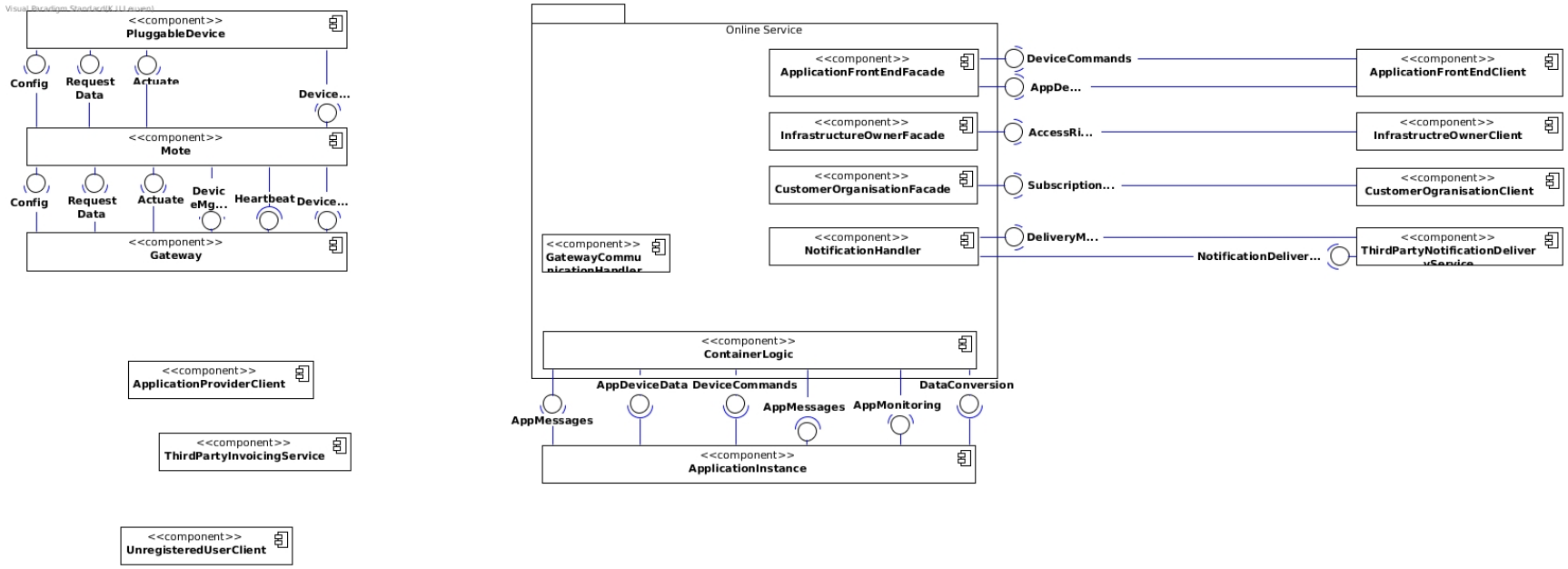

Figure 2.1: Context diagram for the client-server view.

2.2 Primary diagram

The primary diagram of the client-server view is displayed in figure 2.2.

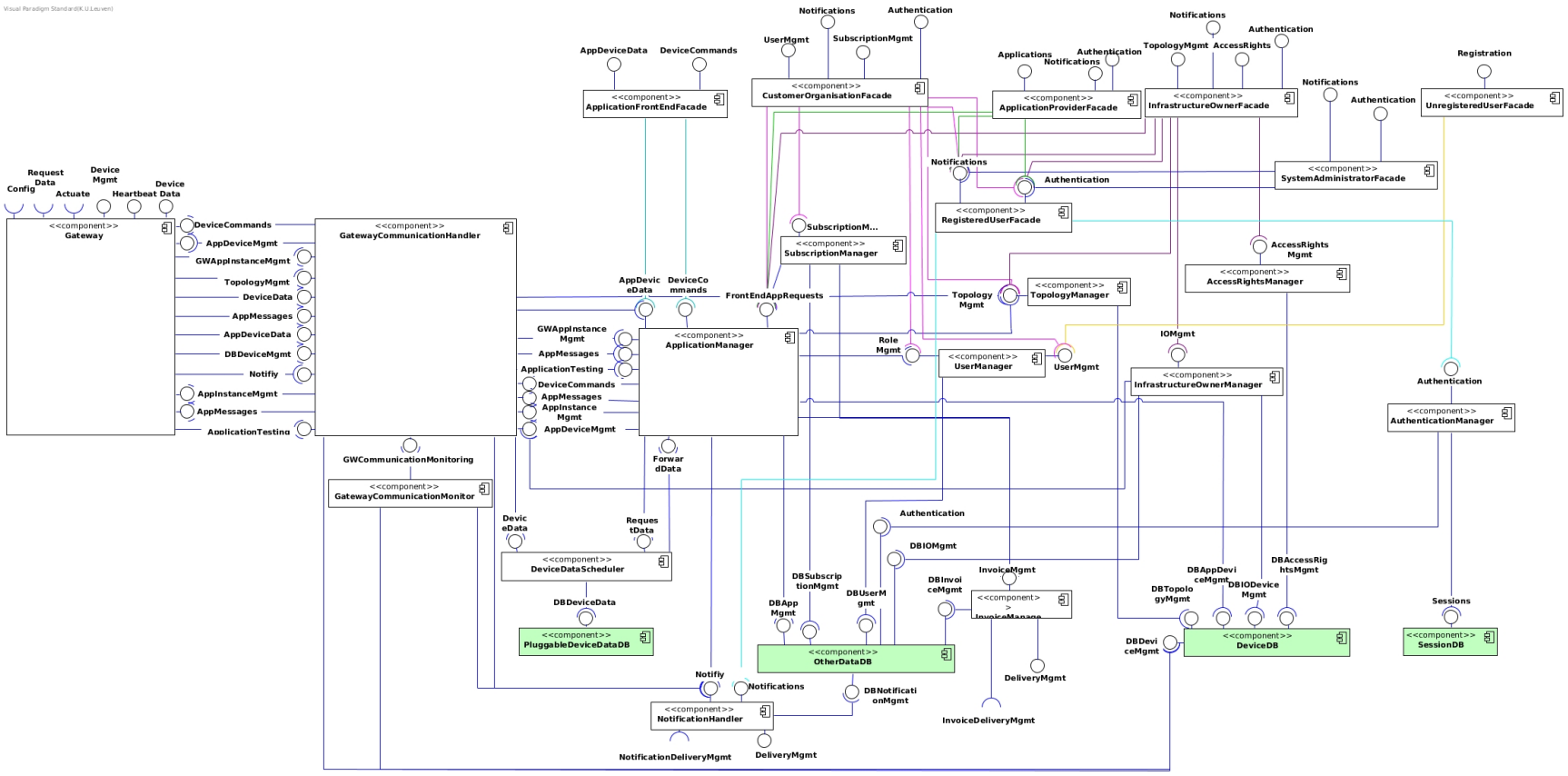

Figure 2.2: Primary diagram of the client-server view.

3. Decomposition view (UML Component diagram)

Figures

3.1 Decomposition of `Gateway` . . . 21
3.2 DDecomposition of `ApplicationManager` . . . 22
3.3 Decomposition of `GatewayBroker` . . . 23
3.4 Decomposition of `OnlineServiceBroker` . . . 24

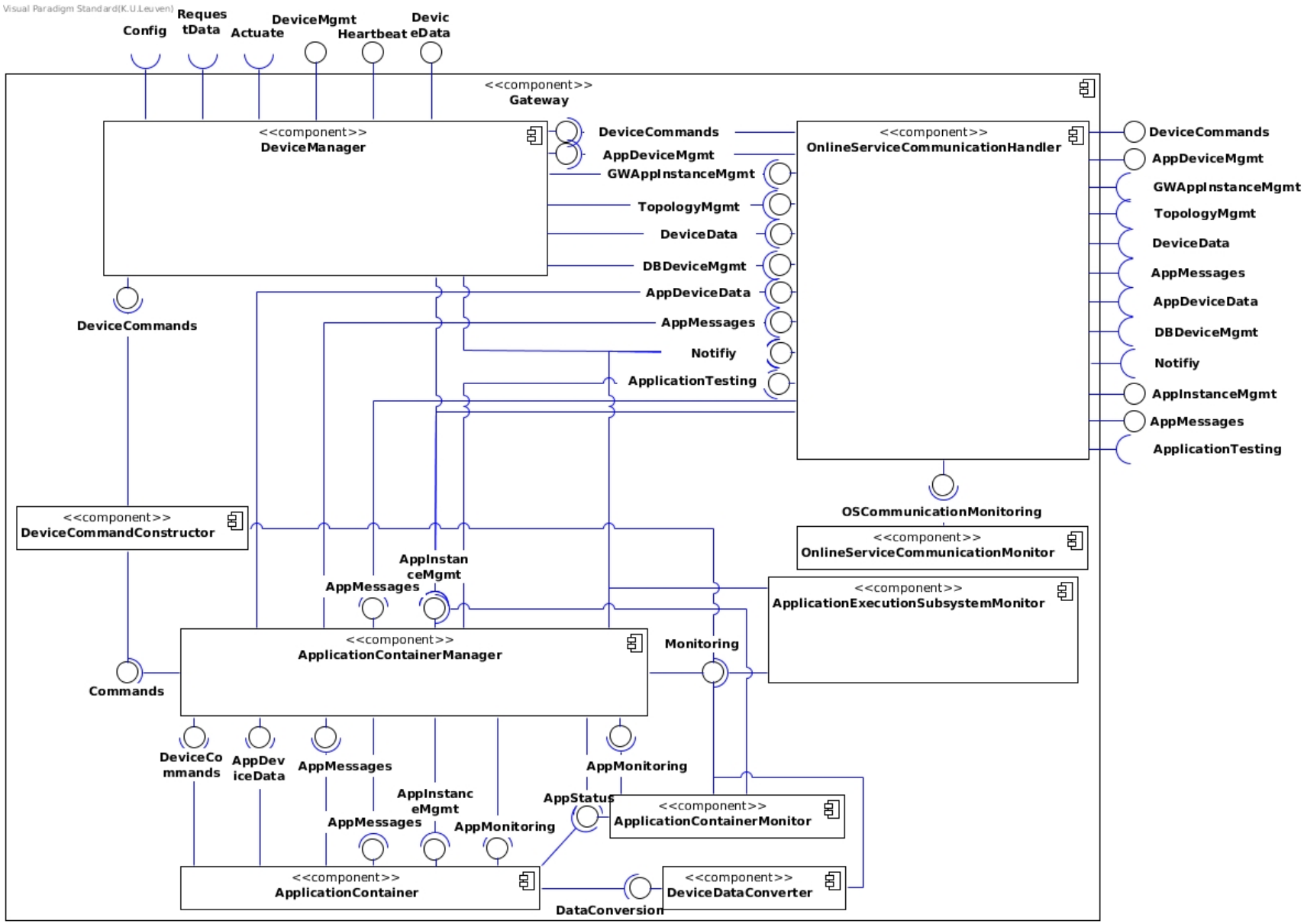

Figure 3.1: Decomposition of `Gateway`

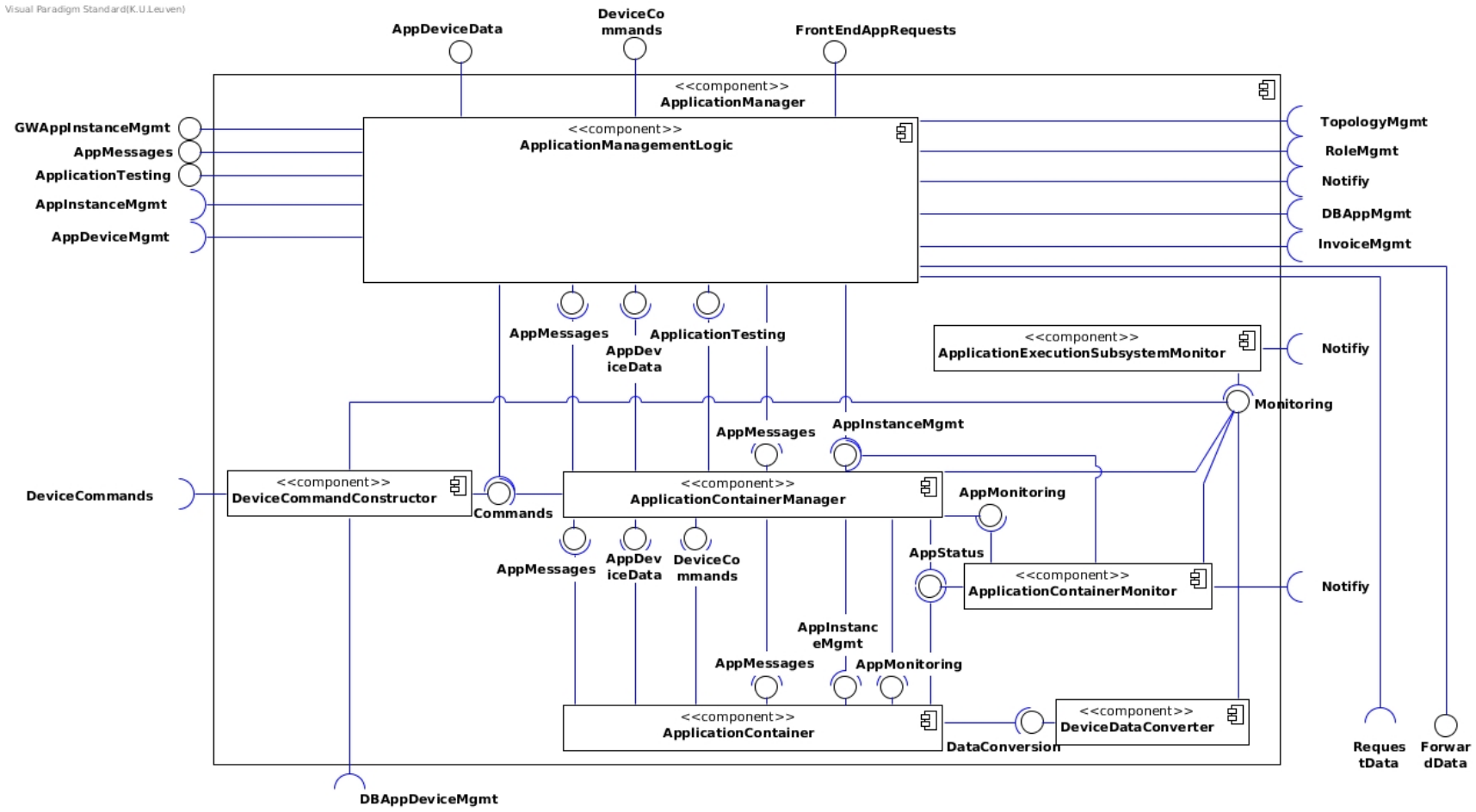

Figure 3.2: DDecomposition of `ApplicationManager`

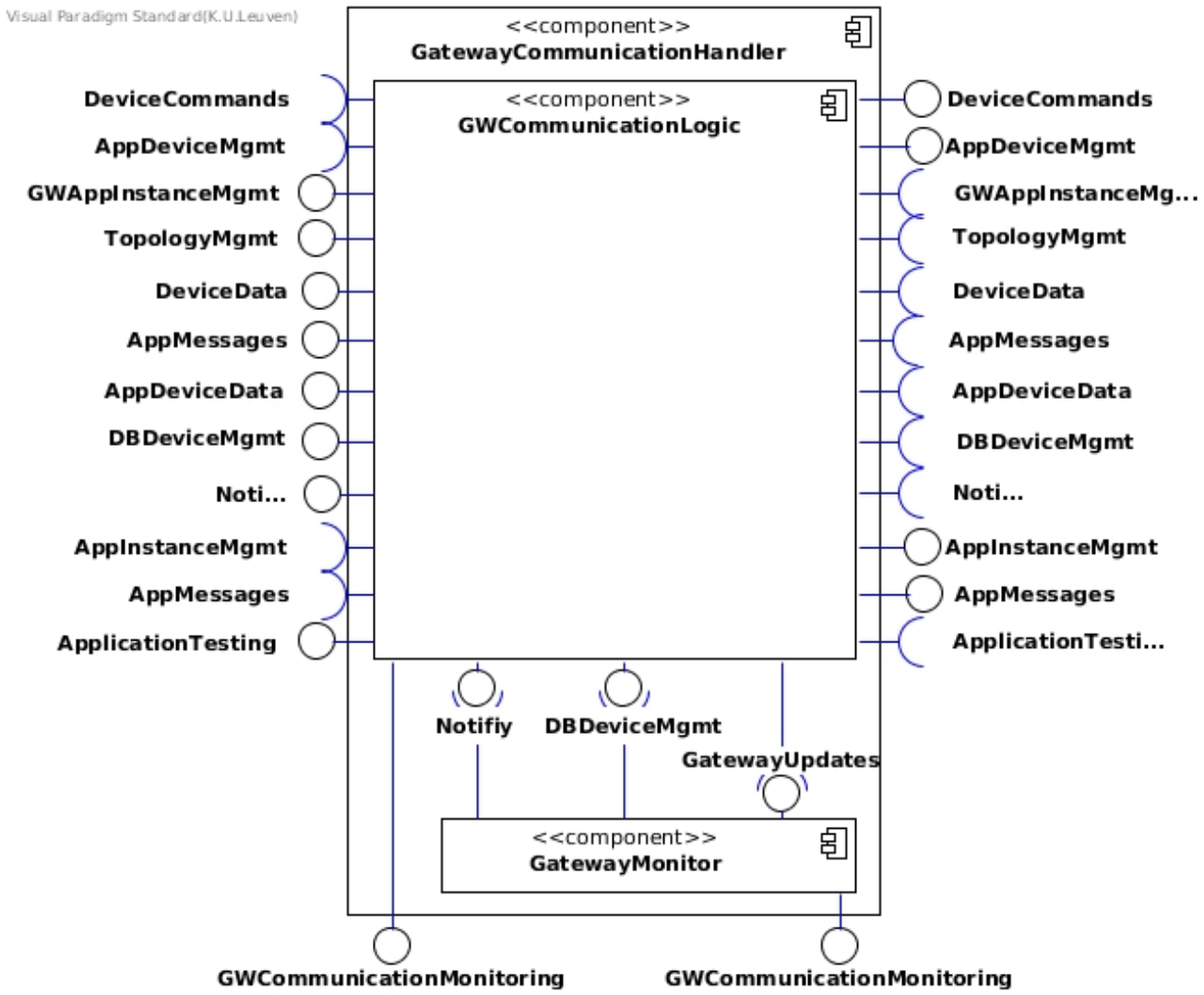

Figure 3.3: Decomposition of `GatewayBroker`

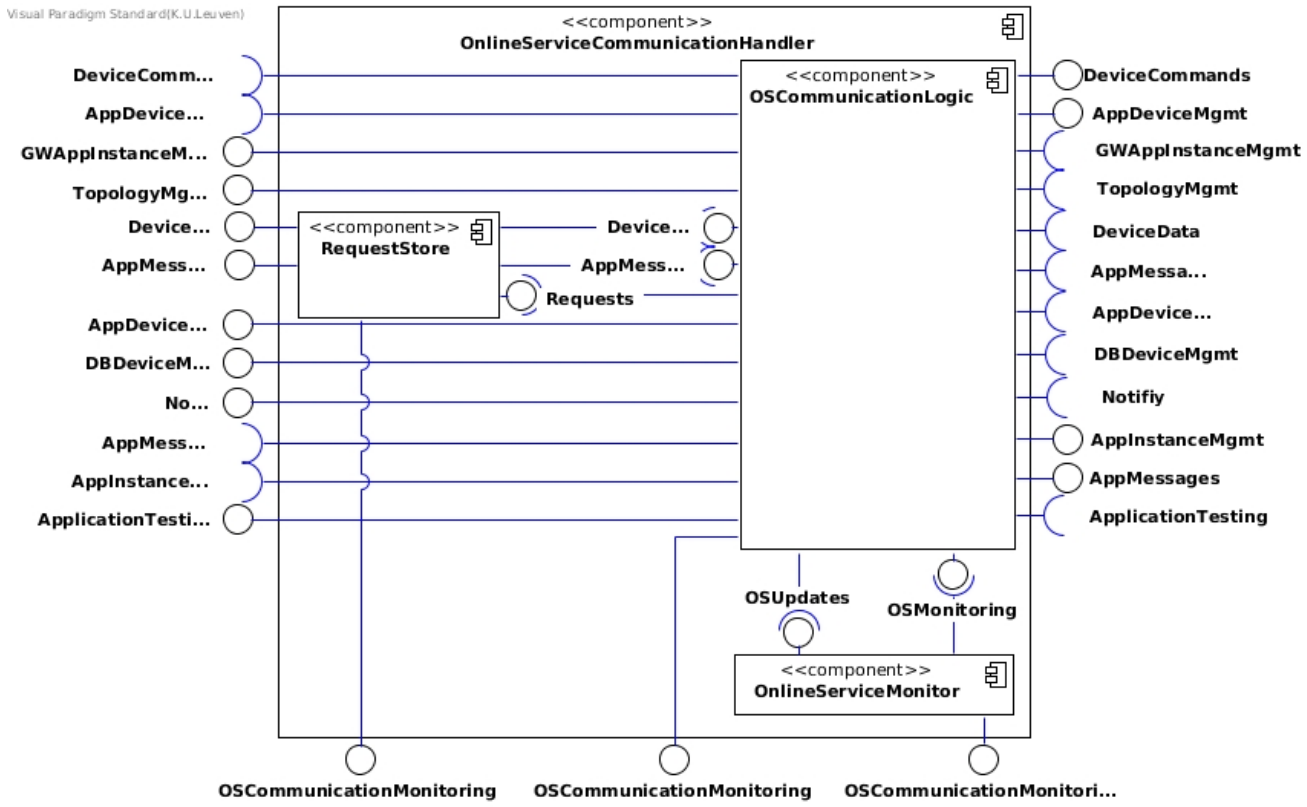

Figure 3.4: Decomposition of `OnlineServiceBroker`

4. Deployment view (UML Deployment diagram)

Figures

4.1 Context diagram for the deployment view. 26
4.2 Primary diagram for the deployment view. 27

4.1 Context diagram

The context diagram for the deployment view is displayed in figure 4.1.

