KU LEUVEN

DEPARTMENT OF COMPUTER SCIENCE

Shared Internet-of-Things Infrastructure Platform (SIoTIP)

Part 2B

Architectural extension

Geens–Jochim–Salim

Academic Year 2018–2019

H09B5B: Software Architectuur
H07Z9B: Software Architecture

Tomas Geens (r0597792)
Stijn Jochim (r0587702)
Afraz Salim (r0439731)

Contents

1. QAS Decisions

1.1 M1: Modifiability repercussions due to the Command data type

key architectural decisions:

- We removed the Command datatype
- We added seven new types of command: MoteCommand, which is an abstract super type of Actuation-Command and ConfigurationCommand, DatabaseCommand, DatabaseResultCommand, InterAppCommand and UpdateCommand

First, we all agreed that there should be multiple types of commands. We ended up settling on the six commands (excluding the MoteCommand) above. We did this so responses 1-4 are met. This way, for every syntax change of a specific type of command, only that command has to be changed.

A difficult decision was whether we would keep the broad Command as super type or not. We decided not to keep the command type, since there is no common ground between the commands: they all route differently and have different purposes. There were two commands where we did find similarities: the ActuationCommand and the ConfigurationCommand are both routed in a similar way, namely to a pluggable device. Therefore, we added a super type MoteCommand which contained the DeviceResourceLocator that is necessary to route to the correct pluggable device.

Another option we considered was to split The ApplicationCommandRouter into different command routers. For example, a DatabaseCommandRouter for Database- and DatabaseResultCommands or a MoteRouter for MoteCommands. We thought this could be good because it would reduce the coupling for the Application-CommandRouter, which is highly coupled right now. We ended up not choosing this. A reason for this is that splitting the ApplicationCommandRouter would reduce it's own coupling, but would increase coupling of other components. For example, The ApplicationHandler would need to be connected to every new router. Additionally, it would add some complexity to the design, whereas one ApplicationCommandRouter is simple to understand.

Before activating the application, it's requirements must be instantiated. This instantiation can either have only device's IDs or full DeviceResourceLocators. If we save all these DeviceResourceLocators in a database then we have extra seek time and transfer time. We decided to keep these DeviceResourceLocators in ApplicationContainer as local state of the application. Applications can consult these instantiated requirements anytime by using AppUsesAPI interface. The application will only get the ID's of the devices and not the whole DeviceResourceLocator.

The application sends a command which contains only the id of a specific device. This ID will be retrieved by the ApplicationContainer and compared to the all device ids in the DeviceResourceLocator of TopologyRequirementInstantiation. If any DeviceResourceLocator has that ID then that DeviceResourceLocator will be appended in that command before ApplicationContainer hands it over to the Router. This functionality can be achieved by overloading the constructor of commands where we allow the application to send different kind of commands with different type of parameters. The router has now full information where it must send that command. However for database requests there are no DeviceResourceLocators and we have to explicitly tell the router where it must send certain kind of commands. The routers are assumed to work together with built-in interpreters. InterAppCommands are routed using the ApplicationInstanceID and an optional GatwayID.

1.2 M2: New pluggable device types

key architectural decisions:

- We used the Adapter pattern to transform manufacturer specific API's into a general API.

- When a new pluggable interface is added for which the gateway does not have the correct adapter, it requests this adapter from the TopologyDB

The Adapter design pattern works by creating adapters that turn a specific API into a different (in our case general) API. This way pluggable devices that use a different API can be converted to a general API, defined by us. Right now this general API is based on MicroPnP, as we think any manufacturer of sensors and actuators will at least use an API similar to MicroPnP.

Because every device specific data type is converted using a specific adapter (ManufacturerSpecificDataAdapter on the diagrams) to a general SystemSensorData data-type, applications do not require any changes to work with these new pluggable devices. Actuation commands are sent by using the ActuationCommand type. Once again the adapter is responsible to convert these ActuationCommand into a command the new actuator can understand (e.g. for MicroPnP this is a String)

To handle new types of actuator commands or sensor data, a developer needs to create the code necessary to create the new adapter component in the gateway. Once he is done this code should be stored in the TopologyDB.