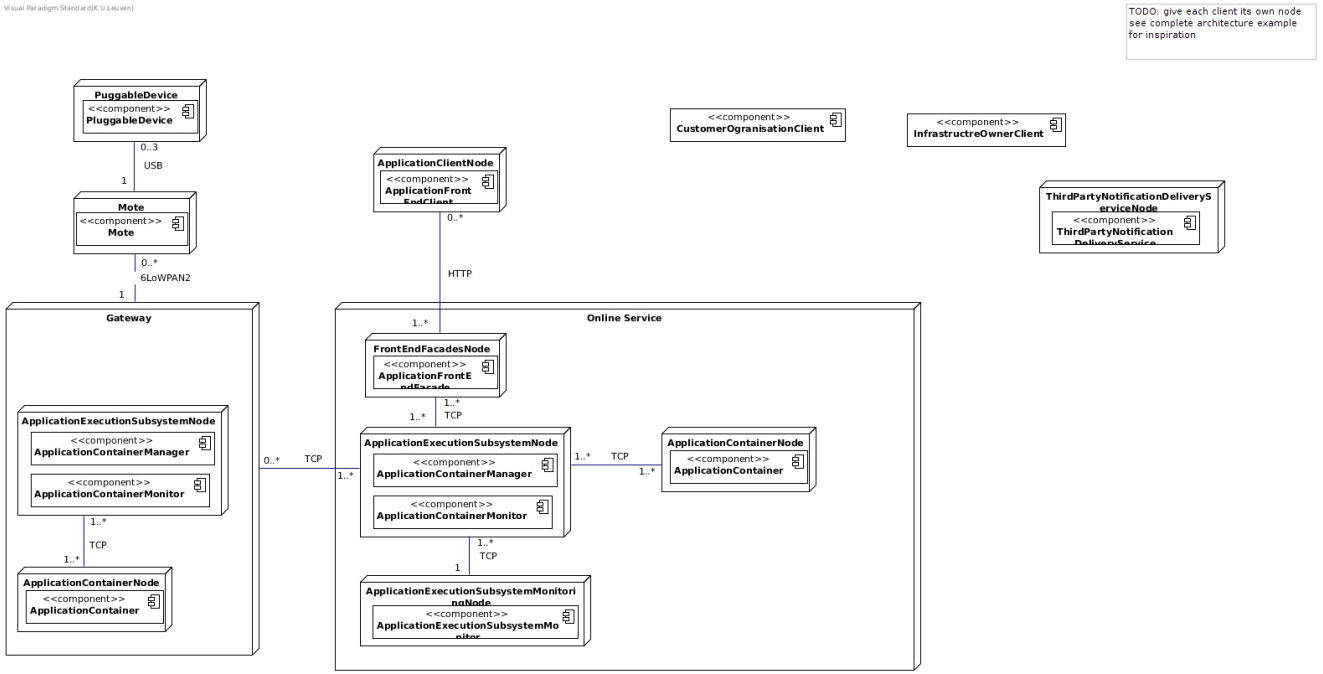

Figure 4.1: Context diagram for the deployment view.

4.2 Primary diagram

The primary diagram for the deployment view is displayed in figure 4.2.

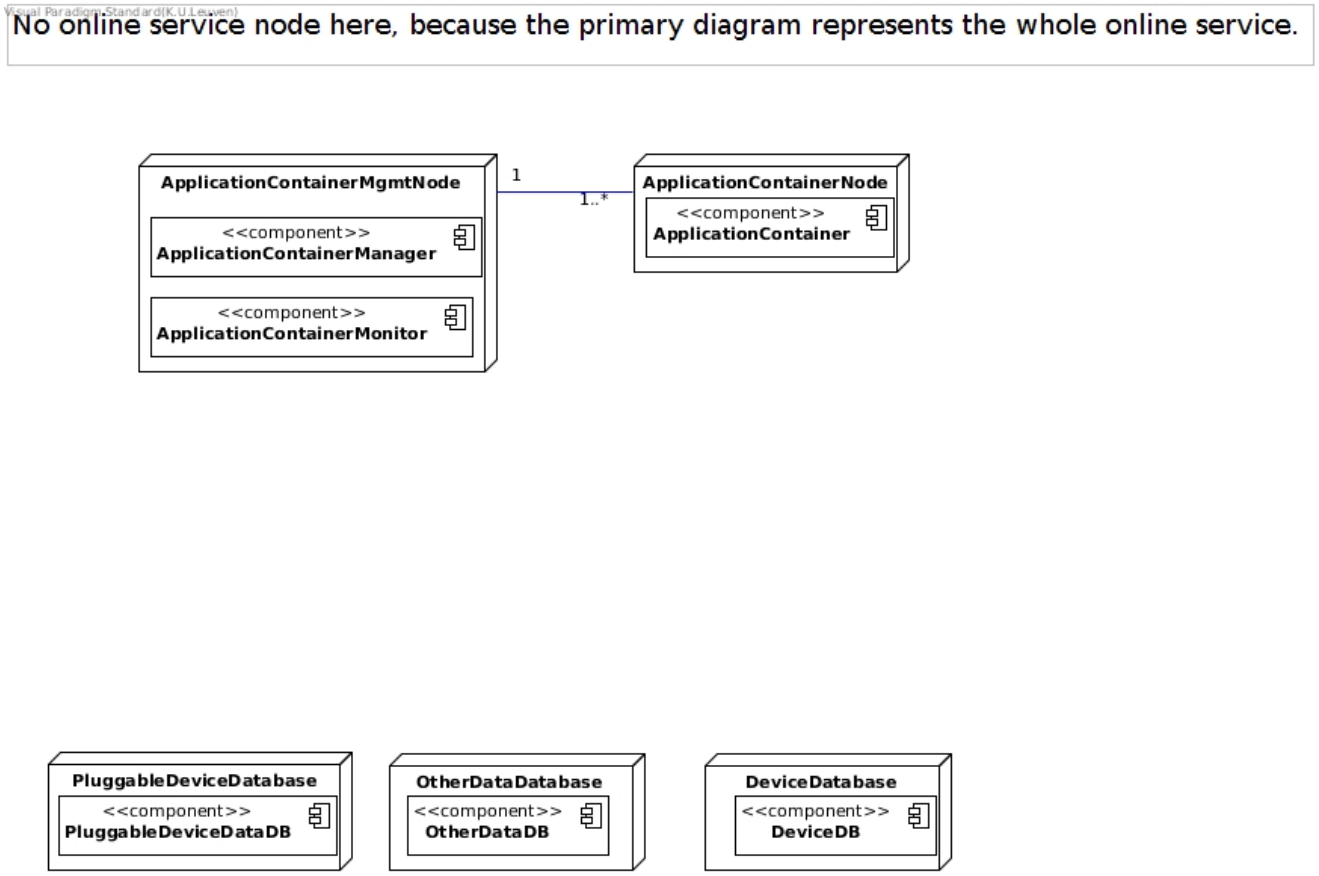

Figure 4.2: Primary diagram for the deployment view.

5. Scenarios

Figures

5.1 Sensor data being processed by the system . . . 29
5.2 Subscribing to an application . . . 30
5.3 Applications issuing actuation commands . . . 31
5.4 Scenario . . . 32
5.5 Application crash . . . 33
5.6 Plugging in a new pluggable device (sensor or actuator) . . . 34
5.7 Detection and handling of communication channel failure . . . 35
5.8 Upgrading an application . . . 36
5.9 Sending actuation commands via a mobile app . . . 37

✔ **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.

5.1 Scenarios

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 you architecture at this point.

This section lists which sequence diagrams belong to which scenarios:

- UC11: Sensor data being processed by the system
  Figure 5.1
- UC19: Subscribing to an application
  Figure 5.2
- UC12: Applications issuing actuation commands
  Figure 5.3
- UC14, Av3, UC18: Sensors/actuators failing
  Figure 5.4
  This scenario displays the data flow when sensors/actuators fail, causing

  - deactivation of specific applications
  - a redundant sensor/actuator to take over in the context of a single application
- Av2: Application crash
  Figure 5.5
- U2, UC4: Plugging in a new pluggable device (sensor or actuator)
  Figure 5.6
- Av1, UC15: Detection and handling of communication channel failure
  Figure 5.7
- UC22, U1: Upgrading an application
  Figure 5.8
- UC26, UC27, UC12: Sending actuation commands via a mobile app
  Figure 5.9

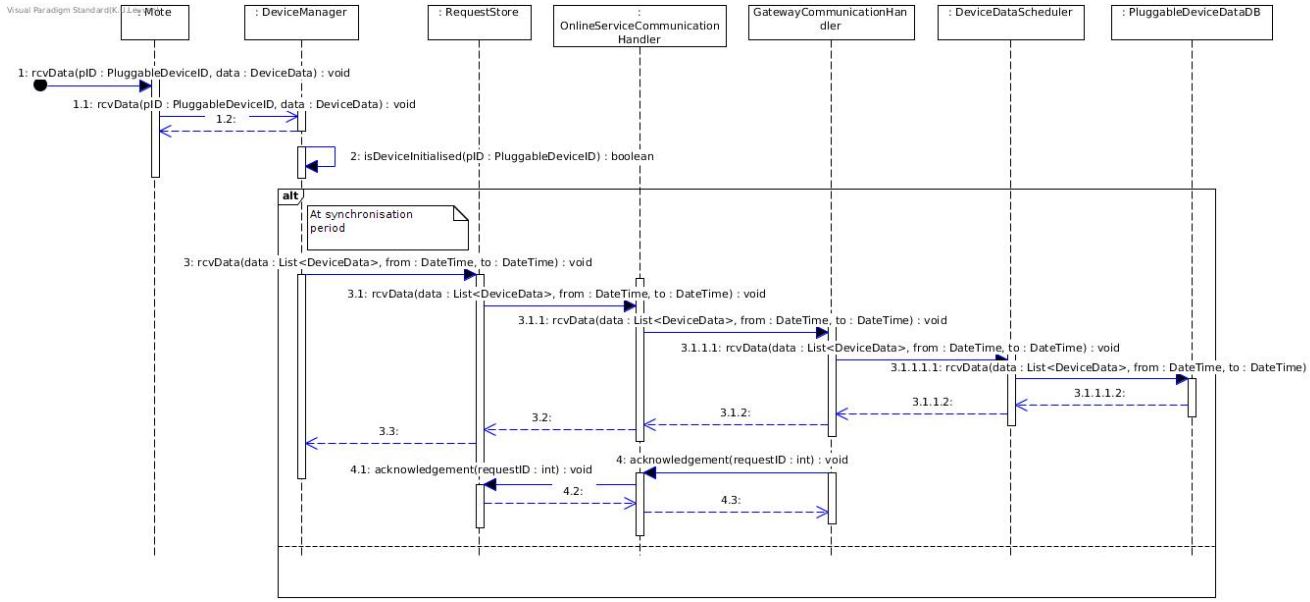

Figure 5.1: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

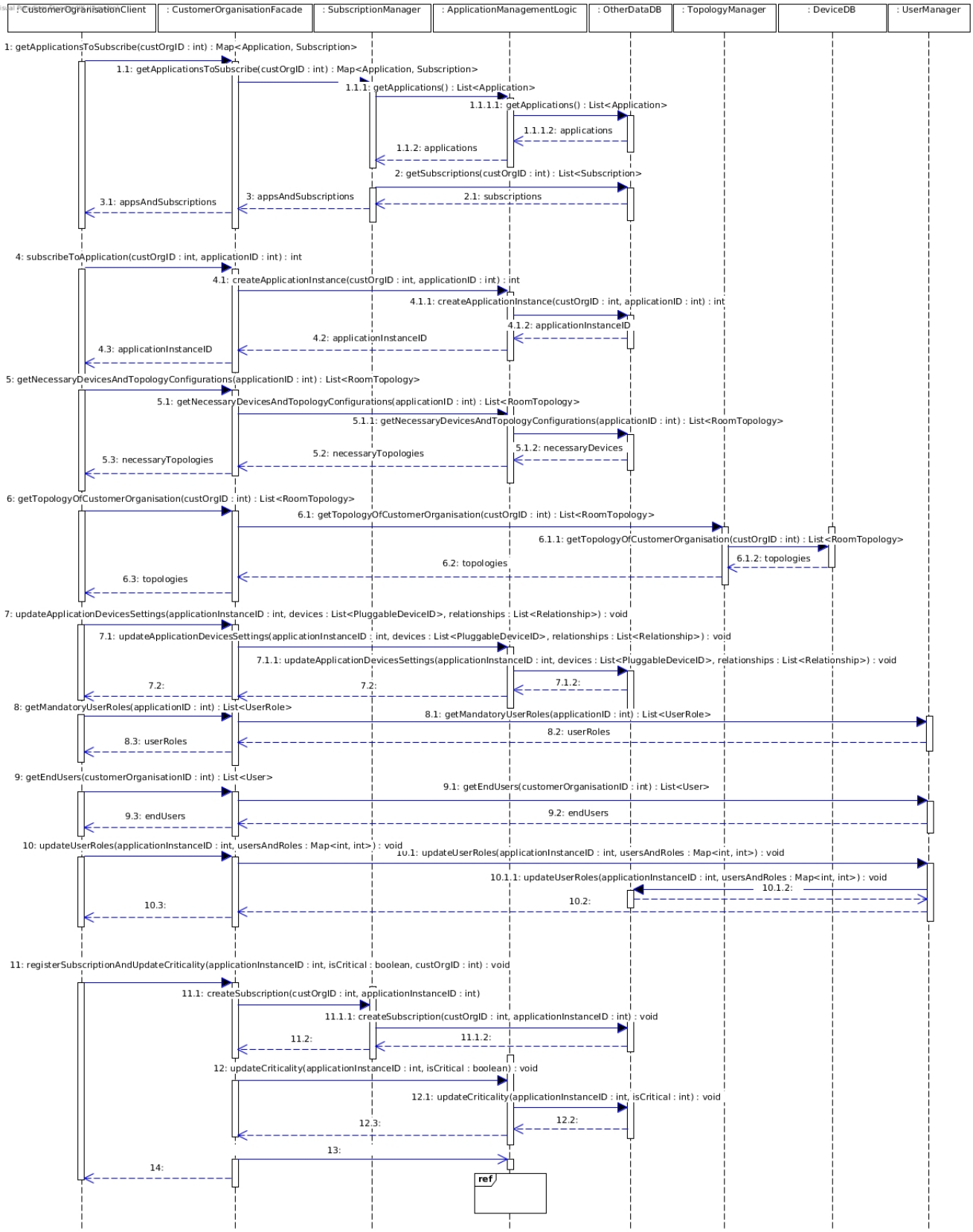

Figure 5.2: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

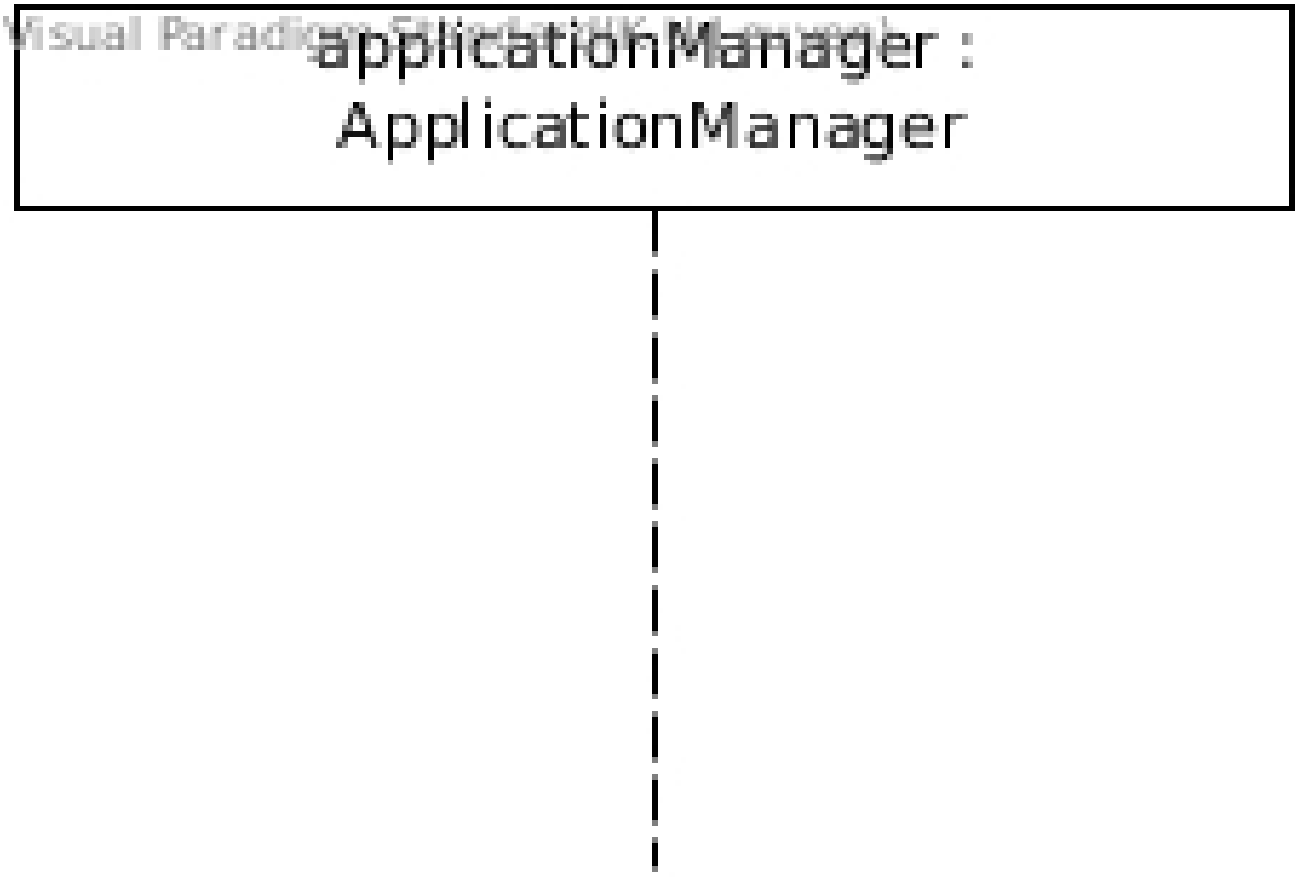

Figure 5.3: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

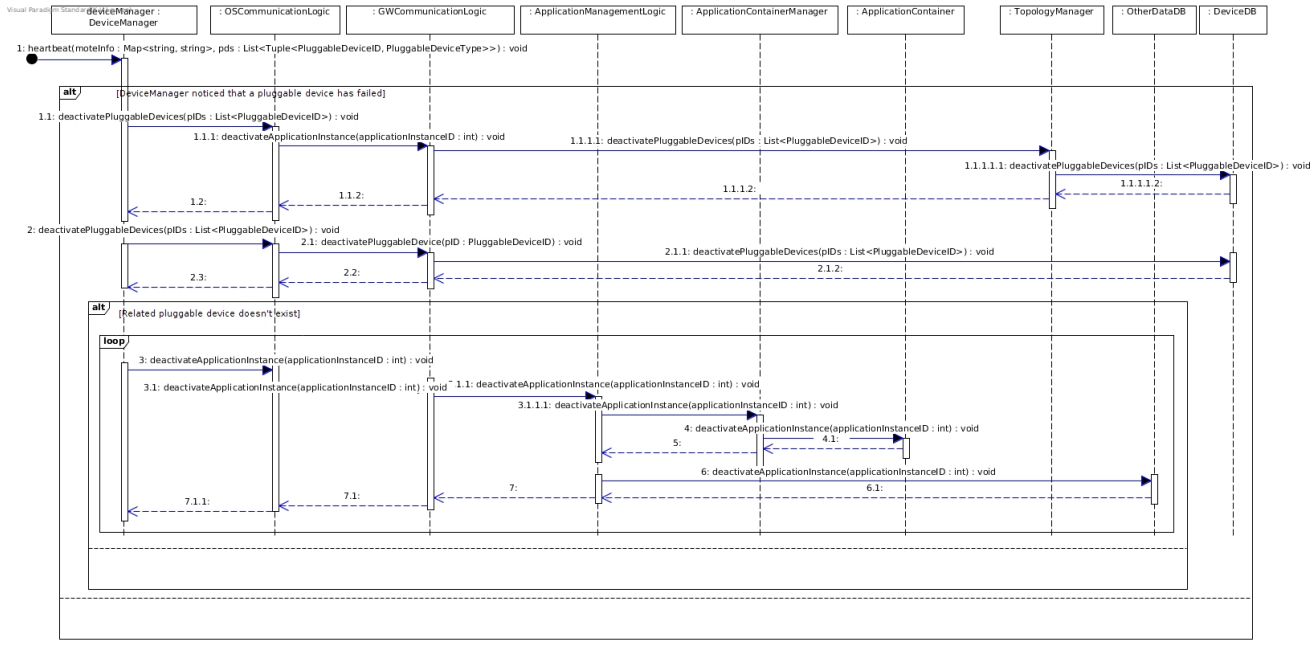

Figure 5.4: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

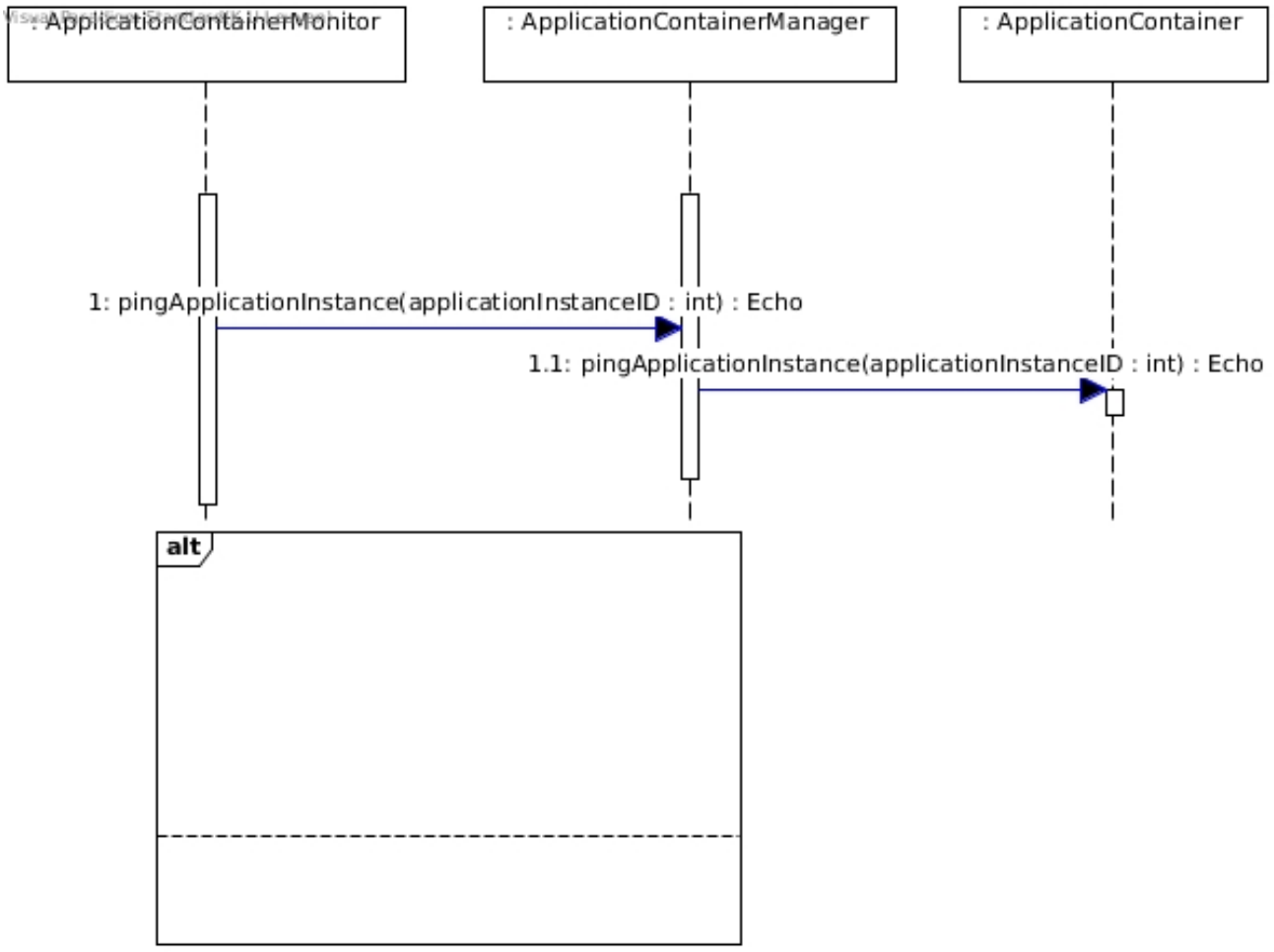

Figure 5.5: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

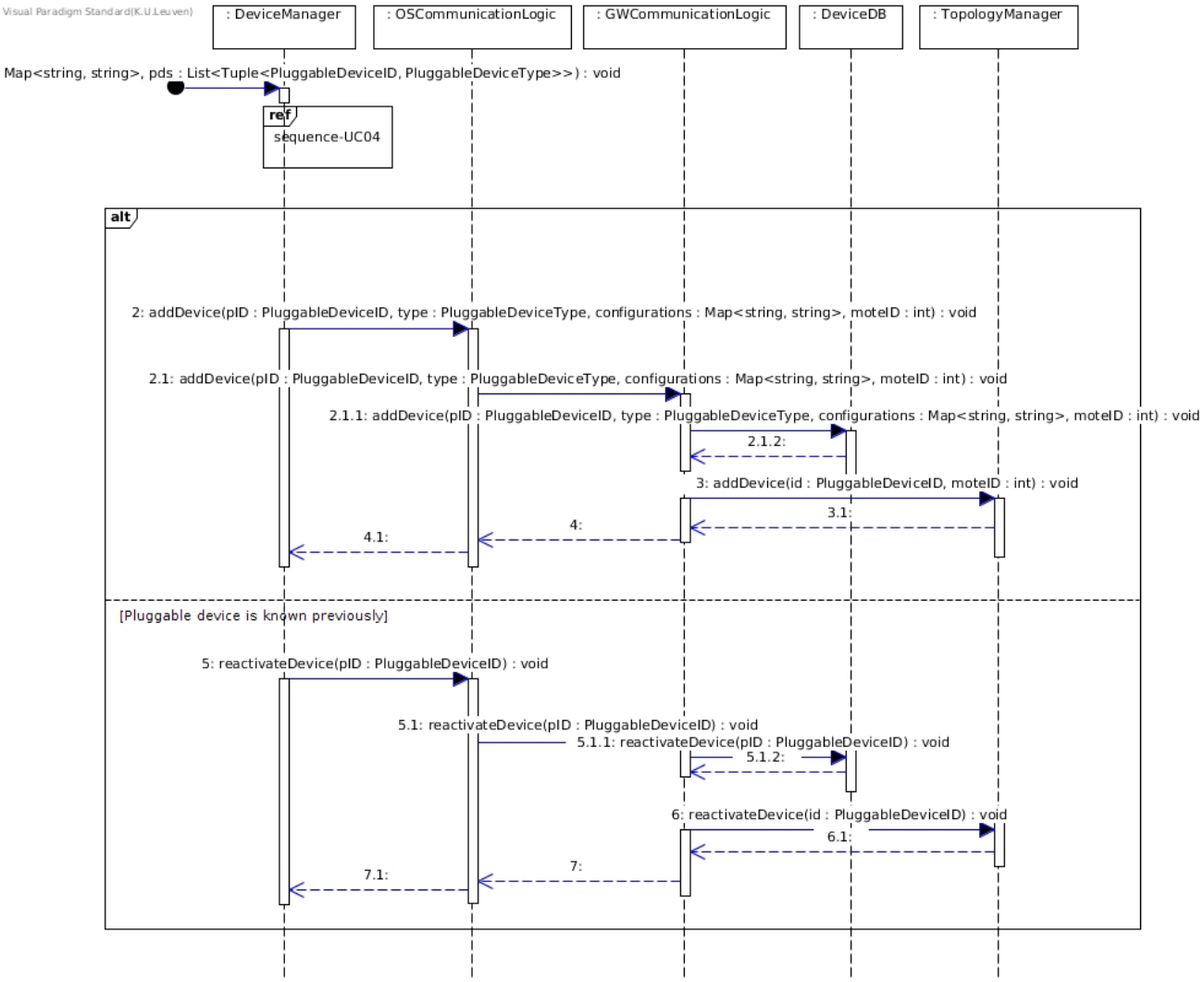

Figure 5.6: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

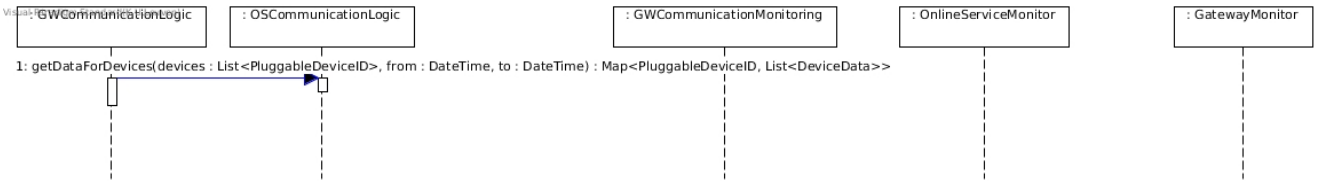

Figure 5.7: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

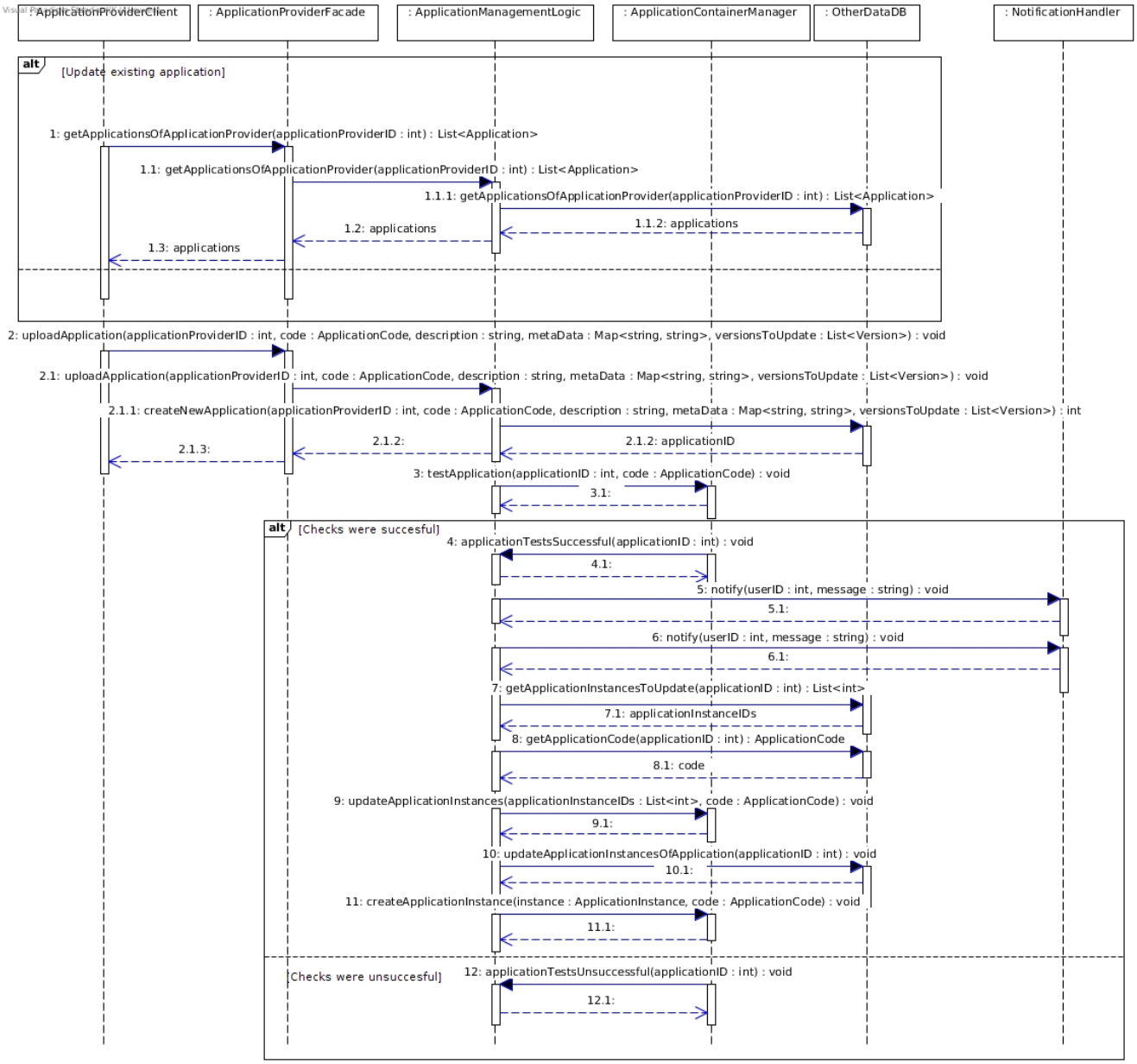

Figure 5.8: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

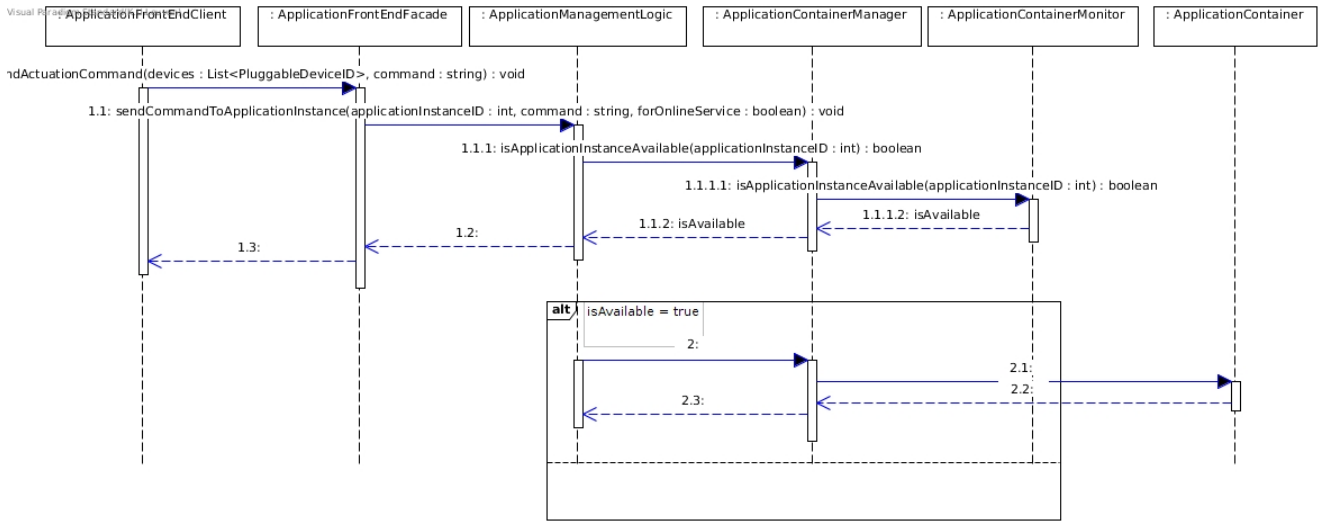

Figure 5.9: EXPLAIN WHAT HAPPENS IN THE SCENARIO.
ADD COMMENTS.
LINK TO OTHER RELEVANT SCENARIO'S.

6. Element Catalog and Datatypes

Each method contains a short note on why the method was added (under "Created for"). This was done to keep track of our decisions and does not mean that the methods can only be used for the Quality Attribute/Use Case referenced in the "Created for" note.

7. Catalog

7.1 Components

7.1.1 AccessRightsManager

Responsibility: Responsible for all functionality related to access rights to pluggable devices. E.g. retrieving the access rights a customer organisation has for a device, updating access rights for customer organisations, etc.
Super-components: None
Sub-components: None
Provided interfaces: AccessRightsMgmt
Required interfaces: DBAccessRightsMgmt

7.1.2 ApplicationContainer

Responsibility: This component contains a sandbox environment for an application instance to execute in. The container itself can detect when applications crash. In that case, the container notifies the ApplicationContainerMonitor.
Super-components: ApplicationManager
Sub-components: ApplicationInstance, ContainerLogic
Provided interfaces: AppInstanceMgmt, AppMessages, AppMonitoring
Required interfaces: AppDeviceData, AppMessages, AppStatus, DataConversion, DeviceCommands

7.1.3 ApplicationContainerManager

Responsibility: Responsible for management of ApplicationContainers. E.g. creation, suspending containers, forwarding data to containers, etc. All communication that is destined to ApplicationContainers goes through here first, since the ApplicationContainerManager knows where exactly ApplicationContainers are deployed.
Super-components: ApplicationManager
Sub-components: None
Provided interfaces: AppDeviceData, AppInstanceMgmt, AppMessages, AppMonitoring, Commands, DeviceCommands, Monitoring
Required interfaces: AppDeviceData, AppInstanceMgmt, ApplicationTesting, AppMessages, AppMonitoring, AppStatus, Commands, Notifiy

7.1.4 ApplicationContainerMonitor

Responsibility: Monitors ApplicationContainers and keeps track of their ApplicationInstances.
Super-components: ApplicationManager
Sub-components: None
Provided interfaces: AppStatus, Monitoring
Required interfaces: AppInstanceMgmt, AppMonitoring, Notifiy

7.1.5 ApplicationExecutionSubsystemMonitor

Responsibility: Monitors the application execution subsystem (ApplicationContainerManager,ApplicationContainerMon
Super-components: ApplicationManager
Sub-components: None
Provided interfaces: None
Required interfaces: Monitoring, Notifiy

7.1.6 ApplicationFrontEndClient

Responsibility: Represents the front end client a an application.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ⊀ AppDeviceData, ⊀ DeviceCommands

7.1.7 ApplicationFrontEndFacade

Responsibility: Acts as an access point for ApplicationClients and handles all functionality that can be done by application's external front ends.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ AppDeviceData, ⊸ DeviceCommands
Required interfaces: ⊀ AppDeviceData, ⊀ DeviceCommands

7.1.8 ApplicationInstance

Responsibility: Represents an application instance running in the system.
Super-components: ◰ ApplicationManager ▷ ◰ ApplicationContainer
Sub-components: None
Provided interfaces: ⊸ AppMessages, ⊸ AppMonitoring
Required interfaces: ⊀ AppDeviceData, ⊀ AppMessages, ⊀ DataConversion, ⊀ DeviceCommands

7.1.9 ApplicationManagementLogic

Responsibility: The entry point for ApplicationManager. Responsible for tying together all components of the ApplicationManager and handling requests related to applications.
Super-components: ◰ ApplicationManager
Sub-components: None
Provided interfaces: ⊸ AppDeviceData, ⊸ ApplicationTesting, ⊸ AppMessages, ⊸ DeviceCommands, ⊸ ForwardData, ⊸ FrontEndAppRequests, ⊸ GWAppInstanceMgmt
Required interfaces: ⊀ AppDeviceMgmt, ⊀ AppInstanceMgmt, ⊀ AppMessages, ⊀ Commands, ⊀ DBAppMgmt, ⊀ InvoiceMgmt, ⊀ Notifiy, ⊀ RequestData, ⊀ RoleMgmt, ⊀ TopologyMgmt

7.1.10 ApplicationManager

Responsibility: Responsible for all functionality related to applications and application instances. e.g. activating/deactivating application instances, updating an ApplicationInstance's pluggable device redundancy relationships, etc.
Super-components: None
Sub-components: ◰ DeviceCommandConstructor, ◰ ApplicationContainerMonitor, ◰ ApplicationContainerManager, ◰ ApplicationManagementLogic, ◰ ApplicationExecutionSubsystemMonitor, ◰ DeviceDataConverter, ◰ ApplicationContainer
Provided interfaces: ⊸ AppDeviceData, ⊸ ApplicationTesting, ⊸ AppMessages, ⊸ DeviceCommands, ⊸ ForwardData, ⊸ FrontEndAppRequests, ⊸ GWAppInstanceMgmt
Required interfaces: ⊀ AppDeviceMgmt, ⊀ AppInstanceMgmt, ⊀ AppMessages, ⊀ DBAppDeviceMgmt, ⊀ DBAppMgmt, ⊀ DeviceCommands, ⊀ InvoiceMgmt, ⊀ Notifiy, ⊀ RequestData, ⊀ RoleMgmt, ⊀ TopologyMgmt

7.1.11 ApplicationProviderClient

Responsibility: Represents the client used by a customer organisation. This is the user's dashboard.
Super-components: None

Sub-components: None
Provided interfaces: None
Required interfaces: None

7.1.12 ApplicationProviderFacade

Responsibility: Acts as an access point for application provider's client and handles all functionality that can be done by application providers.
Super-components: None
Sub-components: None
Provided interfaces: ○ `Applications`, ○ `Authentication`, ○ `Notifications`
Required interfaces: ⊂ `Authentication`, ⊂ `FrontEndAppRequests`, ⊂ `Notifications`

7.1.13 AuthenticationManager

Responsibility: Handles all functionality related to authentication of users.
Super-components: None
Sub-components: None
Provided interfaces: ○ `Authentication`
Required interfaces: ⊂ `Authentication`, ⊂ `Sessions`

7.1.14 ContainerLogic

Responsibility: The logic of an `ApplicationContainer`. Handles communication from/to ApplicationInstances and can maintain/monitor them.
Super-components: `ApplicationManager` ▷ `ApplicationContainer`
Sub-components: None
Provided interfaces: ○ `AppDeviceData`, ○ `AppMessages`, ○ `DataConversion`, ○ `DeviceCommands`
Required interfaces: ⊂ `AppMessages`, ⊂ `AppMonitoring`

7.1.15 CustomerOgranisationClient

Responsibility: Represents the client used by a customer organisation. This is the user's dashboard.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ⊂ `SubscriptionMgmt`

7.1.16 CustomerOrganisationFacade

Responsibility: Acts as an access point for CustomerOrganisationClients and handles all functionality that can be done by customer organisations.
Super-components: None
Sub-components: None
Provided interfaces: ○ `Authentication`, ○ `Notifications`, ○ `SubscriptionMgmt`, ○ `UserMgmt`
Required interfaces: ⊂ `Authentication`, ⊂ `FrontEndAppRequests`, ⊂ `Notifications`, ⊂ `RoleMgmt`, ⊂ `SubscriptionMgmt`, ⊂ `TopologyMgmt`, ⊂ `UserMgmt`

7.1.17 DeviceCommandConstructor

Responsibility: Is responsible for verifying, constructing, and sending commands from applications for pluggable devices. On the Online Service, It fetches data about pluggable devices from the `DeviceDB`. On a `Gateway`, it must be configured to fetch that data from the `DeviceManager`.
Super-components: `ApplicationManager`

Sub-components: None
Provided interfaces: Commands, Monitoring
Required interfaces: DBAppDeviceMgmt, DeviceCommands

7.1.18 DeviceDataConverter

Responsibility: The DeviceDataConverter is responsible for conversions pluggable device data for applications. There is one in the Online Service for application instances running in the Online Service and one in Gateways for application instances running on gateways.
Super-components: ApplicationManager
Sub-components: None
Provided interfaces: DataConversion, Monitoring
Required interfaces: None

7.1.19 DeviceDataScheduler

Responsibility: 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.
Super-components: None
Sub-components: None
Provided interfaces: DeviceData, RequestData
Required interfaces: DBDeviceData, ForwardData

7.1.20 DeviceDB

Responsibility: Contains all information related to devices in the system, but not pluggable device data such as sensor data or actuation statuses. The data includes information about pluggable devices, motes, gateways, topologies, access rights, etc.
Super-components: None
Sub-components: None
Provided interfaces: DBAccessRightsMgmt, DBAppDeviceMgmt, DBDeviceMgmt, DBIODeviceMgmt, DBTopologyMgmt
Required interfaces: None

7.1.21 DeviceManager

Responsibility: 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.
Keeps a local cache of data about pluggable devices from the DeviceDB, such as specific formatting syntax, possible configuration parameters, etc. This data can then be used by the DeviceCommandConstructor on the DeviceManager to convert commands from ApplicationContainers on the gateway before they are sent to pluggable devices.
Super-components: Gateway
Sub-components: None
Provided interfaces: AppDeviceMgmt, DeviceCommands, DeviceData, DeviceMgmt, Heartbeat
Required interfaces: Actuate, AppInstanceMgmt, Config, DBDeviceMgmt, DeviceData, GWAppInstanceMgmt, Notifiy, RequestData, TopologyMgmt

7.1.22 Gateway

Responsibility: Represents a SIoTIP gateway.
Super-components: None
Sub-components: DeviceManager, OnlineServiceCommunicationMonitor, OnlineServiceCommunicationHandler

Provided interfaces: AppDeviceMgmt, AppInstanceMgmt, AppMessages, DeviceCommands, DeviceData, DeviceMgmt, Heartbeat
Required interfaces: Actuate, AppDeviceData, ApplicationTesting, AppMessages, Config, DBDeviceMgmt, DeviceData, GWAppInstanceMgmt, Notifiy, RequestData, TopologyMgmt

7.1.23 GatewayCommunicationHandler

Responsibility: Isolates communication-related concerns between the Online Service Gateways and along with OnlineServiceBroker on Gateways. Forwards requests from one party to the other and transmits results and possible exceptions.
Sends acknowledgements for all messages sent by Gateways so that they can detect failures.
Super-components: None
Sub-components: GWCommunicationLogic, GatewayMonitor
Provided interfaces: AppDeviceData, AppDeviceMgmt, AppInstanceMgmt, ApplicationTesting, AppMessages, DBDeviceMgmt, DeviceCommands, DeviceData, GWAppInstanceMgmt, GWCommunicationMonitoring, Notifiy, TopologyMgmt
Required interfaces: AppDeviceData, AppDeviceMgmt, AppInstanceMgmt, ApplicationTesting, AppMessages, DBDeviceMgmt, DeviceCommands, DeviceData, GWAppInstanceMgmt, Notifiy, TopologyMgmt

7.1.24 GatewayCommunicationMonitor

Responsibility: Monitors the communication component on the Online Service for communication with Gateways on the Online Service. Notifies SIoTIP system administrators in case of failures.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: DBDeviceMgmt, GWCommunicationMonitoring, Notifiy

7.1.25 GatewayMonitor

Responsibility: Monitors the connectivity status of all gateways. Can detect that a gateway is not sending data anymore based on the expected synchronisation interval. If 3 consecutive expected synchronisations do not arrive within 1 minute of their expected arrival time, this is detected as a gateway outage.
When outages of gateways are detected, the infrastructure owners that own the gateways and a SIoTIP system administrator are notified.
When the connectivity status change of a Gateway is detected, this is saved in the DeviceDB.
Quick math to check we are not doing something very dumb here: 5000 gateways * (32 bit gatewayID + 4 bit countSynchronisationsMissed + 64 bit lastSyncPeriod + 64 bit nextSyncPeriod) == approx 100kb => Seems ok
Super-components: GatewayCommunicationHandler
Sub-components: None
Provided interfaces: GatewayUpdates, GWCommunicationMonitoring
Required interfaces: DBDeviceMgmt, Notifiy

7.1.26 GWCommunicationLogic

Responsibility: Handles all actual functionality related to communication. When Gateways synchronise with the Online Service, this component notifies the GatewayMonitor so that it can keep track of Gateway's status.
Super-components: GatewayCommunicationHandler
Sub-components: None

Provided interfaces: ⊸ AppDeviceData, ⊸ AppDeviceMgmt, ⊸ AppInstanceMgmt, ⊸ ApplicationTesting, ⊸ AppMessages, ⊸ DBDeviceMgmt, ⊸ DeviceCommands, ⊸ DeviceData, ⊸ GWAppInstanceMgmt, ⊸ GWCommunicationMonitoring, ⊸ Notifiy, ⊸ TopologyMgmt
Required interfaces: ≺ AppDeviceData, ≺ AppDeviceMgmt, ≺ AppInstanceMgmt, ≺ ApplicationTesting, ≺ AppMessages, ≺ DBDeviceMgmt, ≺ DeviceCommands, ≺ DeviceData, ≺ GatewayUpdates, ≺ GWAppInstanceMgmt, ≺ Notifiy, ≺ TopologyMgmt

7.1.27 InfrastructreOwnerClient

Responsibility: Represents the client used by an infrastructure owner. This is the user's dashboard.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ≺ AccessRights

7.1.28 InfrastructureOwnerFacade

Responsibility: Acts as an access point for InfrastructureOwnerClients and handles all functionality that can be done by infrastructure owners.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ AccessRights, ⊸ Authentication, ⊸ Notifications, ⊸ TopologyMgmt
Required interfaces: ≺ AccessRightsMgmt, ≺ Authentication, ≺ FrontEndAppRequests, ≺ IOMgmt, ≺ Notifications, ≺ TopologyMgmt

7.1.29 InfrastructureOwnerManager

Responsibility: Responsible for all functionality related to infrastructure owners. E.g. looking up the devices they own, retrieving a list of customer organisations that they are associated to, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ IOMgmt
Required interfaces: ≺ AppDeviceMgmt, ≺ DBIODeviceMgmt, ≺ DBIOMgmt

7.1.30 InvoiceManager

Responsibility: Responsible for all functionality related to access rights to invoicing. E.g. creating invoices.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ DeliveryMgmt, ⊸ InvoiceMgmt
Required interfaces: ≺ DBInvoiceMgmt, ≺ InvoiceDeliveryMgmt

7.1.31 Mote

Responsibility: Represents a MicroPnP mote.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ Actuate, ⊸ Config, ⊸ DeviceData, ⊸ RequestData
Required interfaces: ≺ Actuate, ≺ Config, ≺ DeviceData, ≺ DeviceMgmt, ≺ Heartbeat, ≺ RequestData

7.1.32 NotificationHandler

Responsibility: Responsible for generation, storage, and delivery of notifications based on users' preferred communication channel.