Whenever a new pluggable device is plugged in the DeviceStatusMonitor checks whether the gateway already has the capacity to handle this pluggable device (sequence diagram NewPluggableDevice). If it does not the code for the adapter is taken from the database and sent to the gateway where the adapter component is initialised.

The main drawback of this solution is that when the communication channel between the gateway and the online service is down, the gateway cannot not use the new pluggable device until the channel is restored. We also considered to not store the code in a database and instead send the code to every gateway immediately instead of only when the gateway requires the adapter. However even bigger problems would arise when the communication channel between gateway and online service goes down. The online service would need to remember somewhere which gateway has not received what adapter and would have to try sending it again later. Additionally it would require a lot of resources from the online service, potentially slowing down the entire system. For these reasons we chose to only send the correct adapter when the gateway requires it

1.3 U1: Usability of application API

key architectural decisions:

- A (GW)EventManager is responsible for managing events on either the online service or the gateway. Events are stored in a (GW)EventDataDB
- A new interface, (GW)AppUsesAPI was added to allow API calls from an application.

We chose to have an EventManager that operates closely to the ApplicationContainer and for an analogous approach for the Gateway application instance part. Usage of this EventManager is displayed in SaveEventToEventDB, GWReactToData (used in ProcessSensorData) and ReactToData (Used in RouteData). Events that require a certain condition to be true, work by first sending the data to the EventManager where the data is used to check whether the condition is met. If it is met, the Application is woken up. Events where the application need to be woken up by a certain time are handled as in TimeBasedEvent. The EventManager is responsible for activation at the right time. The final event, where applications need to be woken up after a set time of not receiving data, are also handled by the EventManager. The EventManager is responsible for keeping a timer for every event of this kind. This timer is refreshed every time data comes in. When the timer runs out the Application is notified as in TimeBasedEvent.

This solution seemed straight forward to us, so we did not look for many different approaches. We briefly considered a central EventManager for both application instances on the online service and on the gateway, but this seemed bad because critical applications on the Gateway should be woken up as soon as possible when an event happens. For this reason we chose separate EventManagers and EventDataDBs on the online service and gateways

A possible problem with our solution is that it is impossible for application on the online service to schedule an event for a GWApplication or vice versa. Another problem is security. Right now no limit is imposed on how many events an application can schedule. This is problematic, especially on the Gateway where storage is limited.

For the second response measure we added a new interface provided by (GW)ApplicationContainer and the (GW)ApplicationContainer, (GW)AppUsesAPI. This interface allows sending different kinds of commands to the (GW)ApplicationContainer whenever it wants as well as saving it's state and schedule events. Because we chose to remove the command datatype, An application can no longer just return a list of commands. Therefore it has to use this new interface to send all commands.

1.4 Av1: Application instance recovery

key architectural decisions:

- A state database is added to the system.
- Application instances can save their state to this database, and their state will be automatically be saved before being updated.

Application instances can now save their state to a state database as well as rollback to a previously saved state. This database will hold all the saved states for up to 2 weeks. The new database is accompanied by two new components, the AppStateDataScheduler and the AppStateDataLoadBalancer. We did this because critical applications on the gateway should get priority when getting their state, so they can rollback as fast as possible.

The responsibility for the scheduler is to make the read and write requests to the database and remove states that are older than 2 weeks. The load balancer is responsible for sending database queries to the correct server on which part of the application state data is located.

In case of an application instance crash, the application container will try to rollback to the last saved state. If this does not succeed after 3 tries, the application instance is suspended. In case of an update, the state is first saved. Then when the application is initialised again, it is done with the correct state.

A weakness of this implementation is that all states are stored on the online service. When a gateway application crashes, the state needs to be recovered from the online service. Especially for critical applications this could be problematic (e.g. when the communication channel is down). However we opted for this solution since storage on the gateway is limited and we already added an EventDataDB (1.3). A future modification could be to store states from critical applications on the gateway.

1.5 Sec1: Untrusted Applications

key architectural decisions:

- We added a processID and a storage monitor.