Super-components: None
Sub-components: None
Provided interfaces: ⊸ DeliveryMgmt, ⊸ Notifications, ⊸ Notifiy
Required interfaces: ≺ DBNotificationMgmt, ≺ NotificationDeliveryMgmt

7.1.33 OnlineServiceCommunicationHandler

Responsibility: Isolates communication-related concerns between Gateways and the Online Service along with GatewayBroker on the Online Service. Forwards requests from one party to the other and transmits results and possible exceptions.
Keeps track of reachability status of the Online Service.
Can store at least 3 days of pluggable data and application commands before old data has to be overwritten.
Super-components: ▣ Gateway
Sub-components: ▣ OSCommunicationLogic, ▣ OnlineServiceMonitor, ▣ RequestStore
Provided interfaces: ⊸ AppDeviceData, ⊸ AppDeviceMgmt, ⊸ AppInstanceMgmt, ⊸ ApplicationTesting, ⊸ AppMessages, ⊸ DBDeviceMgmt, ⊸ DeviceCommands, ⊸ DeviceData, ⊸ GWAppInstanceMgmt, ⊸ Notifiy, ⊸ OSCommunicationMonitoring, ⊸ TopologyMgmt
Required interfaces: ≺ AppDeviceData, ≺ AppDeviceMgmt, ≺ AppInstanceMgmt, ≺ ApplicationTesting, ≺ AppMessages, ≺ DBDeviceMgmt, ≺ DeviceCommands, ≺ DeviceData, ≺ GWAppInstanceMgmt, ≺ Notifiy, ≺ TopologyMgmt

7.1.34 OnlineServiceCommunicationMonitor

Responsibility: Monitors the communication component on Gateways. If the communication component fails, the monitor tries to restart it. If the failure persists, makes the gateway reboots itself entirely.
Super-components: ▣ Gateway
Sub-components: None
Provided interfaces: None
Required interfaces: ≺ OSCommunicationMonitoring

7.1.35 OnlineServiceMonitor

Responsibility: Monitors the Gateway's connectivity to the Online Service.
If the Online Service or the communication channel has failed, all requests to the Online Service will be stopped and stay stored in the RequestStore. An explicit command for this is not necessary, because the requests in the RequestStore will not be deleted, since no acknowledgements are received anymore from the Online Service.
After the monitor detects that a connection to the Online Service is possible again, it makes the gateway start synchronising again.
When the Online Service is unreachable, application parts running locally on the SIoTIP gateway continue to operate normally.
Super-components: ▣ OnlineServiceCommunicationHandler ▷ ▣ Gateway
Sub-components: None
Provided interfaces: ⊸ OSCommunicationMonitoring, ⊸ OSUpdates
Required interfaces: ≺ OSMonitoring

7.1.36 OSCommunicationLogic

Responsibility: Handles all actual functionality related to communication. When a message comes in from the Online Service, lets the OnlineServiceMonitor know that a message has been received. When an acknowledgements for requests are received from the Online Service, notifies the RequestStore of this so that it can delete the request from its storage.
Super-components: ▣ OnlineServiceCommunicationHandler ▷ ▣ Gateway

Sub-components: None
Provided interfaces: AppDeviceData, AppDeviceMgmt, AppInstanceMgmt, ApplicationTesting, AppMessages, DBDeviceMgmt, DeviceCommands, DeviceData, GWAppInstanceMgmt, Notifiy, OSCommunicationMonitoring, OSMonitoring, TopologyMgmt
Required interfaces: AppDeviceData, AppDeviceMgmt, AppInstanceMgmt, ApplicationTesting, AppMessages, DBDeviceMgmt, DeviceCommands, DeviceData, GWAppInstanceMgmt, Notifiy, OSUpdates, Requests, TopologyMgmt

7.1.37 OtherDataDB

Responsibility: General database for data. For example, storage of data about notifications.
Super-components: None
Sub-components: None
Provided interfaces: Authentication, DBAppMgmt, DBInvoiceMgmt, DBIOMgmt, DBNotificationMgmt, DBSubscriptionMgmt, DBUserMgmt
Required interfaces: None

7.1.38 PluggableDevice

Responsibility: Represents a pluggable device that plugs into MicroPnP motes. Can be a sensor or an actuator.
Sensors produce measurements and send them to the a gateway via a MicroPnP mote.
Actuators have one or more actions associated with them. For example, a switch can "turn on" and "turn off".
Super-components: None
Sub-components: None
Provided interfaces: Actuate, Config, RequestData
Required interfaces: DeviceData

7.1.39 PluggableDeviceDataDB

Responsibility: Database dedicated to pluggable device data only.
Super-components: None
Sub-components: None
Provided interfaces: DBDeviceData
Required interfaces: None

7.1.40 RegisteredUserFacade

Responsibility: Acts as an access point for common functionality between different types of users and handles all functionality that is shared between them.
Super-components: None
Sub-components: None
Provided interfaces: Authentication, Notifications
Required interfaces: Authentication, Notifications

7.1.41 RequestStore

Responsibility: Temporarily stores all pluggable data and issued application commands until they can be deleted (= until an acknowledgement has been received for the request by the Online Service). Passes all requests along to the BrokerLogic and includes a unique requestID in them, so that the Online Service can send an acknowledgement for those messages. Can store at least 3 days of pluggable data before old data has to be overwritten.
Super-components: OnlineServiceCommunicationHandler ▷ Gateway

Sub-components: None
Provided interfaces: -○ `AppMessages`, -○ `DeviceData`, -○ `OSCommunicationMonitoring`, -○ `Requests`
Required interfaces: -◁ `AppMessages`, -◁ `DeviceData`

7.1.42 SessionDB

Responsibility: This database stores sessions.
Super-components: None
Sub-components: None
Provided interfaces: -○ `Sessions`
Required interfaces: None

7.1.43 SubscriptionManager

Responsibility: Responsible for all functionality related to access rights to subscriptions. E.g. retrieving the applications that a customer organisation can subscribe to, creating new subscriptions to ApplicationInstances, etc.
Super-components: None
Sub-components: None
Provided interfaces: -○ `SubscriptionMgmt`
Required interfaces: -◁ `DBSubscriptionMgmt`, -◁ `FrontEndAppRequests`, -◁ `InvoiceMgmt`

7.1.44 SystemAdministratorFacade

Responsibility: Acts as an access point for SystemAdministratorClient and handles all functionality that can be done by application providers.
Super-components: None
Sub-components: None
Provided interfaces: -○ `Authentication`, -○ `Notifications`
Required interfaces: -◁ `Authentication`, -◁ `Notifications`

7.1.45 ThirdPartyInvoicingService

Responsibility: Represents a third party invoicing service.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: None

7.1.46 ThirdPartyNotificationDeliveryService

Responsibility: Represents a third party notification delivery service.
Super-components: None
Sub-components: None
Provided interfaces: -○ `NotificationDeliveryMgmt`
Required interfaces: -◁ `DeliveryMgmt`

7.1.47 TopologyManager

Responsibility: Responsible for all functionality related to topology. E.g. Adding a new mote to the topology of an infrastructure, checking whether or not all devices used by an application are active in the topology, etc.
Super-components: None
Sub-components: None

Provided interfaces: ⊸ `TopologyMgmt`
Required interfaces: ≺ `DBTopologyMgmt`

7.1.48 UnregisteredUserClient

Responsibility: Represents the client used by an unregistered user.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: None

7.1.49 UnregisteredUserFacade

Responsibility: Acts as an access point for unregistered users client and handles all functionality that can be done by unregistered users.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ `Registration`
Required interfaces: ≺ `UserMgmt`

7.1.50 UserManager

Responsibility: Responsible for all functionality related to users and UserRoles. E.g. registering a new end user for a customer organisation, retrieving the user roles that are mandatory for a certain application, etc.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ `RoleMgmt`, ⊸ `UserMgmt`
Required interfaces: ≺ `DBUserMgmt`

7.2 Interfaces

7.2.1 AccessRights

Provided by: `InfrastructureOwnerFacade`
Required by: `InfrastructreOwnerClient`
Operations:

- configureDevice(**PluggableDeviceID** pID)
  - Effect: Returns a map of AccessRights and the IDs of customer organisations that have those AccessRights.
  - Created for: UC9.3 - UC9.4
- List<**PluggableDeviceInfo**> getAccessRights(int infrastructureOwnerID)
  - Effect: Returns a list of PluggableDeviceInfo to display so an infrastructure owner can select a device to configure access rights.
  - Created for: UC9.1
- void updateAccessRights(**PluggableDeviceID** pID, Map<int, **AccessRights**> accessRights)
  - Effect: Updates the access rights on a certain pluggable device for a group of customer organisations.
  - Created for: UC9.6

7.2.2 AccessRightsMgmt

Provided by: `AccessRightsManager`
Required by: `InfrastructureOwnerFacade`
Operations:

- getCustomerOrganisationsRights(**PluggableDeviceID** pID, List<int> custOrgIDs)
  - Effect: Returns a map of AccessRights and the IDs of customer organisations that have those AccessRights on a certain pluggable device.
  - Created for: UC9.4
- void updateAccessRights(**PluggableDeviceID** pID, Map<int, **AccessRights**> accessRights)
  - Effect: Updates the access rights on a certain pluggable device for a group of customer organisations.
  - Created for: UC9.7

7.2.3 Actuate

Provided by: PluggableDevice
Required by: Mote
Operations:
- void sendActuationCommand(string commandName)
  - Effect: Send an actuation command to the actuator. Sending an unknown actuation command has no effect.

7.2.4 Actuate

Provided by: Mote
Required by: DeviceManager, Gateway
Operations:
- void sendActuationCommand(**PluggableDeviceID** pID, string commandName)
  - Effect: Send an actuation command to an actuator. Sending an unknown actuation command has no effect.

7.2.5 AppDeviceData

Provided by: ApplicationContainerManager, ApplicationFrontEndFacade, ApplicationManagementLogic, ApplicationManager, ContainerLogic, GWCommunicationLogic, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler
Required by: ApplicationContainer, ApplicationContainerManager, ApplicationFrontEndClient, ApplicationFrontEndFacade, ApplicationInstance, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler
Operations:
- Map<**PluggableDeviceID**, List<**DeviceData**>> getDataForDevices(List<**PluggableDeviceID**> devices, **DateTime** from, **DateTime** to)
  - Effect: Returns DeviceData of pluggable devices over a specified time period.
  - Created for: UC24.1
- Map<**PluggableDeviceID**, List<**DeviceData**>> getDataForRoom(**RoomTopology** room, **DateTime** from, **DateTime** to)
  - Effect: Returns DeviceData of pluggable devices in a room over a specified time period.
  - Created for: UC24.1
- List<**RoomTopology**> getTopologyOverview(int applicationInstanceID, int customerOrganisationID)
  - Effect: Returns a list of RoomTopology containing devices that an ApplicationInstance has access to.
  - Created for: UC25.1

7.2.6 AppDeviceMgmt

Provided by: DeviceManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler

Required by: ApplicationManagementLogic, ApplicationManager, GWCommunicationLogic, GatewayCommunicationHandler, InfrastructureOwnerManager, OSCommunicationLogic, OnlineServiceCommunicationHandler

Operations:

- bool areEssentialDevicesOperational(int applicationInstanceID)
  - Effect: Returns true if all essential devices for a certain ApplicationInstance are operational.
  - Created for: UC18
- void initialisePluggableDevice(**PluggableDeviceID** pID)
  - Effect: Updates a pluggable device's status to 'active'.
  - Created for: UC8.3
- boolean 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
- void setPluggableDevicesRequirements(int applicationInstanceID, List<**PluggableDeviceInfo**> devices, List<**Relationship**> relationships)
  - Effect: Sets an application's requirements for pluggable devices. e.g. a device needs to be used instead of another device if that other device fails.
  - Created for: Av3: "Application providers can design their applications such that they explicitly require redundancy in the available pluggable devices."

7.2.7 AppInstanceMgmt

Provided by: ApplicationContainer, ApplicationContainerManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler

Required by: ApplicationContainerManager, ApplicationContainerMonitor, ApplicationManagementLogic, ApplicationManager, DeviceManager, GWCommunicationLogic, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler

Operations:

- void activateApplicationInstance(int applicationInstanceID)
  - Effect: Activates an ApplicationInstance.
  - Created for: UC17.3, U2 - easy applications
- void createApplicationInstance(**ApplicationInstance** instance, **ApplicationCode** code)
  - Effect: Effect: Creates a new ApplicationContainer to run an instance of an application.
- void deactivateApplicationInstance(int applicationInstanceID)
  - Effect: Deactivates an ApplicationInstance.
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.
- void destroyApplicationInstance(int applicationInstanceID)
  - Effect: Destroys an ApplicationInstance
  - Created for: UC20.4ii
- boolean isApplicationInstanceAvailable(int applicationInstanceID)
  - Effect: Returns true if a certain ApplicationInstance is available.
  - Created for: UC27.2
- void restartApplicationInstance(int applicationInstanceID)
  - Effect: Restarts an ApplicationInstance
  - Created for: Av2
- void testApplication(int applicationID, **ApplicationCode** code)
  - Effect: Starts the automated application checks for an application.
  - Created for: UC22.12
- void updateApplicationInstances(List<int> applicationInstanceIDs, **ApplicationCode** code)
  - Effect: Updates ApplicationInstance's to a new version.
  - Created for: UC22.15

7.2.8 Applications

Provided by: ApplicationProviderFacade
Required by: None
Operations:

- Map<string, string> getApplicationDetailedStats(int applicationProviderID, int applicationID)
  - Effect: Returns detailed statistics about an application provider's application. If the application is unapproved, the details include the reason.
  - Created for: UC23.3
- List<**Application**> getApplicationsOfApplicationProvider(int applicationProviderID)
  - Effect: Returns a list of applications uploaded by an application provider.
  - Created for: UC22.4
- getApplicationsWithStatsForApplicationProvider(int applicationProviderID)
  - Effect: Returns a list of applications uploaded by the application provider and some information about those applications. e.g. the amount of subscribers
  - Created for: UC23.1
- void uploadApplication(int applicationProviderID, **ApplicationCode** code, string description, Map<string, string> metaData, List<**Version**> versionsToUpdate)
  - Effect: Uploads an Application to the system. 'versionsToUpdate' denotes which versions of the application should be automatically updated. If the application is new, this list is empty.
    This does not make the application available or update the other versions yet. That is done only after testing of the Application is successful.
    Returns the Application's ID.
  - Created for: UC22

7.2.9 ApplicationTesting

Provided by: ApplicationManagementLogic, ApplicationManager, GWCommunicationLogic, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler
Required by: ApplicationContainerManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler
Operations:

- void applicationTestsSuccessful(int applicationID)
  - Effect: Lets the ApplicationManagementLogic know the automatic application checks were successful.
  - Created for: UC22.13
- void applicationTestsUnsuccessful(int applicationID)
  - Effect: Lets the ApplicationManagementLogic know the automatic application checks were not successful.
  - Created for: UC22.13

7.2.10 AppMessages

Provided by: ApplicationContainer, ApplicationContainerManager, ApplicationInstance, ApplicationManagementLogic, ApplicationManager, ContainerLogic, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler, RequestStore
Required by: ApplicationContainer, ApplicationContainerManager, ApplicationInstance, ApplicationManagementLogic, ApplicationManager, ContainerLogic, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler, RequestStore
Operations:

- void applicationInstanceFailed(int applicationInstanceID, int onOnlineService)

  - Effect: Lets the other parts of an application know that one of their parts has become unavailable. This allowd the other parts to possibly run in a degraded mode. Is called by an `ApplicationContainer` after its `ApplicationInstance` has crashed 3 times in a row.
  - Created for: Av1
- void sendCommandToApplicationInstance(int applicationInstanceID, string command, boolean forOnlineService) throws *AppMessageNotAllowedException*
  - Effect: Sends a command to an `ApplicationInstance` that is running on the Online Service or on a `Gateway`.
  - Created for: UC26.1
- void sendMessageToExternalFrontEnd(string hostName, int port, string message) throws *AppMessageNotAllowedException*
  - Effect: Sends a message from an `ApplicationInstance` to an external front end. The `ApplicationInstance` provides the hostname and port.
  - Created for: UC26.1

7.2.11 AppMonitoring

Provided by: `ApplicationContainer`, `ApplicationContainerManager`, `ApplicationInstance`
Required by: `ApplicationContainerManager`, `ApplicationContainerMonitor`, `ContainerLogic`
Operations:
- **Echo** pingApplicationInstance(int applicationInstanceID)
  - Effect: Sends a ping request to an `ApplicationContainer`. The `ApplicationContainer` should respond with an Echo reply. The monitor uses this to check whether or not the container is still available.
  - Created for: The system is able to autonomously detect failures of its individual failing applications.

7.2.12 AppStatus

Provided by: `ApplicationContainerMonitor`
Required by: `ApplicationContainer`, `ApplicationContainerManager`
Operations:
- boolean isApplicationInstanceAvailable(int applicationInstanceID)
  - Effect: Returns true if a certain `ApplicationInstance` is available.
  - Created for: UC27.2
- void updateApplicationActive(int applicationInstanceID)
  - Effect: Updates an `ApplicationInstance`'s status to 'active'.
- void updateApplicationInstanceSuspended(int applicationInstanceID)
  - Effect: Updates an `ApplicationInstance`'s status to 'suspended'.
- void updateApplicationStatusUnavailable(int applicationInstanceID)
  - Effect: Effect: Notifies the `ApplicationContainerMonitor` that an `ApplicationInstance` has crashed.
  - Created for: Av2 - The system is able to autonomously detect failing applications

7.2.13 Authentication

Provided by: `OtherDataDB`
Required by: `AuthenticationManager`
Operations:
- boolean verifyAuthenticationCredentials(Map<string, string> credentials)
  - Effect: Returns true if the provided authentication credentials are correct.
  - Created for: UC28.2

7.2.14 Authentication

Provided by: ApplicationProviderFacade, AuthenticationManager, CustomerOrganisationFacade, InfrastructureOwnerFacade, RegisteredUserFacade, SystemAdministratorFacade
Required by: ApplicationProviderFacade, CustomerOrganisationFacade, InfrastructureOwnerFacade, RegisteredUserFacade, SystemAdministratorFacade
Operations:

- boolean doesSessionExist(string sessionID)
  - Effect: Returns true if a session with the given sessionID exists.
  - Created for: UC28, UC29
- string login(Map<string, string> credentials)
  - Effect: Verifies the provided authentication credentials. If everything is correct, logs the user in, creates a unique session, and returns the session's ID.
    Returns an empty string if the provided credentials are incorrect.
  - Created for: UC28.1-2
- boolean logout(string sessionID)
  - Effect: If a session with the given sessionID exists, logs the user out and returns true.
  - Created for: UC29.1

7.2.15 Commands

Provided by: ApplicationContainerManager, DeviceCommandConstructor
Required by: ApplicationContainerManager, ApplicationManagementLogic
Operations:

- void sendActuationCommand(List<**PluggableDeviceID**> device, string commandName) throws *UnknownCommandException*
  - Effect: Sends a command for a list of actuators to DeviceCommandConstructor for construction of actuation command messages according to the specific formatting syntax for the given actuators. The given command 'commandName' is a command that application developers use for a group of devices.
  - Created for: UC12 - commands from applications
- void verifyAndConstructConfigurationsForDevice(**PluggableDeviceID** pID, Map<string, string> config) throws *UnknownConfigurationParameterException*
  - Effect: Verifies the configuration parameters for a pluggable device. If the parameters have been successfuly verified, constructs a reconfiguration command according to the specific formatting syntax for the pluggable device and sends it to the DeviceManager if everything is correct.
  - Created for: UC13.2-3

7.2.16 Config

Provided by: PluggableDevice
Required by: Mote
Operations:

- Map<String, String> getConfig()
  - Effect: Returns the current configuration of a pluggable device as a parameter-value map.
- boolean setConfig(void Map<string, string>)
  - Effect: Set the given configuration parameters of the pluggable device to the given values. Setting unknown parameters on a pluggable device (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

7.2.17 Config

Provided by: Mote

Required by: DeviceManager, Gateway
Operations:

- Map<String, String> getConfig(**PluggableDeviceID** pID)
  - Effect: Returns the current configuration of a pluggable device as a parameter-value map.
- boolean setConfig(**PluggableDeviceID** pID, Map<string, string> config)
  - Effect: Set the given configuration parameters of the pluggable device to the given values. Setting unknown parameters on a pluggable device (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

7.2.18 DataConversion

Provided by: ContainerLogic, DeviceDataConverter
Required by: ApplicationContainer, ApplicationInstance
Operations:

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

7.2.19 DBAccessRightsMgmt

Provided by: DeviceDB
Required by: AccessRightsManager
Operations:

- getCustomerOrganisationsRights(**PluggableDeviceID** pID, List<int> custOrgIDs)
  - Effect: Returns a map of AccessRights and the IDs of customer organisations that have those AccessRights on a certain pluggable device.
  - Created for: UC9.4
- void updateAccessRights(**PluggableDeviceID** id, Map<int, **AccessRights**> accessRights)
  - Effect: Updates the access rights on a certain pluggable device for a group of customer organisations.
  - Created for: UC9.7

7.2.20 DBAppDeviceMgmt

Provided by: DeviceDB
Required by: ApplicationManager, DeviceCommandConstructor
Operations:

- Map<**PluggableDeviceID**, string> getFormattingSyntaxForDevices(List<**PluggableDeviceID**> devices)
  - Effect: Effect: Retrieves the specific formatting syntax for a list of pluggable devices.
  - Created for: UC12.2-3
- Map<**PluggableDeviceID**, **GatewayInfo**> getGatewaysForDevices(List<**PluggableDeviceID**> devices)
  - Effect: Returns a map of PluggableDevices with the gateways they are connected to.
  - Created for: UC12.2
- string getPossibleConfigurationParametersForDevice(**PluggableDeviceID** pID)
  - Effect: Retrieves the possible configuration parameters for a pluggable device.
  - Created for: UC13.2

7.2.21 DBAppMgmt

Provided by: OtherDataDB
Required by: ApplicationManagementLogic, ApplicationManager
Operations:

- void activateApplicationInstance(int applicationInstanceID)
  - Effect: Sets an `ApplicationInstance`'s status in the database to 'active'.
  - Created for: UC17.4, U2 - easy applications
- int createApplicationInstance(int custOrgID, int applicationID)
  - Effect: Creates a new `ApplicationInstance` for an application for a customer organisation and returns its id. The `ApplicationInstance` gets status 'inactive', since it will be activated after a subscription is completed.
  - Created for: UC19.4, U2 - easy applications
- int createNewApplication(int applicationProviderID, **ApplicationCode** code, string description, Map<string, string> metaData, List<**Version**> versionsToUpdate)
  - Effect: Uploads an Application to the system. 'versionsToUpdate' denotes which versions of the application should be automatically updated. If the application is new, this list is empty.
    This does not make the application available or update the other versions yet. That is done only after testing of the Application is successful.
    Returns the Application's ID.
  - Created for: UC22
- void deactivateApplicationInstance(int applicationInstanceID)
  - Effect: Sets an `ApplicationInstance`'s status in the database to 'active'.
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.
- **ApplicationCode** getApplicationCode(int applicationID)
  - Effect: Returns an application's code.
  - Created for: UC22.15
- Map<string, string> getApplicationDetailedStats(int applicationID)
  - Effect: Returns detailed statistics about an application provider's application. If the application is unapproved, the details include the reason.
  - Created for: UC23.4
- **GatewayInfo** getApplicationInstanceGateway(int applicationInstanceID)
  - Effect: Returns information about a gateway that an application instance is running on.
  - Created for: Av2 - Need to know which gateways to send messages to for e.g. application commands.
- List<int> getApplicationInstancesToUpdate(int applicationID)
  - Effect: Returns a list of application instances that would need to be updated because of a certain updated application.
  - Created for: UC22.15
- List<**Application**> getApplications()
  - Effect: Returns a list of applications in the system.
  - Created for: UC19.2, U2 - easy applications
- List<int> getApplicationsForDevice()
  - Effect: Returns a list of applications that can use the device with id "pID".
  - Created for: UC11: the system looks up the list of applications that use the pluggable device
- List<**Application**> getApplicationsOfApplicationProvider(int applicationProviderID)
  - Effect: Returns a list of applications uploaded by an application provider.
  - Created for: UC22.4
- getApplicationsWithStatsForApplicationProvider(int applicationProviderID)
  - Effect: Returns a list of applications uploaded by the application provider and some information about those applications. e.g. the amount of subscribers
  - Created for: UC23.2
- List<**PluggableDeviceID**> getDevicesForApplication(int applicationInstanceID)
  - Effect: Returns a list of PluggableDeviceID of pluggable devices that an `ApplicationInstance` can use.
  - Created for: UC17.2, U2 - easy applications
- string getInstallationInstructions(int applicationID)
  - Effect: Returns the installation instructions of a certain application. If there are no installation instructions set, returns an empty string.