We decided to run the application instances on the online service and it's counter part on the gateway with the same application instance ID. We explicitly made difference between ApplicationInstanceID and processID. ProcessID is determined by the OS but applicationInstanceID can be determined (uniquely) by the ApplicationStarter or with any other attributes as mentioned in initial architecture. We let the ApplicationScheduler store the applicationInstanceID and it's container in a map. ApplicationScheduler will either find the instance of the container from it's map upon receiving a request or it will get the null value. This makes it impossible for ApplicationScheduler to pass a command to a wrong container. An alternative strategy where we allow the application instance on online service to have a different applicationInstanceID then it's counter part and online service would require us to save three values instead of two and this can be avoided by simply making difference between ProcessID and applicationInstanceID.

All kinds of commands pass through the ApplicationContainer. ApplicationContainer has the requirements for each application instance it runs. Before sending out the command, ApplicationContainer has to check if the deviceID is in the instantiated requirements or in the list of application requirements. When we request data from the database, we can also check that the requested data from a specific sensor is in our requirement list. If the devices are not found in the list then a violation is registered and no database request will be sent to the database. An alternative solution would be to explicitly check from the AppInstanceInfoDB that the application instance is associated with the application instance which sent this request but checking the condition as early as possible is a better option.

We let the application developer determine the max amount of memory that his application will use. If SIOTIP accepts that application then after starting the application we can register this application to a storage monitor along with it's processID. The processID is obtained from the kernel by providing applicationInstanceID or it's name. As an alternative we can check the memory of the application in ApplicationContainer but it would require ApplicationContainer to constantly ask update of the memory from the OS and this can drastically effect the performance of the application if the load increases.

Any application which violates the constraints will be suspended if it reaches 20 violations within 24 hours. Each application's local state is stored in ApplicationContainer and we decided that every command, including MobileAppComamnds, must pass through ApplicationContainer. ApplicationContainer seems a better place to control the request per second. A local counter is maintained which asynchronously gets reset after 10 seconds and is incremented if a request is sent from application. Any violation gets stored and reaching the maximum would limit results in suspension of the application.

1.6 P1: Large number of users

No explicit changes were made to the architecture to accommodate the needs of an increasing customer base. We noticed that when the number of gateways increase The OSWithGWCommunication becomes the bottleneck for all incoming traffic. This could be solved using multiple OnlineServiceNodes. We thought about deploying these nodes and their applications per geographical location, but this has some problems. What if a customer organisation starts a settlement in a different province? Do these new gateways connent to a different OnlineServiceNode or to the same? What if there are too many gateways in a province? Should the province then be divided, how could this be done? For these reasons we did not divide the online service in different nodes based on geography.

Another option we considered was having multiple OnlineServiceNodes, but allow every gateway to connect to every node. However this would complicate routing quite a lot. For example, how does a gateway know what node it's online service counterpart is running on? We would also need to change the infrastructure to accommodate for applications changing the node they run on when the communication channel is overloaded. In hindsight this is what we should have implemented, but unfortunately we found this solution too late and did not have time to change the architecture.

Executing a larger amount of application would be a bit simpler. we could have multiple ApplicationExectionNodes on every online service. An ApplicationExectionLoadBalancer would be required to route to the correct node, this should not be very hard.

2. Discussion

In retrospect, we should have started with 1.6. We first thought that this would not make a big impact on the architecture and that we could add it at the end. But now we see that this impacts the architecture in a big way. Overall we implemented solutions for five Quality Attribute Scenarios that we think would work. However, some did impose new architectural risk as mentioned in 1.2 and 1.3.

By not using inheritance for the commands, we cannot return a list of commands like was done in many cases. Now the application sends the commands directly to the CommandRouter (instead of returning a list). In our design we made use of the fact that applications are state machines. This made it easy to store and save different states, rollback to those states, and update applications with the same state as before the update.