  - Created for: UC17.6, U2 - easy applications
- List<**RoomTopology**> getNecessaryDevicesAndTopologyConfigurations(int applicationID)
  - Effect: Returns a list of RoomTopology which is a minimal requirement for a certain application to run. This can used to display the requirements to a user or to check if requirements are fulfilled.
  - Created for: UC19.5, U2 - easy applications
- void softDeleteApplicationInstance(int applicationInstanceID)
  - Effect: Updates an `ApplicationInstance`'s status in the database to reflect that the customer organisation has unsubscribed from it.
  - Created for: UC20.4ii
- void updateApplicationAvailable(int applicationID)
  - Effect: Makes an application in the system available in the application store.
  - Created for: UC22.14
- void updateApplicationDevicesSettings(int applicationInstanceID, List<**PluggableDeviceID**> devices, List<**Relationship**> relationships)
  - Effect: Updates an `ApplicationInstance`'s device settings. This includes which devices the instance can use and which relationships exist between those devices.
  - Created for: UC19.6, U2 - easy applications
- void updateApplicationInstance(int applicationInstanceID)
  - Effect: Updates an `ApplicationInstance` in the database (e.g. change state to 'inactive').
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.
- void updateApplicationInstanceGateway(int applicationInstanceID, int gatewayID)
  - Effect: Updates the gateway on which an application instance is running.
  - Created for: Av2 - Need to know which gateways to send messages to for e.g. application commands.
- void updateApplicationInstancesOfApplication(int applicationID)
  - Effect: Updates the version of a list of application instances that need to be updated because of a certain application update. The application instances to update is found in the data provided when the application was uploaded.
  - Created for: UC22.15
- void updateCriticality(int applicationInstanceID, int isCritical)
  - Effect: Updates the criticality of an `ApplicationInstance`.
  - Created for: UC19.11, U2 - easy applications
- void updateSubscription(**Subscription** subscription)
  - Effect: Updates a subscription in the database (e.g. change state to 'disabled').
  - Created for: UC18

7.2.22 DBDeviceData

Provided by: `PluggableDeviceDataDB`
Required by: `DeviceDataScheduler`
Operations:

- 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
- getDataForDevices(List<**PluggableDeviceID**> devices, **DateTime** from, **DateTime** to)
  - Effect: Returns DeviceData of pluggable devices over a specified time period.
  - Created for: UC24.2
- 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

7.2.23 DBDeviceMgmt

Provided by: `DeviceDB`, `GWCommunicationLogic`, `GatewayCommunicationHandler`, `OSCommunicationLogic`,

OnlineServiceCommunicationHandler

Required by: DeviceManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, GatewayCommunicationMonitor, GatewayMonitor, OSCommunicationLogic, OnlineServiceCommunicationHand

Operations:

- void addDevice(**PluggableDeviceID** pID, **PluggableDeviceType** type, Map<string, string> configurations, int moteID)
  - Effect: Adds a new pluggable device in the DeviceDB and adds a reference to a mote. The device's status is 'uninitialised' by default and it's current configurations (which are now the default configurations) are stored as well. If the device already exists, removes the data first (in case the device is plugged into a different mote or on a different network).
  - Created for: UC6.3, U2 - easy pluggable device installation
- int addMote(**MoteInfo** mote, int gatewayID, **IPAddress** moteIPAddress)
  - Effect: Adds a new mote in the DeviceDB along with an IP address and a reference to a gateway. Returns the DB id for the mote.
  - Created for: UC4.3, U2 - easy mote installation
- void deactivateMote(int moteID)
  - Effect: Changes the status of a mote to 'inactive'.
  - Created for: UC5.2a
- void deactivatePluggableDevice(**PluggableDeviceID** pID)
  - Effect: Changes the status of a pluggable device to 'inactive'.
  - Created for: UC7.3a
- void deactivatePluggableDevices(List<**PluggableDeviceID**> pIDs)
  - Effect: Changes the status of a list of pluggable devices to 'inactive'.
  - Created for: UC5.2b
- Map<string, string> getConfigDB(**PluggableDeviceID** pID)
  - Effect: Gets the last set configurations of a pluggable device from the DeviceDB.
  - Created for: UC6.3 - reintroduced device
- double getPercentageOfGatewaysUnreachable()
  - Effect: Returns the percentage of deployed gateways that have status 'unreachable'.
  - Created for: Av1 - A SIoTIP system administrator should be notified of a simultaneous outage of more than 1% of the registered gateways.
- Map<string, string> getPluggableDeviceData(**PluggableDeviceID** pID)
  - Effect: Effect: Returns all data about a pluggable device in the DeviceDB, such as status, type information, specific formatting syntax (for applications), etc.
  - Created for: UC12 - construction of commands, UC13 - verification of configuration parameters
- void reactivateDevice(**PluggableDeviceID** pID)
  - Effect: Changes the status of a pluggable device to 'active'.
  - Created for: UC6.3 - reintroduced device
- void reactivateMote(int moteID)
  - Effect: Changes the status of the mote with DB id 'moteID' to 'active'.
  - Created for: U2 - Reintroducing a previously known mote should not require any con guration.
- void registerGateway(int gatewayID, **IPAddress** address)
  - Effect: Sets a gateway's status to 'active' and updates its IP address.
  - Created for: U2 - gateway installation
- void updateGatewayStatus(int gatewayID)
  - Effect: Changes a gateway's status.
  - Created for: Av1 - Connectivity of gateways

7.2.24 DBInvoiceMgmt

Provided by: OtherDataDB
Required by: InvoiceManager
Operations:

- void createInvoice(**Invoice** invoice)
  - Effect: Stores a new invoice in the database.
  - Created for: UC20.4iii
- **Invoice** getInvoicesForCustomerOrganisation(int customerOrganisationID)
  - Effect: Returns outstanding invoices for a customer organisation.
  - Created for: UC21.1
- void markActivatedApplication(int applicationInstanceID, int custOrgID, **DateTime** date)
  - Effect: Updates an `ApplicationInstance`'s billing information: marks the start of a billing period.
  - Created for: UC17.4, U2 - easy applications

7.2.25 DBIODeviceMgmt

Provided by: `DeviceDB`
Required by: `InfrastructureOwnerManager`
Operations:
- List<**PluggableDeviceInfo**> getDevices(int infrastructureOwnerID)
  - Effect: Returns a list of PluggableDeviceInfo of devices owned by an infrastructure owner.
  - Created for: UC9.2
- void initialisePluggableDevice(**PluggableDeviceID** pID)
  - Effect: Updates a pluggable device's status to 'active'.
  - Created for: UC8.3

7.2.26 DBIOMgmt

Provided by: `OtherDataDB`
Required by: `InfrastructureOwnerManager`
Operations:
- List<int> getCustomerOrganisations(int infrastructureOwnerID)
  - Effect: Returns a list of IDs of all customer organisations associated with an infrastructure owner.
  - Created for: UC9.4

7.2.27 DBNotificationMgmt

Provided by: `OtherDataDB`
Required by: `NotificationHandler`
Operations:
- int createNotification(**Notification** notification)
  - Effect: Stores a new notification entry in the database. Returns the id of the new notification.
  - Created for: UC15, Av3: notifications
- **Notification** getDetailedNotification(int userID, int notificationID)
  - Effect: Returns detailed information about a notification for a certain registered user.
  - Created for: UC16.4
- List<**Notification**> getNotifications(int userID)
  - Effect: Returns a list of Notification for a certain registered user. Not all details in the Notification objects are filled in.
  - Created for: UC16.2
- int lookupNotificationChannelForUser(int userID)
  - Effect: Returns the id of the type of communication channel a user prefers.
  - Created for: UC15
- void setNotificationRead(int notificationID)
  - Effect: Marks a notification as 'read'.
  - Created for: UC16.5
- int updateNotification(**Notification** notification)
  - Effect: Updates an existing notification (e.g. change status to "sent").

- Created for: UC15

7.2.28 DBSubscriptionMgmt

Provided by: `OtherDataDB`
Required by: `SubscriptionManager`
Operations:

- void createSubscription(int custOrgID, int applicationInstanceID)
  - Effect: Creates a subscription for a customer organisation to an `ApplicationInstance`. If the customer organisation is already subscribed to an older version of the the application, then the organisation is unsubscribed from that earlier version.
  - Created for: UC19.12-13, U2 - easy applications
- List<**Subscription**> getSubscriptions(int custOrgID)
  - Effect: Returns a list of subscriptions a customer organisation has.
  - Created for: UC19.2, U2 - easy applications
- List<**Subscription**> getSubscriptions(int customerOrganisationID)
  - Effect: Returns a list of the subscriptions of a customer organisation.
  - Created for: UC20.2
- void softDeleteSubscription(int subscriptionID)
  - Effect: Ends a subsription.
  - Created for: UC20.4i

7.2.29 DBTopologyMgmt

Provided by: `DeviceDB`
Required by: `TopologyManager`
Operations:

- void addDevice(**PluggableDeviceID** pID, int moteID)
  - Effect: Adds a new pluggable device to the topology of the infrastructure owner and links it to a mote. The device gets the mote's location by default. If the device is already linked to another mote, overwrites that link.
  - Created for: UC6.3, U2 - easy pluggable device installation
- void addMote(int moteID, int infrastructureOwnerID, int gatewayID)
  - Effect: Adds a new mote to a topology of an infrastructure owner. The mote is linked to a gateway and gets status 'unplaced' by default.
  - Created for: UC4.3, U2 - easy mote installation
- boolean arePluggableDevicesPlaced(List<**PluggableDeviceID**> devices)
  - Effect: Returns true if all pluggable devices in the given list have status 'placed' in the topology.
  - Created for: UC17.2, U2 - easy applications
- void deactivateMote(int moteID)
  - Effect: hanges the status of a mote in the topology to 'inactive'.
  - Created for: UC5.2a
- void deactivatePluggableDevice(**PluggableDeviceID** pID)
  - Effect: Changes the status of a pluggable device in the topology to 'inactive'.
  - Created for: UC7.3b
- void deactivatePluggableDevices(List<**PluggableDeviceID**> pIDs)
  - Effect: Changes the status of a list of pluggable devices in the topology to 'inactive'.
  - Created for: UC5.2b
- List<**DeviceStatus**> getStatusDetailsForGateway(int gatewayID)
  - Effect: Returns details about the status of a certain gateway. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**DeviceStatus**> getStatusDetailsForMote(int moteID)

  - Effect: Returns details about the status of a certain mote. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**DeviceStatus**> getStatusDetailsForPluggableDevice(**PluggableDeviceID** pID)
  - Effect: Returns details about the status of a certain pluggable device. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**RoomTopology**> getTopologyOfCustomerOrganisation(int custOrgID)
  - Effect: Returns a list of RoomTopology associated to a customer organisation.
  - Created for: UC19.5, U2 - easy applications
- List<**RoomTopology**> getTopologyOfInfrastructureOwner(int infrastructureOwnerID)
  - Effect: Returns the topology owned by an infrastructure owner.
  - Created for: UC10.2
- List<**MoteInfo**> getUnplacedMotes(int infrastructureOwnerID)
  - Effect: Returns info about unplaced motes that are owned by an infrastructure owner
  - Created for: UC10.2
- List<**PluggableDeviceInfo**> getUnplacedPluggableDevices(int infrastructureOwnerID)
  - Effect: Returns info about unplaced motes that are owned by an infrastructure owner
  - Created for: UC10.2
- void reactivateDevice(**PluggableDeviceID** id)
  - Effect: Changes the status of a pluggable device in the topology to 'placed'.
  - Created for: UC6.3 - reintroduced device
- void reactivateMote(int moteID)
  - Effect: Changes the status of the mote in the topology to 'placed'. The location of the mote is unchanged, it has already been set.
  - Created for: U2 - Reintroducing a previously known mote should not require any con guration.
- void updateTopology(int infrastructureOwnerID, List<**RoomTopology**> updatedTopology)
  - Effect: Updates the topology of an infrastructure owner. The 'updatedTopology' is the new one submitted by the infrastructure owner.
  - Created for: UC10.5a

7.2.30 DBUserMgmt

Provided by: `OtherDataDB`
Required by: `UserManager`
Operations:

- boolean areMandatoryUserRolesAssigned(int applicationInstanceID)
  - Effect: Returns true if all mandatory UserRoles for the application have been assigned to users. Finds the relevant customer organisations through the `ApplicationInstance`.
  - Created for: UC17.1, U2 - easy applications
- boolean canCustomerOrganisationRegister(int infrastructureOwnerID, string hash)
  - Effect: Returns true if the combination of infrastructure owner and hash exists in the system. This means that the infrastructure owner has allowed for a certain customer organistaion to register.
  - Created for: UC1.1
- boolean confirmEmail(string emailAddress, string randomString)
  - Effect: Return true if the given e-mail and randomString match in the database.
    Returns false if user did not click the link in the received e-mail within 30 minutes.
  - Created for: UC1.5
- int createCustomerOrganisation(Map<string, string> data) throws *AddressInvalidException*, *UserNameAlreadyExistsException*, *PaymentInformationInvalidException*
  - Effect: Creates a customer organisation and returns its id. Links the customer organisation to an infrastructure owner by using the information stored in 'createEmailAddressForRegistration'.
  - Created for: UC1.8

- boolean createEmailAddressForRegistration(string emailAddress, int infrastructureOwnerID) throws *EmailAddressInvalidException*, *EmailAddressAlreadyExistsException*
  - Effect: If the given e-mail address is valid and not yet in the database, saves the e-mail address in the database, links it to the infrastructure owner and a unique random string for the confirmation e-mail, and the current timestamp. Afterwards, returns true.
    Otherwise, returns false.
  - Created for: UC1.4
- boolean createEndUser(int customerOrganisationID, Map<string, string> data) throws *EmailAddressAlreadyExistsException*, *EmailAddressInvalidException*, *EndUserAlreadyExistsException*, *PhoneNumberInvalidException*
  - Effect: Verifies provided data for a new end-user, checks that the end-user does not exist in the system already, and then stores the end-user information and associates it with the customer organisation. Returns true if everything succeeded.
  - Created for: UC2.3-4
- List<**User**> getEndUsers(int custOrgID)
  - Effect: Returns a list of Users which are associated to a customer organisation.
  - Created for: UC19.8, U2 - easy applications
- List<**UserRole**> getMandatoryUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles that which need to be assigned in order for an `ApplicationInstance` to run.
  - Created for: UC19.7, U2 - easy applications
- List<**UserRole**> getOptionalUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles which can optionally be assigned for an `ApplicationInstance`.
  - Created for: UC19.7, U2 - easy applications
- List<**User**> getUsersWithRoles(int applicationInstanceID)
  - Effect: Returns a list of Users associated to an `ApplicationInstance` that were assigned UserRoles.
  - Created for: UC17.6, U2 - easy applications
- void softDeleteEndUser(int customerOrganisationID, int userID)
  - Effect: Marks the profile of an end-user as 'inactive'.
  - Created for UC3.4
- void updateUserRoles(int applicationInstanceID, Map<int, int> usersAndRoles)
  - Effect: Updates the UserRoles assigned to Users for a certain `ApplicationInstance`. 'usersAndRoles' maps User IDs to UserRole IDs.
  - Created for: UC19.9, U2 - easy applications

7.2.31 DeliveryMgmt

Provided by: `NotificationHandler`
Required by: `ThirdPartyNotificationDeliveryService`
Operations:
- void acknowledgement(int notificationID)
  - Effect: Sends an acknowledgement to the system for a certain notification to denote that a notification has been received.
  - Created for: UC15

7.2.32 DeliveryMgmt

Provided by: `InvoiceManager`
Required by: None
Operations:
- void acknowledgement(int invoiceID)
  - Effect: Sends an acknowledgement to the system for a certain invoice to denote that the invoice has been received.

7.2.33 DeviceCommands

Provided by: DeviceManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler

Required by: ApplicationManager, DeviceCommandConstructor, GWCommunicationLogic, GatewayCommunicationH OSCommunicationLogic, OnlineServiceCommunicationHandler

Operations:

- Map<**PluggableDeviceID**, string> getFormattingSyntaxForDevices(List<**PluggableDeviceID**> devices)
  - Effect: Retrieves the specific formatting syntax for a list of pluggable devices from the DeviceManager. Used by the DeviceCommandConstructor on Gateways.
  - Created for: UC12 - commands from applications on gateways
- string getPossibleConfigurationParametersForDevice(**PluggableDeviceID** pID)
  - Effect: Retrieves the possible configuration parameters for a pluggable device. Used by the DeviceCommandConstructor on Gateways.
  - Created for: UC13 - configuration commands from applications on gateways
- void sendActuationCommand(Map<**PluggableDeviceID**, string> commandsForDevices)
  - Effect: Sends correctly constructed actuation commands for a group of actuators to the DeviceManager on a Gateway.
  - Created for: UC12 - send command message to the intended actuators
- void setConfiguration(**PluggableDeviceID** pID, Map<string, string> config)
  - Effect: Sets configuration parameters of a pluggable device. The DeviceManager first determines whether the pluggable device needs to be reconfigured. To do this, it checks the data it has about configurations set by other applications on the pluggable device. If the device can be reconfigured, then the configuration command is propagated to the pluggable device.
  - Created for: UC13.3

7.2.34 DeviceCommands

Provided by: ApplicationContainerManager, ApplicationFrontEndFacade, ApplicationManagementLogic, ApplicationManager, ContainerLogic

Required by: ApplicationContainer, ApplicationFrontEndClient, ApplicationFrontEndFacade, ApplicationInstance

Operations:

- void sendActuationCommand(List<**PluggableDeviceID**> devices, string command)
  - Effect: Sends a command for a list of actuators for construction of the actuation command messages.
  - Created for: UC12.1
- void setConfiguration(**PluggableDeviceID** pID, Map<string, string> config)
  - Effect: Sends a configuration command for a pluggable device for construction of the actual command messages.
  - Created for: UC13.1

7.2.35 DeviceData

Provided by: DeviceDataScheduler, GWCommunicationLogic, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler, RequestStore

Required by: DeviceManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler, RequestStore

Operations:

- 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

7.2.36 DeviceData

Provided by: DeviceManager, Gateway, Mote

Required by: Mote, PluggableDevice
Operations:
- void rcvData(**PluggableDeviceID** pID, **DeviceData** data)
  - Effect: Provides pluggable device data to the gateway (Initiated by the device).
- void rcvDataCallback(**PluggableDeviceID** pID, **DeviceData** data, int requestID)
  - Effect: Provides device data to the gateway (Callback of getDataAsync).

7.2.37 DeviceMgmt

Provided by: DeviceManager, Gateway
Required by: Mote
Operations:
- void pluggableDevicePluggedIn(**MoteInfo** mInfo, **PluggableDeviceID** pID, **PluggableDeviceType** type)
  - Effect: Notify the gateway that a new PluggableDevice of the given type is connected to the mote.
- void pluggableDeviceRemoved(**PluggableDeviceID** pID)
  - Effect: Notify the gateway that a PluggableDevice is removed.

7.2.38 ForwardData

Provided by: ApplicationManagementLogic, ApplicationManager
Required by: DeviceDataScheduler
Operations:
- List<int> getApplicationsForDevice(**PluggableDeviceID** pID)
  - Effect: Returns a list of application instances that can use the device with id "pID".
  - Created for: UC11: the system looks up the list of applications that use the pluggable device
- 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

7.2.39 FrontEndAppRequests

Provided by: ApplicationManagementLogic, ApplicationManager
Required by: ApplicationProviderFacade, CustomerOrganisationFacade, InfrastructureOwnerFacade, SubscriptionManager
Operations:
- void activateApplicationInstance(int applicationInstanceID)
  - Effect: Checks and activates an ApplicationInstance (UC17).
  - Created for: UC19.14, U2 - easy applications
- void checkApplicationsForActivationForCustomerOrganisations(List<int> custOrgIDs)
  - Effect: Checks and activates 'inactive' ApplicationInstances which can now execute again for a list of customer organisations.
  - Created for: UC9.7
- void checkApplicationsForActivationForInfrastructureOwner(int infrastructureOwnerID)
  - Effect: Checks and activates ApplicationInstances which can now execute (again). The applications checked are those that are subscribed to by customers organisations associated to the given infrastructure owner.
  - Created for: UC10.5b
- void checkApplicationsForDeactivationForCustomerOrganisations(List<int> custOrgIDs)
  - Effect: Checks for ApplicationInstances that require deactivation for a list of customer organisations.
  - Created for: UC9.7
- void checkApplicationsForDectivationForInfrastructureOwner(int infrastructureOwnerID)
  - Effect: Checks and deactivates ApplicationInstances which require deactivation. The applications checked are those that are subscribed to by customers organisations associated to the given infrastructure owner.

  - Created for: UC10.5c
- int createApplicationInstance(int custOrgID, int applicationID)
  - Effect: Creates a new `ApplicationInstance` for an application for a customer organisation and returns its id. The `ApplicationInstance` gets status 'inactive', since it will be activated after a subscription is completed.
  - Created for: UC19.4, U2 - easy applications
- void destroyApplicationInstance(int applicationInstanceID)
  - Effect: Destroys an `ApplicationInstance`
  - Created for: UC20.4ii
- Map<string, string> getApplicationDetailedStats(int applicationID)
  - Effect: Returns detailed statistics about an application provider's application. If the application is unapproved, the details include the reason.
  - Created for: UC23.4
- List<**Application**> getApplications()
  - Effect: Returns a list of applications in the system.
  - Created for: UC19.2, U2 - easy applications
- List<**Application**> getApplicationsOfApplicationProvider(int applicationProviderID)
  - Effect: Returns a list of applications uploaded by an application provider.
  - Created for: UC22.4
- getApplicationsWithStatsForApplicationProvider(int applicationProviderID)
  - Effect: Returns a list of applications uploaded by the application provider and some information about those applications. e.g. the amount of subscribers
  - Created for: UC23.2
- List<**RoomTopology**> getNecessaryDevicesAndTopologyConfigurations(int applicationID)
  - Effect: Returns a list of RoomTopology which is a minimal requirement for a certain application to run. This can used to display the requirements to a user or to check if requirements are fulfilled.
  - Created for: UC19.5, U2 - easy applications
- void softDeleteApplicationInstance(int applicationInstanceID)
  - Effect: Updates an `ApplicationInstance`'s status in the database to reflect that the customer organisation has unsubscribed from it.
  - Created for: UC20.4ii
- void updateApplicationDevicesSettings(int applicationInstanceID, List<**PluggableDeviceID**> devices, List<**Relationship**> relationships)
  - Effect: Updates an `ApplicationInstance`'s device settings. This includes which devices the instance can use and which relationships exist between those devices.
  - Created for: UC19.6, U2 - easy applications
- void updateCriticality(int applicationInstanceID, boolean isCritical)
  - Effect: Updates the criticality of an `ApplicationInstance`.
  - Created for: UC19.11, U2 - easy applications
- void uploadApplication(int applicationProviderID, **ApplicationCode** code, string description, Map<string, string> metaData, List<**Version**> versionsToUpdate)
  - Effect: Uploads an Application to the system. 'versionsToUpdate' denotes which versions of the application should be automatically updated. If the application is new, this list is empty.
    This does not make the application available or update the other versions yet. That is done only after testing of the Application is successful.
    Returns the Application's ID.
  - Created for: UC22

7.2.40 GatewayUpdates

Provided by: `GatewayMonitor`
Required by: `GWCommunicationLogic`
Operations:

- void gatewayUpdate(int gatewayID, **DateTime** time)

- Effect: Lets the `GatewayMonitor` know that data has been received from a certain `Gateway`. Is called after a synchronisation of the gateway. When 3 consecutive expected synchronisations that do not arrive within 1 minute of their expected arrival time, a gateway outage is detected by the monitor.
- Created for: Av1 - The Online Service is able to detect that a SIoTIP gateway is not sending data anymore based on the expected synchronisation interval.

7.2.41 GWAppInstanceMgmt

Provided by: `ApplicationManagementLogic`, `ApplicationManager`, `GWCommunicationLogic`, `GatewayCommunicationHandler`, `OSCommunicationLogic`, `OnlineServiceCommunicationHandler`
Required by: `DeviceManager`, `GWCommunicationLogic`, `Gateway`, `GatewayCommunicationHandler`, `OSCommunicationLogic`, `OnlineServiceCommunicationHandler`
Operations:

- void activateApplicationInstance(int applicationInstanceID)
  - Effect: Activates an `ApplicationInstance`.
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.
- void checkApplicationsForActivationForInfrastructureOwner(int infrastructureOwnerID)
  - Effect: Checks and activates ApplicationInstances which can now execute (again). The applications checked are those that are subscribed to by customers organisations associated to the given infrastructure owner.
  - Created for: UC17, UC6.3 - reintroduced device
- void checkApplicationsForDeactivationForPluggableDevice(**PluggableDeviceID** pID)
  - Effect: Checks and possibly deactivates ApplicationInstances which are affected by the unavailability of a pluggable device (Include: UC18: Check and deactivate applications).
  - Created for: UC7.3c
- void checkApplicationsForDeactivationForPluggableDevices(List<**PluggableDeviceID**> pIDs)
  - Effect: Checks and possibly deactivates ApplicationInstances which are affected by the unavailability of a list of pluggable devices (Include: UC18: Check and deactivate applications).
  - Created for: UC5.2c
- void deactivateApplicationInstance(int applicationInstanceID)
  - Effect: Deactivates an `ApplicationInstance`.
  - Created for: UC18, Av3: automatic suspension/reactivation of applications.

7.2.42 GWCommunicationMonitoring

Provided by: `GWCommunicationLogic`, `GatewayCommunicationHandler`, `GatewayMonitor`
Required by: `GatewayCommunicationMonitor`
Operations:

- **Echo** ping()
  - Effect: Sends a ping request to a component. The component should respond with an Echo reply. The monitor uses this to check whether or not the component is still available.
  - Created for: Av1 - The Online Service is able to autonomously detect failures of its individual internal communication components.

7.2.43 Heartbeat

Provided by: `DeviceManager`, `Gateway`
Required by: `Mote`
Operations:

- void heartbeat(Map<string, string> moteInfo, List<Tuple<**PluggableDeviceID**, **PluggableDeviceType**>> pds)
  - Effect: Sends a heartbeat from a mote to a gateway, including a list of the pluggable devices and their device types (i.e. those currently plugged into the mote)
  - Created for: Given constraint, UC14, Av3: failure detection

7.2.44 InvoiceDeliveryMgmt

Provided by: None
Required by: InvoiceManager
Operations:
- void sendInvoice(Map<string, string> invoiceData)
  - Effect: Sends an invoice using a third party invoicing service.
  - Created for: UC21.2

7.2.45 InvoiceMgmt

Provided by: InvoiceManager
Required by: ApplicationManagementLogic, ApplicationManager, SubscriptionManager
Operations:
- void constructInvoiceForApplicationInstance(int applicationInstanceID, int customerOrganisationID)
  - Effect: Constructs an invoice for outstanding payments for an ApplicationIntance
  - Created for: UC20.4iii
- void markActivatedApplication(int applicationInstanceID, int custOrgID, **DateTime** date)
  - Effect: Updates an ApplicationInstance's billing information: marks the start of a billing period.
  - Created for: UC17.4, U2 - easy applications

7.2.46 IOMgmt

Provided by: InfrastructureOwnerManager
Required by: InfrastructureOwnerFacade
Operations:
- List<int> getCustomerOrganisations(int infrastructureOwnerID)
  - Effect: Returns a list of IDs of all customer organisations associated with an infrastructure owner.
  - Created for: UC9.4
- List<**PluggableDeviceInfo**> getDevices(int infrastructureOwnerID)
  - Effect: Returns a list of PluggableDeviceInfo of devices owned by an infrastructure owner.
  - Created for: UC9.2
- void initialisePluggableDevice(**PluggableDeviceID** pID, int gatewayID)
  - Effect: Updates a pluggable device's status to 'active'.
  - Created for: UC8.3

7.2.47 Monitoring

Provided by: ApplicationContainerManager, ApplicationContainerMonitor, DeviceCommandConstructor, DeviceDataConverter
Required by: ApplicationExecutionSubsystemMonitor
Operations:
- **Echo** ping()
  - Effect: Sends a ping request to a component. The component should respond with an Echo reply. The monitor uses this to check whether or not the component is still available.
  - Created for: The system is able to autonomously detect failures of its individual application execution components
- void restart()
  - Effect: Restarts the monitored component
  - Created for: Av2

7.2.48 NotificationDeliveryMgmt

Provided by: ThirdPartyNotificationDeliveryService
Required by: NotificationHandler

Operations:

- void notify(Map<string, string> data)
  - Effect: Delivers a notification to an end user using a third party notification delivery service.
  - Created for: UC15

7.2.49 Notifications

Provided by: NotificationHandler
Required by: RegisteredUserFacade
Operations:

- **Notification** getDetailedNotification(int userID, int notificationID)
  - Effect: Returns detailed information about a notification for a certain registered user.
  - Created for: UC16.4
- List<**Notification**> getNotifications(int userID)
  - Effect: Returns a list of Notification for a certain registered user. Not all details in the Notification objects are filled in.
  - Created for: UC16.2

7.2.50 Notifications

Provided by: ApplicationProviderFacade, CustomerOrganisationFacade, InfrastructureOwnerFacade, RegisteredUserFacade, SystemAdministratorFacade
Required by: ApplicationProviderFacade, CustomerOrganisationFacade, InfrastructureOwnerFacade, SystemAdministratorFacade
Operations:

- **Notification** consultNotification(int userID, int notificationID)
  - Effect: Returns detailed information about a notification for a certain registered user.
  - Created for: UC16.3
- List<**Notification**> consultNotifications(int userID)
  - Effect: Returns a list of Notification for a certain registered user.
  - Created for: UC16.1

7.2.51 Notifiy

Provided by: GWCommunicationLogic, GatewayCommunicationHandler, NotificationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler
Required by: ApplicationContainerManager, ApplicationContainerMonitor, ApplicationExecutionSubsystemMo ApplicationManagementLogic, ApplicationManager, DeviceManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, GatewayCommunicationMonitor, GatewayMonitor, OSCommunicationLogic, OnlineServiceCommunicationHandler
Operations:

- 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
- void notifySystemAdministrator(string message)
  - Effect: Stores a new notification in the system and causes it to be sent to a SIoTIP system administrator.

7.2.52 OSCommunicationMonitoring

Provided by: OSCommunicationLogic, OnlineServiceCommunicationHandler, OnlineServiceMonitor, RequestStore
Required by: OnlineServiceCommunicationMonitor
Operations:

- **Echo** ping()
  - Effect: Sends a ping request to a component. The component should respond with an Echo reply. The monitor uses this to check whether or not the component is still available.
  - Created for: Av1 - The SIoTIP gateway is able to autonomously detect failures of its individual internal communication components.
- void restart()
  - Effect: Restarts the Gateways' communication component.
  - Created for: Av1 - If an internal SIoTIP gateway component fails, the gateway first tries to restart the affected component.

7.2.53 OSMonitoring

Provided by: `OSCommunicationLogic`
Required by: `OnlineServiceMonitor`
Operations:
- **Echo** pingOnlineService()
  - Effect: Makes the BrokerLogic send a ping to the Online Service. The Online Service should respond with an Echo reply. The monitor uses this to check whether or not the component is still available.
  - Created for: The SIoTIP gateway will start synchronising with the Online service within 1 minute after the communication channel becomes available.
- void synchroniseWithOnlineService()
  - Effect: Makes the BrokerLogic fetch all pluggable data and application commands as new requests to start sending to the Online Service again.
  - Created for: Av1 - The SIoTIP gateway will start synchronising with the Online service within 1 minute after the communication channel becomes available.

7.2.54 OSUpdates

Provided by: `OnlineServiceMonitor`
Required by: `OSCommunicationLogic`
Operations:
- void onlineServiceUpdate()
  - Effect: Lets the `OnlineServiceMonitor` know that a message has been received from the Online Service
  - Created for: Av1 - The Online Service should acknowledge each message sent by the SIoTIP gateway so that the gateway can detect failures.

7.2.55 Registration

Provided by: `UnregisteredUserFacade`
Required by: None
Operations:
- boolean canCustomerOrganisationRegister(int infrastructureOwnerID, string hash)
  - Effect: Returns true if the combination of infrastructure owner and hash exists in the system. This means that the infrastructure owner has allowed for a certain customer organistaion to register.
  - Created for: UC1.1
- boolean confirmEmail(string emailAddress, string randomString)
  - Effect: Return true if the given e-mail and randomString match in the database.
    Returns false if user did not click the link in the received e-mail within 30 minutes.
  - Created for: UC1.5
- int createCustomerOrganisation(Map<string, string> data) throws *AddressInvalidException*, *UserNameAlreadyExistsException*, *PaymentInformationInvalidException*

  - Effect: Creates a customer organisation and returns its id. Links the customer organisation to an infrastructure owner by using the information stored in 'createEmailAddressForRegistration'.
  - Created for: UC1.7
- boolean createEmailAddressForRegistration(string emailAddress, int infrastructureOwnerID) throws *EmailAddressAlreadyExistsException*, *EmailAddressInvalidException*
  - Effect: If the given e-mail address is valid and not yet in the database, saves the e-mail address in the database, links it to the infrastructure owner and a unique random string for the confirmation e-mail, and the current timestamp. Afterwards, returns true.
    Otherwise, returns false.
  - Created for: UC1.3-4

7.2.56 RequestData

Provided by: PluggableDevice
Required by: Mote
Operations:
- **DeviceData** getData()
  - Effect: Synchronously retrieve the device data of a device.

- void getDataAsync(int requestID)
  - Effect: Asynchronously retrieve the device data of a device (by calling rcvDataCallback).

7.2.57 RequestData

Provided by: Mote
Required by: DeviceManager, Gateway
Operations:
- **DeviceData** getData(**PluggableDeviceID** pID)
  - Effect: Synchronously retrieve the device data of a pluggable device.
- void getDataAsync(**PluggableDeviceID** pID, int requestID)
  - Effect: Asynchronously retrieve the device data of a pluggable device (by calling rcvDataCallback).

7.2.58 RequestData

Provided by: DeviceDataScheduler
Required by: ApplicationManagementLogic, ApplicationManager
Operations:
- List<**DeviceData**> getData(**PluggableDeviceID** pID, **DateTime** from, **DateTime** to)
  - Effect: Requests data from a specific device in a certain time period.
  - Created for: P2: requests from applications
- Map<**PluggableDeviceID**, List<**DeviceData**>> getDataForDevices(List<**PluggableDeviceID**> devices, **DateTime** from, **DateTime** to)
  - Effect: Returns DeviceData of pluggable devices over a specified time period.
  - Created for: UC24.2

7.2.59 Requests

Provided by: RequestStore
Required by: OSCommunicationLogic
Operations:
- void acknowledgement(int requestID)
  - Effect: Sends an acknowledgement for a request sent by the SIoTIP gateway. Deletes a request that is stored in the RequestStore.

  - Created for: Av1 - the SIoTIP gateway will temporarily store all incoming pluggable data and any issued application commands internally
- List<List<byte>> getNewRequestsForSynchronisation()
  - Effect: Returns pluggable data and application commands as new requests to start sending to the Online Service again.
  - Created for: Av1 - The SIoTIP gateway will start synchronising with the Online service within 1 minute after the communication channel becomes available.

7.2.60 RoleMgmt

Provided by: `UserManager`
Required by: `ApplicationManagementLogic`, `ApplicationManager`, `CustomerOrganisationFacade`
Operations:
- boolean areMandatoryUserRolesAssigned(int applicationInstanceID)
  - Effect: Returns true if all mandatory UserRoles for the application have been assigned to users. Finds the relevant customer organisations through the `ApplicationInstance`.
  - Created for: UC17.1, U2 - easy applications
- List<**UserRole**> getMandatoryUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles that which need to be assigned in order for an `ApplicationInstance` to run.
  - Created for: UC19.7, U2 - easy applications
- List<**UserRole**> getOptionalUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles which can optionally be assigned for an `ApplicationInstance`.
  - Created for: UC19.7, U2 - easy applications
- List<**User**> getUsersWithRoles(int applicationInstanceID)
  - Effect: Returns a list of Users associated to an `ApplicationInstance` that were assigned UserRoles.
  - Created for: UC17.6, U2 - easy applications
- void updateUserRoles(int applicationInstanceID, Map<int, int> usersAndRoles)
  - Effect: Updates the UserRoles assigned to Users for a certain `ApplicationInstance`. 'usersAndRoles' maps User IDs to UserRole IDs.
  - Created for: UC19.9, U2 - easy applications

7.2.61 Sessions

Provided by: `SessionDB`
Required by: `AuthenticationManager`
Operations:
- string createNewSession(int userID)
  - Effect: Returns the ID of a new login session for a certain user.
  - Created for: UC28.2
- boolean deleteSession(string sessionID)
  - Effect: If a session with the given sessionID exists, deletes the session and returns true.
  - Created for: UC29.1
- boolean doesSessionExist(string sessionID)
  - Effect: Returns true if a session with the given sessionID exists.
  - Created for: UC28, UC29

7.2.62 SubscriptionMgmt

Provided by: `CustomerOrganisationFacade`
Required by: `CustomerOgranisationClient`
Operations:
- Map<**Application**, **Subscription**> getApplicationsToSubscribe(int custOrgID)

  - Effect: Returns a map of Applications and Subscriptions a given customer organisation has to those applications
  - Created for: UC19.1, U2 - easy applications
- List<**RoomTopology**> getNecessaryDevicesAndTopologyConfigurations(int applicationID)
  - Effect: Returns a list of RoomTopology which is a minimal requirement for a certain application to run. This can used to display the requirements to a user or to check if requirements are fulfilled.
  - Created for: UC19.5, U2 - easy applications
- List<**Subscription**> getSubscriptions(int customerOrganisationID)
  - Effect: Returns a list of the subscriptions of a customer organisation.
  - Created for: UC20.1
- List<**RoomTopology**> getTopologyOfCustomerOrganisation(int custOrgID)
  - Effect: Returns a list of RoomTopology associated to a customer organisation.
  - Created for: UC19.5, U2 - easy applications
- void registerSubscriptionAndUpdateCriticality(int applicationInstanceID, boolean isCritical, int custOrgID)
  - Effect: Registers a new subscription and updates the criticality of an `ApplicationInstance`.
  - Created for: UC19.11-12, U2 - easy applications
- int subscribeToApplication(int custOrgID, int applicationID)
  - Effect: Creates a new `ApplicationInstance` for an application for a customer organisation and returns its id.
  - Created for: UC19.4, U2 - easy applications
- void unsubscribeFromApplication(int customerOrganisationID, int subscriptionID, int applicationInstanceID)
  - Effect: Unsubscribes a customer organisation from an application.
  - Created for: UC20.3
- void updateApplicationDevicesSettings(int applicationInstanceID, List<**PluggableDeviceID**> devices, List<**Relationship**> relationships)
  - Effect: Updates an `ApplicationInstance`'s device settings. This includes which devices the instance can use and which relationships exist between those devices.
  - Created for: UC19.6, U2 - easy applications
- void updateUserRoles(int applicationInstanceID, Map<int, int> usersAndRoles)
  - Effect: Updates the UserRoles assigned to Users for a certain `ApplicationInstance`. 'usersAndRoles' maps User IDs to UserRole IDs.
  - Created for: UC19.9, U2 - easy applications

7.2.63 SubscriptionMgmt

Provided by: `SubscriptionManager`
Required by: `CustomerOrganisationFacade`
Operations:

- void createSubscription(int custOrgID, int applicationInstanceID)
  - Effect: Creates a subscription for a customer organisation to an `ApplicationInstance`. If the customer organisation is already subscribed to an older version of the the application, then the organisation is unsubscribed from that earlier version.
  - Created for: UC19.12-13, U2 - easy applications
- Map<**Application**, **Subscription**> getApplicationsToSubscribe(int custOrgID)
  - Effect: Returns a map of Applications and Subscriptions a given customer organisation has to those applications
  - Created for: UC19.2, U2 - easy applications
- List<**Subscription**> getSubscriptions(int customerOrganisationID)
  - Effect: Returns a list of the subscriptions of a customer organisation.
  - Created for: UC20.2
- void softDeleteSubscription(int subscriptionID)
  - Effect: Ends a subsription.
  - Created for: UC20.4i

7.2.64 TopologyMgmt

Provided by: InfrastructureOwnerFacade
Required by: None
Operations:

- List<**DeviceStatus**> getStatusDetailsForGateway(int gatewayID)
  - Effect: Returns details about the status of a certain gateway. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**DeviceStatus**> getStatusDetailsForMote(int moteID)
  - Effect: Returns details about the status of a certain mote. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**DeviceStatus**> getStatusDetailsForPluggableDevice(**PluggableDeviceID** pID)
  - Effect: Returns details about the status of a certain pluggable device. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**RoomTopology**> getTopology(int infrastructureOwnerID)
  - Effect: Returns the topology owned by an infrastructure owner.
  - Created for: UC10.1
- void updateTopology(int infrastructureOwnerID, List<**RoomTopology**> updatedTopology)
  - Effect: Updates the topology of an infrastructure owner. The 'updatedTopology' is the new one submitted by the infrastructure owner.
  - Created for: UC10.4

7.2.65 TopologyMgmt

Provided by: GWCommunicationLogic, GatewayCommunicationHandler, OSCommunicationLogic, OnlineServiceCommunicationHandler, TopologyManager
Required by: ApplicationManagementLogic, ApplicationManager, CustomerOrganisationFacade, DeviceManager, GWCommunicationLogic, Gateway, GatewayCommunicationHandler, InfrastructureOwnerFacade, OSCommunicationLogic, OnlineServiceCommunicationHandler
Operations:

- void addDevice(**PluggableDeviceID** id, int moteID)
  - Effect: Adds a new pluggable device to the topology of the infrastructure owner and links it to a mote. The device gets the mote's location by default. If the device is already linked to another mote, overwrites that link.
  - Created for: UC6.3, U2 - easy pluggable device installation
- void addGateway(int gatewayID, int infrastructureOwnerID)
  - Effect: Adds a gateway to a topology of an infrastructure owner. The gateway status 'unplaced' by default.
  - Created for: U2 - gateway installation
- void addMote(int moteID, int gatewayID, int infrastructureOwnerID)
  - Effect: Adds a new mote to a topology of an infrastructure owner. The mote is linked to a gateway and gets status 'unplaced' by default.
  - Created for: UC4.3, U2 - easy mote installation
- boolean arePluggableDevicesPlaced(List<**PluggableDeviceID**> devices)
  - Effect: Returns true if all pluggable devices in the given list have status 'placed' in the topology.
  - Created for: UC17.2, U2 - easy applications
- void deactivateGateway(int gatewayID)
  - Effect: Changes the status of a gateway in the topology to 'inactive'.
  - Created for: U2 - gateway installation
- void deactivateMote(int moteID)
  - Effect: Changes the status of a mote in the topology to 'inactive'.