A. Client-server view (UML Component Diagram)

Figures

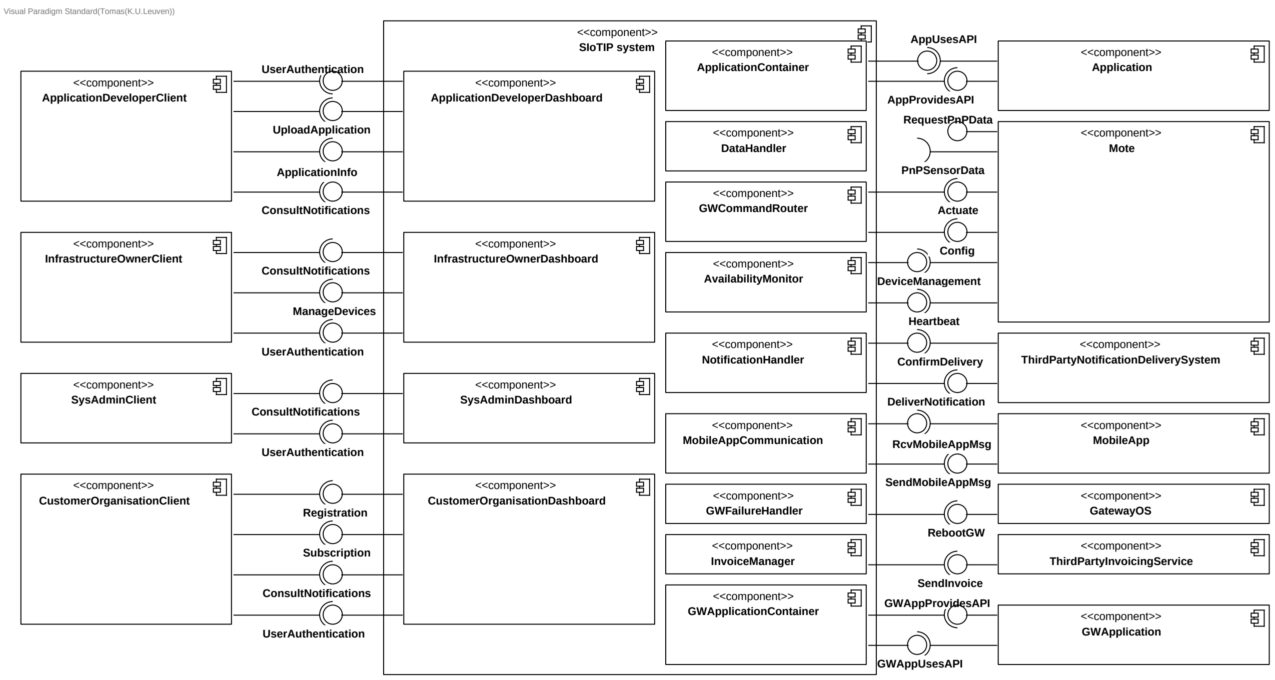

Visual Paradigm Standard(Tomas(K.U.Leuven))
<<component>> SIoTIP system
<<component>> ApplicationDeveloperClient
<<component>> InfrastructureOwnerClient
<<component>> SysAdminClient
<<component>> CustomerOrganisationClient
<<component>> ApplicationDeveloperDashboard
<<component>> InfrastructureOwnerDashboard
<<component>> SysAdminDashboard
<<component>> CustomerOrganisationDashboard
<<component>> ApplicationContainer
<<component>> DataHandler
<<component>> GWCommandRouter
<<component>> AvailabilityMonitor
<<component>> NotificationHandler
<<component>> MobileAppCommunication
<<component>> GWFailureHandler
<<component>> InvoiceManager
<<component>> GWApplicationContainer
<<component>> Application
<<component>> Mote
<<component>> ThirdPartyNotificationDeliverySystem
<<component>> MobileApp
<<component>> GatewayOS
<<component>> ThirdPartyInvoicingService
<<component>> GWApplication
UserAuthentication
UploadApplication
ApplicationInfo
ConsultNotifications
ManageDevices
Registration
Subscription
AppUsesAPI
AppProvidesAPI
RequestPnPData
PnPSensorData
Actuate
Config
DeviceManagement
Heartbeat
ConfirmDelivery
DeliverNotification
RcvMobileAppMsg
SendMobileAppMsg
RebootGW
SendInvoice
GWAppProvidesAPI
GWAppUsesAPI

Figure A.1: Context diagram for the client-server view. Note that the application execution subsystem on the `Gateway` a variation is on the one on the Online Service. This is because of the difference in computing power between the two and thus the difference in solution possibilities.