  - Created for: UC5.2a
- void deactivatePluggableDevice(**PluggableDeviceID** pID)
  - Effect: Changes the status of a pluggable device in the topology to 'inactive'.
  - Created for: UC7.3b
- void deactivatePluggableDevices(List<**PluggableDeviceID**> pIDs)
  - Effect: Changes the status of a list of pluggable devices in the topology to 'inactive'.
  - Created for: UC5.2b
- List<**DeviceStatus**> getStatusDetailsForGateway(int gatewayID)
  - Effect: Returns details about the status of a certain gateway. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**DeviceStatus**> getStatusDetailsForMote(int moteID)
  - Effect: Returns details about the status of a certain mote. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**DeviceStatus**> getStatusDetailsForPluggableDevice(**PluggableDeviceID** pID)
  - Effect: Returns details about the status of a certain pluggable device. e.g. indicating that the device was unavailable between 2 and 3 a.m.
  - Created for: UC10.3a
- List<**RoomTopology**> getTopologyOfCustomerOrganisation(int custOrgID)
  - Effect: Returns a list of RoomTopology associated to a customer organisation.
  - Created for: UC19.5, U2 - easy applications
- List<**RoomTopology**> getTopologyOfInfrastructureOwner(int infrastructureOwnerID)
  - Effect: Returns the topology owned by an infrastructure owner.
  - Created for: UC10.2
- List<**MoteInfo**> getUnplacedMotes(int infrastructureOwnerID)
  - Effect: Returns info about unplaced motes that are owned by an infrastructure owner
  - Created for: UC10.2
- List<**PluggableDeviceInfo**> getUnplacedPluggableDevices(int infrastructureOwnerID)
  - Effect: Returns info about unplaced motes that are owned by an infrastructure owner
  - Created for: UC10.2
- void reactivateDevice(**PluggableDeviceID** id)
  - Effect: Changes the status of a pluggable device in the topology to 'placed'.
  - Created for: UC6.3 - reintroduced device
- void reactivateMote(int moteID)
  - Effect: Changes the status of the mote in the topology to 'placed'. The location of the mote is unchanged, it has already been set.
  - Created for: U2 - Reintroducing a previously known mote should not require any con guration.
- void updateTopology(int infrastructureOwnerID, List<**RoomTopology**> updatedTopology)
  - Effect: Updates the topology of an infrastructure owner. The 'updatedTopology' is the new one submitted by the infrastructure owner.
  - Created for: UC10.5a

7.2.66 UserMgmt

Provided by: `UserManager`
Required by: `CustomerOrganisationFacade`, `UnregisteredUserFacade`
Operations:

- boolean canCustomerOrganisationRegister(int infrastructureOwnerID, string hash)
  - Effect: Returns true if the combination of infrastructure owner and hash exists in the system. This means that the infrastructure owner has allowed for a certain customer organistaion to register.
  - Created for: UC1.1
- boolean confirmEmail(string emailAddress, string randomString)

  - Effect: Return true if the given e-mail and randomString match in the database.
    Returns false if user did not click the link in the received e-mail within 30 minutes.
  - Created for: UC1.5
- int createCustomerOrganisation(Map<string, string> data) throws *AddressInvalidException*, *UserNameAlreadyExistsException*, *PaymentInformationInvalidException*
  - Effect: Creates a customer organisation and returns its id. Links the customer organisation to an infrastructure owner by using the information stored in 'createEmailAddressForRegistration'.
  - Created for: UC1.8
- boolean createEmailAddressForRegistration(string emailAddress, int infrastructureOwnerID) throws *EmailAddressAlreadyExistsException*, *EmailAddressInvalidException*
  - Effect: If the given e-mail address is valid and not yet in the database, saves the e-mail address in the database, links it to the infrastructure owner and a unique random string for the confirmation e-mail, and the current timestamp. Afterwards, returns true.
    Otherwise, returns false.
  - Created for: UC1.3-4
- boolean createEndUser(int customerOrganisationID, Map<string, string> data) throws *EmailAddressAlreadyExistsException*, *EmailAddressInvalidException*, *EndUserAlreadyExistsException*, *PhoneNumberInvalidException*
  - Effect: Verifies provided data for a new end-user, checks that the end-user does not exist in the system already, and then stores the end-user information and associates it with the customer organisation. Returns true if everything succeeded.
  - Created for: UC2.3-4
- List<**User**> getEndUsers(int customerOrganisationID)
  - Effect: Returns a list of Users which are associated to a customer organisation.
  - Created for: UC19.8, U2 - easy applications
- void softDeleteEndUser(int customerOrganisationID, int userID)
  - Effect: Marks the profile of an end-user as 'inactive'.
  - Created for: UC3.4

7.2.67 UserMgmt

Provided by: `CustomerOrganisationFacade`
Required by: None
Operations:

- List<**User**> getEndUsers(int customerOrganisationID)
  - Effect: Returns a list of all end-users associated with a customer organisation.
  - Created for: UC3.1
- List<**UserRole**> getMandatoryUserRoles(int applicationID)
  - Effect: Returns a list of UserRoles that which need to be assigned in order for an `ApplicationInstance` to run.
  - Created for: UC19.7, U2 - easy applications
- boolean registerEndUser(int customerOrganisationID, Map<string, string> data) throws *EmailAddressAlreadyExistsException*, *EmailAddressInvalidException*, *PhoneNumberInvalidException*, *EndUserAlreadyExistsException*
  - Effect: Verifies provided data for a new end-user, checks that the end-user does not exist in the system already, and then stores the end-user information and associates it with the customer organisation. Returns true if everything succeeded.
  - Created for: UC2.3-4
- void unregisterEndUser(int customerOrganisationID, int userID)
  - Effect: Unregisters an end-user.
  - Created for: UC3.3

7.3 Exceptions

- *AddressInvalidException* Thrown if a given address is invalid. e.g. does not exist.

- *AppMessageNotAllowedException* Thrown if one part of an application is not allowed to send a certain message or command to another part of the application. Contains the reason why the message is not allowed.
- *EmailAddressAlreadyExistsException* Thrown if a new given email address already exists in the system.
- *EmailAddressInvalidException* Thrown if a given email address is invalid. e.g. badly formatted
- *EndUserAlreadyExistsException* Thrown if an entered end user already exists in the system.
- *PaymentInformationInvalidException* Thrown if given payment information is invalid.
- *PhoneNumberInvalidException* Thrown if a given phone number is invalid. e.g. invalid country code
- *UnknownCommandException* Thrown if a command is sent by an application that is not recognised.
- *UnknownConfigurationParameterException* Thrown if an application tries to set a configuration parameter on a device that is not recognised.
- *UserNameAlreadyExistsException* Thrown if a given username already exists in the system.

7.4 Data types

- **AccessRights**:
  Contains information about access rights to a pluggable device.
- **Application**:
  Contains information about an application uploaded to the SIoTIP system.
- **ApplicationCode**:
  Contains all of the code for an application. This could include code for the online service version of the application, code for the gateway version of the application, configuration files, etc.
- **ApplicationInstance**:
  Attributes: int id, int status, int customerOrganisationID, boolean isGatewayVersion
  Contains information on an application instance. When an ApplicationInstance is running on the Online Service and on a Gateway, the ApplicationIntances in the ApplicationContainers share the same id.
- **DateTime**:
  Represents an instant in time, expressed as a date and time of day.
- **DeviceData**:
  Data from a pluggable device. For sensors, this contains sensor values. For actuators, this contains the state of the actuator. The data is encapsulated within a JSON message, and should be converted into something meaningful based on the device type of the pluggable device that sent the data.
- **DeviceStatus**:
  Attributes: **PluggableDeviceID** pID, int moteID, int gatewayID, **DateTime** from, **DateTime** to, string status, string details
  Contains status details for a specific gateway/mote/pluggable device.
- **Echo**:
  Is a reply to an echo request (ping). Contains an Identifier and Sequence Number that matches the request that caused the Echo reply.
- **GatewayInfo**:
  Attributes: int gatewayID, **IPAddress** IPAddress, int infrastructureOwnerID
  An object containing information on a Gateway.
- **Invoice**:
  Contains information about an invoice, such as true provided service or product, the amount to be paid, the date the payment is due, the contact details of the receiver of the bill, etc.
- **IPAddress**:
  Represents an Internet Protocol address.
- **MoteInfo**:
  Attributes: int moteID, int manufacturerID, int productID, int batteryLevel
  An object containing information on a Mote. This is a list of key-value pairs. The values depend on the type of mote. For example, only a battery-powered mote would include the

batterylevel info.

- **Notification**:
  Attributes: int id, int recipientUserID, string message, int communicationChannelID, int notificationTypeID, **DateTime** dateTime, int triggerID, int status
  Contains information about a notification. The communicationChannelID represents the communication channel that will be used to send the notification to the user. The notificationTypeID denotes the type of the notification (normal / alarm / ...).
- **PluggableDeviceID**:
  A unique identifier of a pluggable device.
- **PluggableDeviceInfo**:
  Attributes: **PluggableDeviceID** id, **PluggableDeviceType** type, Map<string, string> config
  Contains information on a pluggable device.
- **PluggableDeviceType**:
  Attributes: int manufacturerID, int productID, int type, int measurementUnits
  Denotes the type of a pluggable device. It specifies the manufacturer, a model identifier (if any) of the pluggable (e.g. heat-o sensor 5000), the type of pluggable device (e.g. temperature sensor or power socket actuator) and the used measurement units (e.g. degrees Celsius, degrees Fahrenheit or decibel). Within MicroPnP this information is also used to retrieve and install the correct drivers (this is outside the scope of the system),
  This description is taken from the Discussion Board on Toledo.
- **Relationship**:
  Attributes: **PluggableDeviceID** pID1, **PluggableDeviceID** pID2, string relationship
  Represents a relationship between two devices. e.g. device2 can be used for the purpose that device1 fulfills, device2 is preferred over device1 for a responsibility, etc.
- **RoomTopology**:
  Attributes: int roomID, List<**PluggableDeviceInfo**> pluggableDevices, **Relationship** relationships, int buildingID, List<**MoteInfo**> motes, List<**GatewayInfo**> gateways
  Represents a room in a topology. Contains the room's devices and the relationships between those devices. Depending on whether this data type is used for customer organisation requests or infrastructure owner requests, the motes and gateways field will be left blank.
- **Subscription**:
  Attributes: int id, int status, int customerOrganisationID, int applicationInstanceID, int applicationID
  Contains data about a subscription by a customer organisation for an application instance. Data about period/length of the subscription is stored in invoices.
- **User**:
  Represents a user in the SIoTIP system.
- **UserRole**:
  Attributes: int userID, int roleID, int applicationInstanceID, int customerOrganisationID
  Contains information on the role that a User fulfills for a certain ApplicationInstance.
- **Version**:
  Represents the version of an application.

A. Attribute-driven design documentation

A.1 Introduction

This chapter contains our ADD log. First, we list the changes we made the ADD process so it fits our workflow better. The remaining part of this chapter is the ADD log. Decompositions 1 and 2 have been changed relative to phase 2a of this project, because we forgot about the given interfaces for gateways and pluggable devices and made up our own (but similar) interfaces instead. The decompositions have been updated to use the given interfaces.

A.2 Adapted ADD process

We left off step a ("Pick an element that needs to be decomposed") since we never really chose a single Element to decompose. Instead, we chose the drivers for each decomposition first and then looked at which elements/subsystems would require changes or which new elements we would need to satisfy those drivers.
For component, interfaces, datatypes: we list the new ones, but refer to the plugin exported catalog for descriptions

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

We changed these titles to reflect the architectural drivers we chose first and then denote which elements/subsystems needed changes to satisfy the drivers.

New data types and Interfaces for child modules

For each decomposition, we have listed all new interfaces and data types that we added during the decomposition, but all details have been left out. We used the Visual Paradigm plugin provided by the SA team to generate the element catalog of chapter 6. All details can be found in there.

Also, since an ADD log was no longer a requirement for phase 2b, we have left out intermediary "OtherFunctionality" components and figures of diagrams. This was done to save time.

Verify and refine

We have skipped the verify and refine step because we chose to handle all chosen architectural drivers completely in every decomposition. We did not find this step to be useful after decompositions 1 and 2.

A.3 Decomposition 1: Av3, UC14, UC15, UC18 (SIoTIP System)

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.

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 `DeviceManager` 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 occured 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 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 `DeviceManager`. It stores the requirements for pluggable devices set by applications for all applications that use the gateway that the the `DeviceManager` 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 `DeviceManager` 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 funcionality 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 `DeviceManager` 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.

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- ApplicationManager: Av3
- Database: /
- DeviceManager: Av3
- GatewayFacade: /
- Mote: UC14
- NotificationHandler: UC15

Decomposition Figure A.1 shows the components resulting from the decomposition in this run.

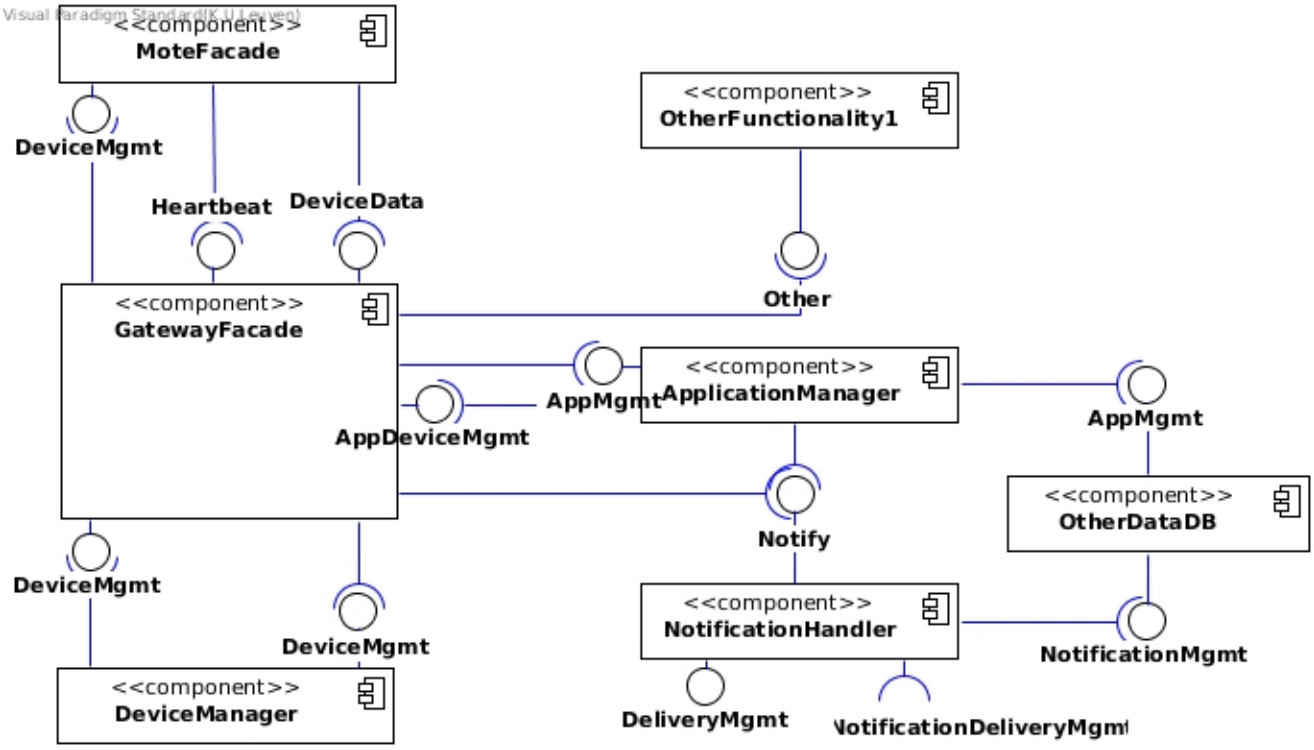

Figure A.1: Component-and-connector diagram of this decomposition.

Deployment Figure A.2 shows the allocation of components to physical nodes.

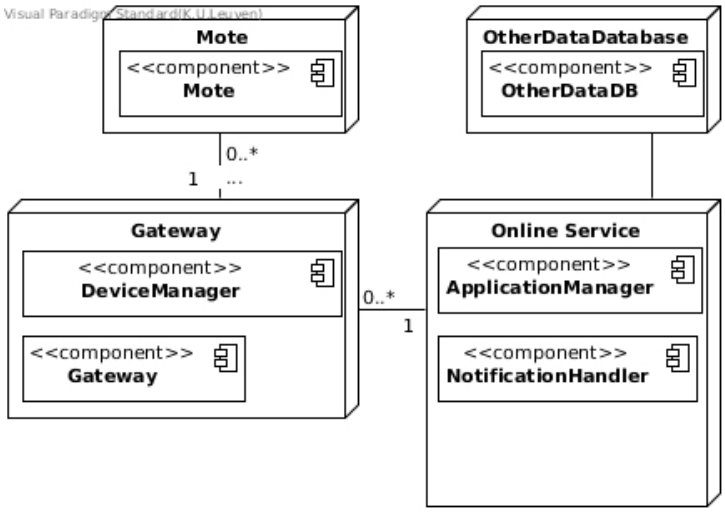

Figure A.2: Deployment diagram of this decomposition.

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ApplicationManager

- GWAppInstanceMgmt

Database

- NotificationMgmt
- AppMgmt

GatewayFacade

- Heartbeat
- DeviceData
- DeviceMgmt
- AppDeviceMgmt

Mote

- DeviceMgmt

NotificationHandler

- Notify
- DeliveryMgmt

External notification delivery service

- NotificationDeliveryMgmt

DeviceManager

- DeviceMgmt

New data types

This section lists the data types introduced in this decomposition.

- PluggableDeviceInfo
- Notification
- ApplicationInstance
- Subscription
- PluggableDeviceID
- PluggableDeviceType
- DeviceData
- Map<String,String>

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

Mote

- *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 SIoTIP 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

DeviceManager

- *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

A.4 Decomposition 2: M1, P2, UC11 (OtherFunctionality1)

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.

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 `PluggableDeviceDataDB` 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 `PluggableDeviceDataDB`. 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 statisfy this constraint, all data directly related to pluggable data has been separated into the `PluggableDeviceDataDB`. All requests concerning pluggable data will be handled by this new component. `PluggableDeviceDataDB` will run on a node different from the node that the `Datbase` 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:

- *PluggableDevice*: 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.
- *PluggableDeviceDataDB*: 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.

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- DeviceDataConverter: M1
- DeviceDataScheduler: P2
- PluggableDeviceDataDB: P2
- PluggableDevice: UC11

Decomposition Figure A.3 shows the components resulting from the decomposition in this run.

Deployment Figure A.4 shows the allocation of components to physical nodes.

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ApplicationManager

- ForwardData

Mote

- DeviceData

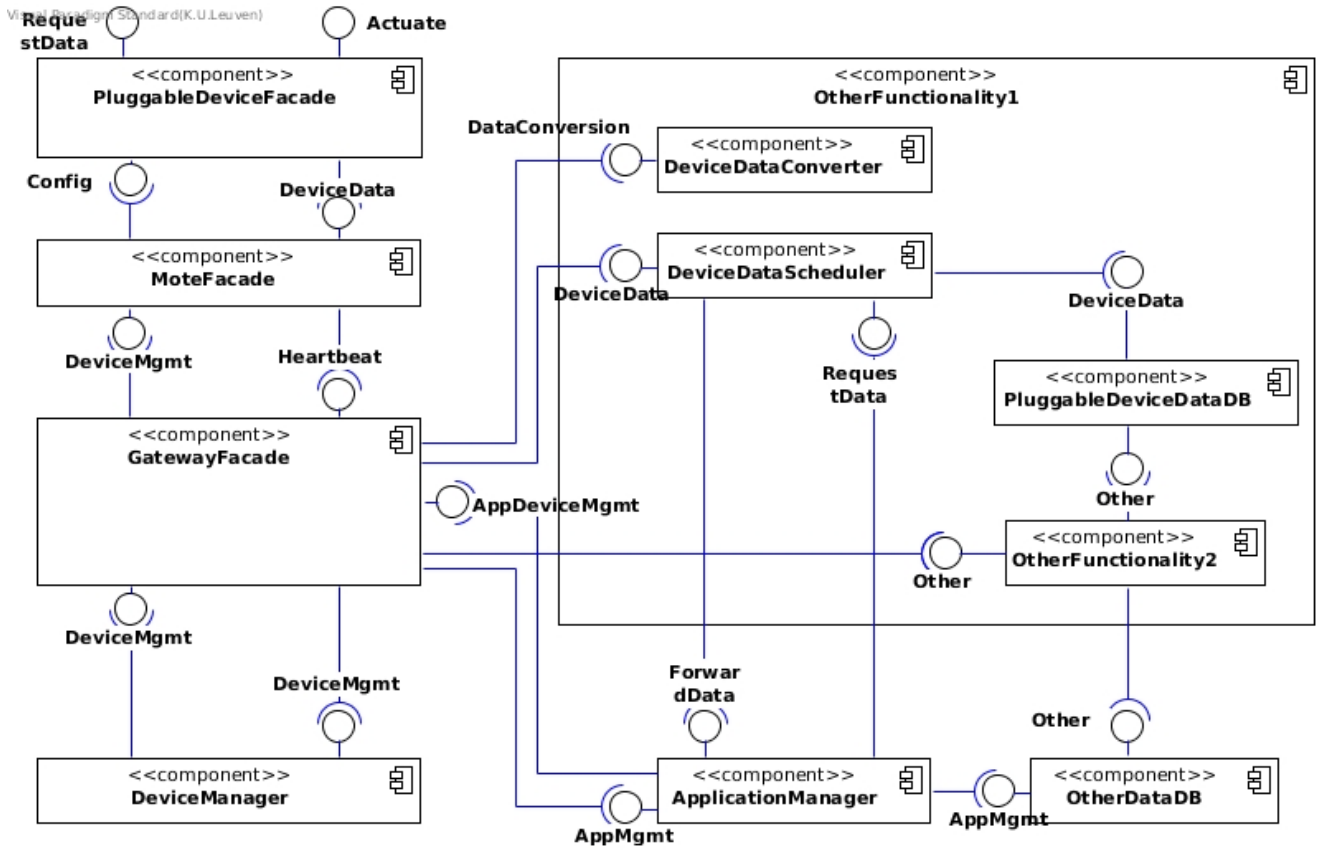

Figure A.3: Component-and-connector diagram of this decomposition.

PluggableDevice

- Actuate
- Config
- RequestData

DeviceDataConverter

- DataConversion

DeviceDataScheduler

- RequestData
- DeviceData

PluggableDeviceDataDB

- DeviceData

New data types

This section lists the new data types introduced during this decomposition.

- DateTime

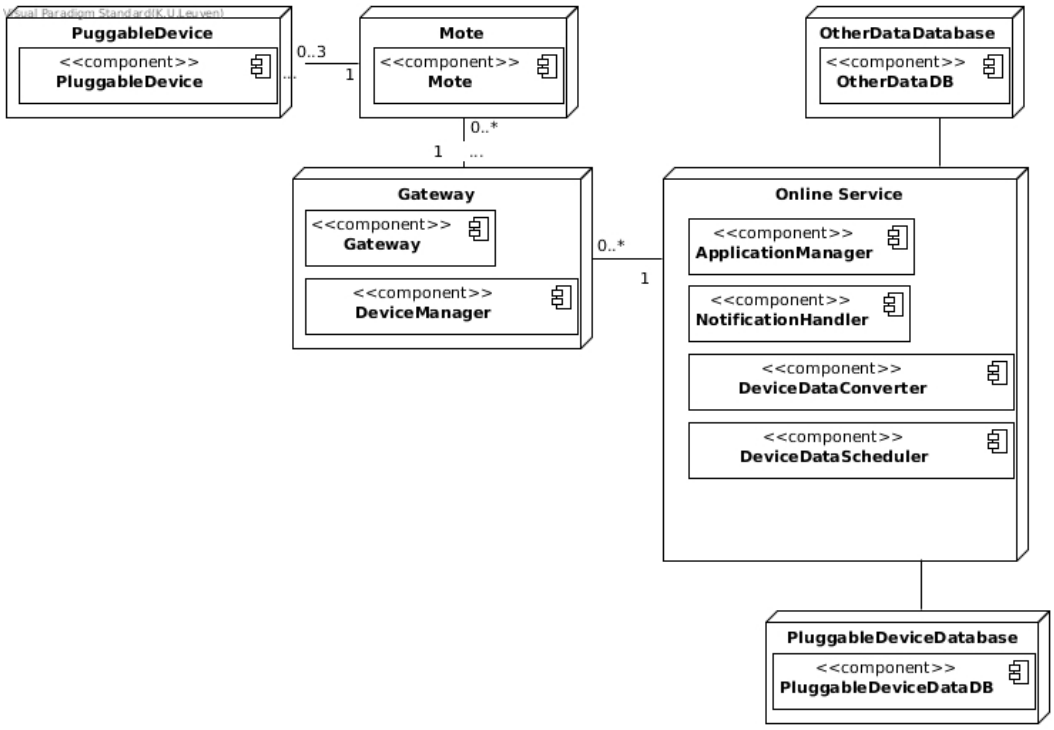

Figure A.4: Deployment diagram of this decomposition.

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

OtherDataDB

- None

GatewayFacade

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

Mote

- *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

PluggableDeviceDataDB

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

PluggableDevice

- *U2*: Easy Installation: d

DeviceManager

- *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

A.5 Decomposition 3: U2, UC4, UC6, UC9, UC10, UC17, UC19

Selected architectural drivers

The non-functional drivers for this decomposition are:

- *U2*: Easy installation

The related functional drivers are:

- *UC4*: Install mote
- *UC6*: Insert a pluggable device into a mote
- *UC9*: Configure pluggable device access rights
- *UC10*: Consult and configure topology
- *UC17*: Activate an application
- *UC19*: Subscribe to application

Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.
Did you ever hear the tragedy of Darth Plagueis The Wise? I thought not. It's not a story the Jedi would tell you. It's a Sith legend. Darth Plagueis was a Dark Lord of the Sith, so powerful and so wise he could use the Force to influence the midichlorians to create life... He had such a knowledge of the dark side that he could even keep the ones he cared about from dying. The dark side of the Force is a pathway to many abilities some consider to be unnatural. He became so powerful... the only thing he was afraid of was losing his power, which eventually, of course, he did. Unfortunately, he taught his apprentice everything he knew, then his apprentice killed him in his sleep. Ironic. He could save others from death, but not himself.

U2: Gateway installation The gateway should not require any configuration, other than being connected to the local wired or WiFi network, after it is plugged into an electrical socket. An infrastructure owner should be able get the SIoTIP gateway up-and-running (connected) within 10 minutes given that the information (e.g. WiFi SSID and passphrase) is available to the person responsible for the installation.

A connection to the internet is a constraint of the GatewayFacade. After the gateway is connected to the internet (we don't model this), it connects to the gateway (we don't model this?) and registers itself (we model this).
When an infrastructure owner orders a gateway, that gateway is linked to the IO. Gateway was already in the DeviceDB, but it was not linked to anyone. It has a gatewayID => unique identifier gatewayID same like motes.
Important info related to gateways: GatewayID (new class), infrastructureOwnerID, IPAddress, status (active/inactive), location (in topology table) GatewayInfo(int gatewayID, int manufacturerID, int productID, int infrastructureOwnerID, IPAddress ip, int status)
gateway registers with online service:
We cannot link the gateway to an exact location for the infrastructure owner, because he might be managing multiple buildings an IP addresses can be dynamic. If we let the IO choose for which building the gateway is, then this is bad for usability and he has to configure anyways. Also in that case he cannot buy spare gateways, unless he buys spare gateways for every gateway in the building

U2: Mote installation Installing a new mote should not require more configuration than adding it to the topology. Adding new motes, sensors or actuators should not involve more than just starting motes, and plugging devices into motes âĂŞ plug-and-play! Reintroducing a previously known mote, with the same pluggable devices attached to it, should not require any configuration. It is automatically re-added on its last known location on the topology. The attached pluggable devices are automatically initialised and configured with their last known configuration and access rights.

Thing that need to happen automatically: *) mote should find the gateway (mote sends a broadcast message->ReceiveBroadcast) => this is done automatically? see remarks of the use case *) gateway should register the mote (DeviceManager update, store entry in DB) *) on reintroduction of motes: DeviceManager notices this, makes the gateway send a message to online service to reuse some old topology

U2: Pluggable device installation Adding new sensors or actuators should require no further customer actions besides plugging it into the mote. Configurable sensors and actuators should have a working default configuration. Pluggable devices added to an already known mote are automatically added in the right location on the topology. Making (initialised) sensors and actuators available to customer organisations and applications should not require more effort than configuring access rights (cf. UC9).
) After devices are plugged in: connect to mote, set up default configurations *) if the mote is already known, the device is added to the right location on the topology *) need something for configuration of access rights, can only happen for initialised devices
*) for reactivating last configurations: just set status to active and don't change configuration field, it will still be the same as in the past alternative: current_configuration and last_configuration in DB alternative: store all configurations on Gateway -> but it has bad resources alternative: store all versions on DeviceDB -> but lots of useless data then = extra work for db
*) Pluggable devices added to an already known mote are automatically added in "the right location" on the topology. what exactly is a location? => when a pluggable device is connected to a new mote, the pluggable device gets the location of the mote by default

U2: Easy applications Applications should work out of the box if the required sensors and actuators are available. Only when mandatory end-user roles must be assigned, additional explicit configuration actions are required from a customer organisation (cf. UC17, UC19).
) if there is a subsription and new hardware is plugged in: need something to check if some application can be activated now => see UC6: checkApplicationsForActivationForInfrastructureOwner *) need something to assign user roles to users during UC19

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- AccessRightsManager: U2, UC9
- ApplicationContainerManager: U2, UC17
- CustomerOrganisationClient: U2, UC19
- CustomerOrganisationFacade: U2, UC19
- DeviceDB: U2, UC4, UC6
- InfrastructureOwnerClient: U2, UC9
- InfrastructureOwnerFacade: U2, UC9
- InfrastructureOwnerManager: U2, UC9
- InvoiceManager: U2, UC17
- TopologyManager: U2, UC4, UC6, UC17
- SubscriptionManager: U2, UC19
- UserManager: U2, UC17, UC19

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

AccessRightsManager

- AccessRightsMgmt

ApplicationContainerManager

- AppMgmt

ApplicationManager

- FrontEndAppMgmt
- IOAppMgmt

CustomerOrganisationFacade

- SubscriptionMgmt

DeviceDB

- AccessRightsMgmt
- DeviceMgmt
- TopologyMgmt
- IODeviceMgmt

InfrastructureOwnerFacade

- AccessRights

InfrastructureOwnerManager

- IOMgmt

InvoiceManager

- InvoiceMgmt

OtherDataDB

- InvoiceMgmt
- IOMgmt
- SubscriptionMgmt
- UserRoleMgmt

TopologyManager

- TopologyMgmt

SubscriptionManager

- SubscriptionMgmt

UserManager

- RoleMgmt

New data types

This section lists the new data types introduced during this decomposition.

- Application
- IPAddress
- Relationship
- RoomTopology
- User
- UserRole

A.6 Decomposition 4: Av2, UC12, UC25, UC26, UC27 (application execution subsystem)

Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av2*: Application failure

The related functional drivers are:

- *UC12*: Perform actuation command
- *UC13*: Configure pluggable device
- *UC25*: Access topology and available devices
- *UC24*: Consult historical data
- *UC26*: Send application command or message to external front-end
- *UC27*: Receive application command or message from external front-end

Rationale At this point the remaining drivers were Av1, Av2, and P1, which all had medium priority. We chose decompositions 4, 5, and 6 based on the priorities of the use cases that are related to the quality attributes.
The related use cases from now on are the ones that would use components that are going to be changed in the decomposition.

Architectural design

This section describes what needs to be done to satisfy the requirements for this decomposition and how involved problems/obstacles are solved.
If you read this, you are cool!
RATIONALE: ApplicationContainerManager and DeviceCommandConstructor need to be reconfigured ivm some used interfaces

Av2: Detection of failures The system is able to autonomously detect failures of its individual application execution components, failing applications, and failing application containers.
Upon detection, a SIoTIP system administrator is notified.
The failure of an internal application execution component is detected within 30 seconds. Detection of failed hardware or crashed software happens within 5 seconds. SIoTIP system administrators are notified within 1 minute.

To detect failures, we made use of the `Container` pattern. The application execution subsystem is composed of:

- `ApplicationContainer`
- `ApplicationContainerMonitor`
- `ApplicationContainerManager`
- `ApplicationExecutionSubsystemMonitor`

The `ApplicationContainer`s are deployed in groups on different nodes.
`ApplicationContainer`: is a container/sandbox that has 1 running application instance `ApplicationContainerMonitor`: monitors the `ApplicationContainer` instances
To detect failing applications, `ApplicationContainer` and `ApplicationContainerMonitor` ApplicationContainer -> ApplicationContainerMonitor: void applicationCrashed(id applicationInstanceID)
To detect failling application containers, `ApplicationContainerMonitor` ApplicationContainerMonitor -> ApplicationContainer: Echo ping() -> we say container has crashed/failed when the following has no response:

To detect failures of individual application execution components, This means that one of `ApplicationContainer`, `ApplicationContainerMonitor`, `ApplicationContainerManager` crashed. If the `ApplicationContainer` failed, then the `ApplicationContainerMonitor` would detect this. If one of the other two components failed, the `ApplicationExecutionSubsystemMonitor` is put in place to detect this.
These components will be deployed on the Online Service and on gateways. Since gateways are weaker machines than the ones on the Online Service, the ApplicationContainer can be confgured differently for gateways. The ApplicationContainers will then have stricter limits on resources used of the node they are working on. DeviceCommandConstructor and ApplicationContainerManager interfaces need to begin re-routed depening on whether they are in a Gateway or on the Online Service.

Av2: Resolution of application failures and application execution component failures In case of application crash, the system autonomously restarts failed applications. If part of an application fails, the remaining parts remain operational, possibly in a degraded mode (graceful degradation).
After 3 failed restarts the application is suspended, and the application developer and customer organisation are notified within 5 minutes.

In case of failure of application execution components or an application container, a system administrator is notified.
SIoTIP system administrators are notified within 1 minute.

When an application instance fails, the ApplicationContainerMonitor detects this and sends a command to the ApplicationContainerManager to restart the application instance. The ApplicationContainerMonitor keeps track of how many times the application instance has been restarted after a failure. After 3 failed restarts, the monitor send a command to the ApplicationContainerManager to suspend the application instance and send a notification to the application developers of the application and to the affected customer organisaiton. Also, to achieve graceful degradation, the ApplicationContainerManager notifies other parts of the application instance of its suspension.
If one of the components of the application execution subsystem fails, a SIoTIP system administrator is notified.

Av2: Failures do not impact other applications or other functionality of the system This does not affect other applications that are executing on the Online service or SIoTIP gateway. This does not affect the availability of other functionality of the system, such as the dashboards.
Applications fail independently: they are executed within their own container to avoid application crashes to affect other applications.

Each ApplicationContainer contains one application instance. If an application fails, then this will be handled by the application execution subsystem so this does not affect any other application or other functionality of the system. The ApplicationContainers are constructed such that failures of applications do not affect the containers. The ApplicationContainers are to be deployed on different nodes alone or grouped with other containers. Write something here.

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- ApplicationClient, Av2, UC12, UC13, UC24, UC25, UC26
- ApplicationContainer, Av2, UC12, UC13, UC24, UC25, UC26, UC27
- ApplicationContainerManager, Av2, UC12, UC13, UC24, UC25, UC26, UC27
- ApplicationContainerMonitor, Av2, UC27
- ApplicationExecutionSubsystemMonitor, Av2

- ApplicationFacade, Av2, UC12, UC13, UC24, UC25, UC26
- ApplicationManagementLogic, Av2, UC12, UC13, UC24, UC25, UC26
- DeviceCommandConstructor, Av2, UC12, UC13

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ApplicationContainer

- AppMessages
- AppInstanceMgmt
- AppMonitoring

ApplicationContainerManager

- AppMessages
- AppInstanceMgmt
- DeviceCommands
- DeviceData

ApplicationContainerMonitor

- Monitoring
- AppStatus

ApplicationFacade

- DeviceData
- DeviceCommands

ApplicationManagementLogic

- DeviceData
- DeviceCommands
- FrontEndAppRequests
- GWAppInstanceMgmt
- AppMessages

DeviceCommandConstructor

- Commands

DeviceManager

- DeviceCommands

DeviceDB

- AppDeviceMgmt

New data types

This section lists the new data types introduced during this decomposition.

- Echo the response to a ping message

A.7 Decomposition 5: Av1 (Gateway - Online Service communication subsystem)

Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av1*: Communication between SIoTIP gateway and Online Service

Architectural design

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

Av1: New Gateway responsibilities The SIoTIP gateway is able to autonomously detect failures of its individual internal communication components.
The Online Service should acknowledge each message sent by the SIoTIP gateway so that the gateway can detect failures.
If an internal SIoTIP gateway component fails, the gateway first tries to restart the affected component. If the failure persists, the SIoTIP gateway reboots itself entirely. Note that the SIoTIP gateway, due to the occurred failure, cannot contact a system administrator itself.
If (an internal communication component of) the Online Service or the communication channel has failed, the SIoTIP gateway will temporarily store all incoming pluggable data and any issued application commands internally.
If the Online Service becomes unreachable, application parts running locally on the SIoTIP gateway continue to operate normally.
The SIoTIP gateway will start synchronising with the Online service within 1 minute after the communication channel becomes available.
The SIoTIP gateway can store at least 3 days of pluggable data before old data has to be overwritten.

OnlineServiceBroker: Isolates communication-related concerns between Gateways and the Online Service along with GatewayBroker on the Online Service. Forwards requests from one party to the other and transmits results and possible exceptions.
OnlineServiceBrokerMonitor: Monitors the communication component on Gateways. If the communication component fails, the monitor tries to restart it. If the failure persists, the gateway reboots itself entirely.
In OnlineServiceBroker: BrokerLogic: Handles all functionality related to communication. RequestStore: Temporarily stores all pluggable data and issued application commands until they can be deleted (= until an acknowledgement has been received for the request by the Online Service). Can store at least 3 days of pluggable data before old data has to be overwritten. OnlineServiceMonitor: Monitors the Gateway's connectivity to the Online Service. If the Online Service or the communication channel has failed, all requests to the Online Service will be stopped and stored in the RequestStore. An explicit command for this is not necessary, because the requests in the RequestStore will not be deleted, since no acknowledgements are received anymore from the Online Service. After the monitor detects that a connection to the Online Service is possible again, it makes the gateway start synchronising again. When the Online Service is unreachable, application parts running locally on the SIoTIP gateway continue to operate normally.
INTERFACES: OnlineServiceBroker: HAS ALL INTERFACES PROVIDED BY COMPONENTS INSIDE OF IT
interface CommunicationComponentMonitoring used by OnlineServiceBrokerMonitor: boolean check() void restart()
BrokerLogic: interface OSCommunication used by Online Service: void acknowledgement(int requestID) void send(...) void receive(...)
interface OSMonitoring used by OnlineServiceMonitor: Echo pingOnlineService() void synchroniseWithOnlineService()

RequestStore: All interfaces that are going to the online service now go to the broker All requests come in here and are stored The requests are then forwarded to BrokerLogic containing a unique requestID
Interface Communication used by BrokerLogic: void deleteRequest(int requestID) List<Object> getNewRequestsForSynchronisation()
OnlineServiceMonitor: Online service keeps track of connection to Online Service. If the Online Service becomes unreachable (= does not send ACKs anymore), then start pinging the Online Service
Interface OSUpdates used by BrokerLogic: void onlineServiceUpdate()

Av1: New Online Service responsibilities The Online Service is able to autonomously detect failures of its individual internal communication components.
The Online Service is able to detect that a SIoTIP gateway is not sending data anymore based on the expected synchronisation interval.
The Online Service notifies the infrastructure manager and a SIoTIP system administrator when the outage of a SIoTIP gateway is detected.
The failure of an internal SIoTIP Online Service component is detected within 30 seconds.
The detection time for a failed SIoTIP gateway or channel depends on the transmission rate of the gateway. An outage is de
ned as 3 consecutive expected synchronisations that do not arrive within 1 minute of their expected arrival time.
The infrastructure owner is noti
ed within 5 minutes after the detection of an outage of their gateway.
A SIoTIP system administrator should be noti
ed within 1 minute after the detection of a simultaneous outage of more than 1% of the registered gateways.

GatewayBroker: Isolates communication-related concerns between the Online Service Gateways and along with OnlineServiceBroker on Gateways. Forwards requests from one party to the other and transmits results and possible exceptions.
Sends acknowledgements for all messages sent by Gateways so that they can detect failures. GatewayBrokerMonitor: Monitor the communication component for communication with gateways on the Online Service.
In GatewayBroker: BrokerLogic: Handles all functionality related to communication. GatewayMonitor: Monitors the connectivity status all gateways. Can detect that a gateway is not sending data anymore based on the expected synchronisation interval. If 3 consecutive expected synchronisations do not arrive within 1 minute of their expected arrival time, this is detected as a gateway outage. When outages of gateways are detected, the infrastructure owners that own the gateways and a SIoTIP system administrator are notified. When the connectivity status change of a Gateway is detected, this is saved in the DeviceDB.
INTERFACES: GatewayBroker: HAS ALL INTERFACES PROVIDED BY COMPONENTS INSIDE OF IT interface CommunicationComponentMonitoring used by GatewayBrokerMonitor: boolean check()
BrokerLogic: ALL INTERFACES FROM OS TO GATEWAY ARE NOW TO THIS COMPONENT
interface GWCommunication used by Gateway/BrokerLogic:
GatewayMonitor:
interface GatewayUpdates used by BrokerLogic: void gatewayUpdate(int gatewayID, DateTime time)
NotificationHandler: Interface Notify used by GatewayMonitor
DeviceDB: Interface DeviceMgmt used by GatewayMonitor: void setGatewayUnreachable() void getPercentageOfUnreachableGateways()
DEPLOYMENT RATIONALE: OnlineServiceBroker OnlineServiceBrokerMonitor
GatewayBroker GatewayBrokerMonitor
For both the Online Service and Gateways, the components used for communication (`OnlineServiceBroker`, `GatewayBroker`) are to be deployed on different nodes than their monitoring components (`OnlineServiceBrokerMonitor`, `GatewayBrokerMonitor`). Otherwise, if the node of a communication component fails, its monitoring component would also fail and thus nothing would be detected.

Alternative for monitoring of gateways: Gateway updated come to GatewayMonitor We could make GatewayMonitor ping all the gateways However, this would increase traffic on the network
Alternative for communication: Messaging, Publishher Subscriber

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- BrokerLogic: Av1
- OnlineServiceBroker: Av1
- OnlineServiceBrokerMonitor: Av1
- GatewayBroker: Av1
- GatewayBrokerMonitor: Av1
- RequestStore: Av1
- GatewayMonitor: Av1
- OnlineServiceMonitor: Av1

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

BrokerLogic

- Communication
- OSCommunication
- GWCommunication
- OSMonitoring

OnlineServiceBroker

- CommunicationComponentMonitoring

GatewayBroker

- CommunicationComponentMonitoring

RequestStore

- Communication

GatewayMonitor

- GatewayUpdates

OnlineServiceMonitor

- OSMonitoring

A.8 Decomposition 6: P1, UC1, UC2, UC3, UC5, UC7, UC8, UC16, UC20

Selected architectural drivers

The non-functional drivers for this decomposition are:

- *P1*: Large number of users

The related functional drivers are:

- *UC1*: Register a customer organisation
- *UC2*: Register an end-user
- *UC3*: Unregister an end-user
- *UC5*: Uninstall mote
- *UC7*: Remove a pluggable device from its mote
- *UC8*: Initialise a pluggable device
- *UC10*: Consult and configure the topology
- *UC16*: Consult notification message
- *UC20*: Unsubscribe from application

Architectural design

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

*) Most components can be duplicated for load balancing
) Related DB's should use a DBMS that can be scaled horizontally.
) DB calls caused by IO vs CO have been mostly split up in DeviceDB and OtherDataDB, so those 2 groups of users don't influence each other too much

The initial deployment of SIoTIP should be able to deal with at least 5000 gateways in total, and should be provisioned to service at least 3000 registered users simultaneously connected to SIoTIP.
-> 5000 gateways * 4 motes per gateway * 3 devices per mote = 60000 devices
-> Keep communication with gateways at a minimum
e.g. gateway messages are of type "send data", "new device connected", ...
-> LOAD BALANCING

P1: Problem title Scaling up to service an increasing amount of infrastructure owners, customers organisations and applications should (in worst case) be linear ; i.e. it should not require proportionally more resources (machines, etc.) than the initial amount of resources provisioned per customer organisation/infrastructure owner and per gateway.

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- ApplicationProviderClient: UC16
- ApplicationProviderFacade: UC16
- RegisteredUserFacade: UC16
- SystemAdministratorFacade: UC16
- UnregisteredUserClient: UC1
- UnregisteredUserFacade: UC1

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ApplicationProviderFacade

- Notifications

CustomerOrganisationFacade

- Notifications
- UserMgmt

InfrastructureOwnerFacade

- Notifications
- TopologyMgmt

NotificationHandler

- Notifications

RegisteredUserFacade

- Notifications

SystemAdministratorFacade

- Notifications

UnregisteredUserFacade

- Registration

UserManager

- UserMgmt

New data types

This section lists the new data types introduced during this decomposition.

- Invoice
- EmailAddressAlreadyExistsException
- EmailAddressInvalidException
- EndUserAlreadyExistsException
- PaymentInformationInvalidException
- PhoneNumberInvalidException
- UserNameAlreadyExistsException

A.9 Decomposition 7: UC28, UC29 (authentication subsystem)

At this point, all quality attributes have been handled. The remaining decompositions handle all of the use cases that are left. The order is based on the priority of the use cases.

Selected architectural drivers

The functional drivers are:

- *UC28*: Log in
- *UC29*: Log out

Sessions are used to check whether or not a user is currently logged in.
On successful login, a new session is created and stored in the SessionDB with a unqiue sessionID which is returned to the front end. The client then sends this sessionID in future requests to represent that the user is logged in.
A database is used for checking the sessions, so that replication can be done on components that handle front end requests. If the session was stored in files, this would add extra overhead to keep all replica's that store session files consistent. It is easier to do this using a database. Also, a separate database is used to not add a lot of extra requests for sessions to another database.

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- SessionDB: UC28, UC29
- AuthenticationManager: UC28, UC29

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ApplicationProviderFacade

- Authentication

AuthenticationManager

- Authentication

CustomerOrganisationFacade

- Authentication

InfrastructureOwnerFacade

- Authentication

NotificationHandler

- Authentication

OtherDataDB

- Authentication

RegisteredUserFacade

- Authentication

SessionDB

- Sessions

SystemAdministratorFacade

- Authentication

A.10 Decomposition 8: UC22, UC23 (application upload and statistics)

Selected architectural drivers

The functional drivers are:

- *UC22*: Upload an application
- *UC23*: Consult application statistics

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ApplicationProviderFacade

- Applications

ApplicationManagementLogic

- ApplicationTesting

New data types

This section lists the new data types introduced during this decomposition.

- Version
- ApplicationCode

A.11 Decomposition 9: UC21 (invoicing subsystem)

Selected architectural drivers

The functional drivers are:

- *UC21*: Send invoice

Instantiation and allocation of functionality

This section lists the new components which instantiate our solutions described in the section above. For each component we note the quality attribute or use case that prompted us to create it. Descriptions about the components can be found under chapter 6.

- ThirdPartyInvoicingService: UC21

Interfaces for child modules

This section lists new interfaces assigned to the components defined in the section above. Detailed information about each interface and its methods can be found under chapter 6.

ThirdPartyInvoicingService

- InvoiceDeliveryMgmt

InvoiceManager

- DeliveryMgmt