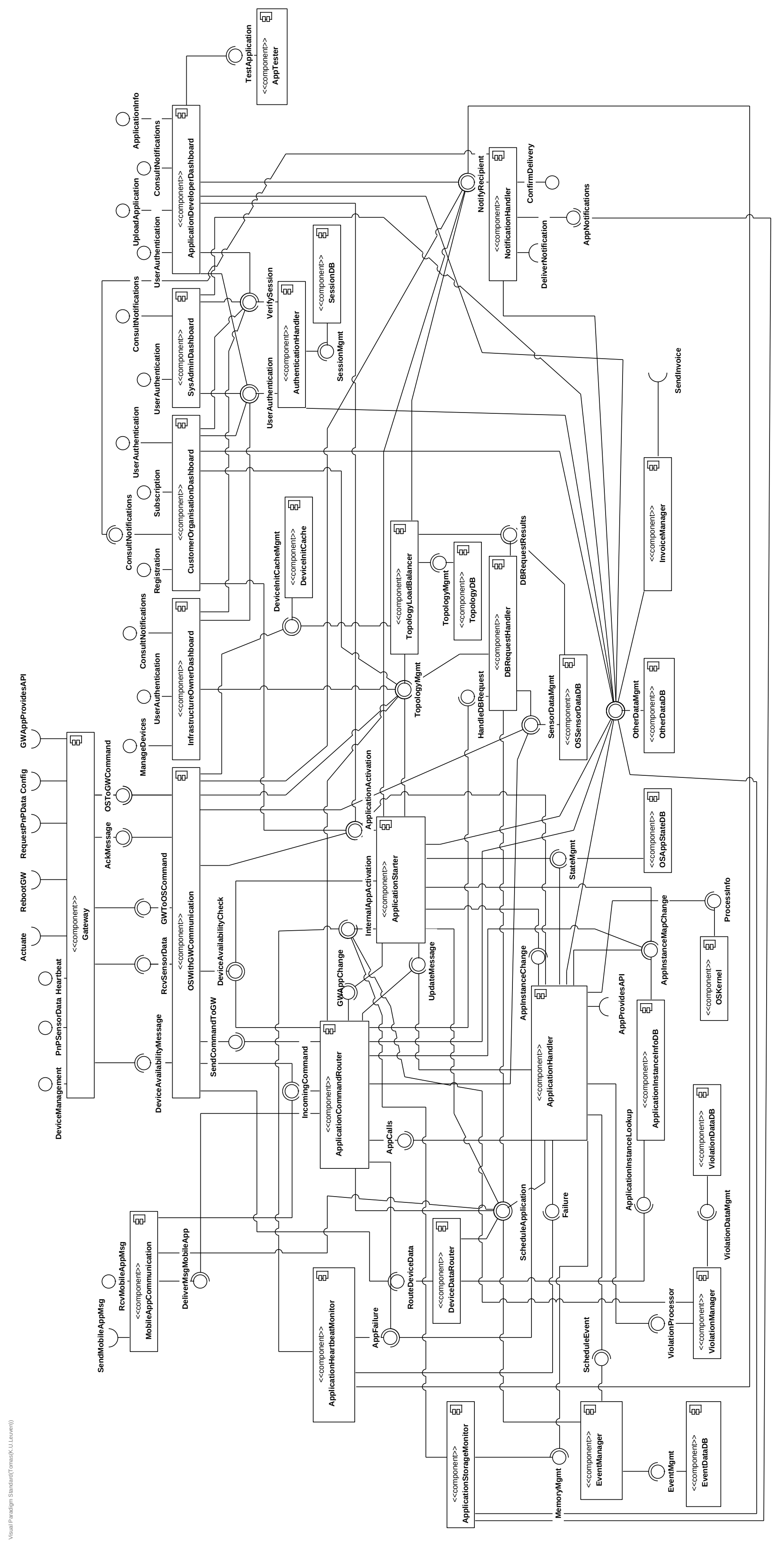


Figure A.2: Primary diagram of the client-server view.

B. Decompositions (UML Component Diagram

Figures

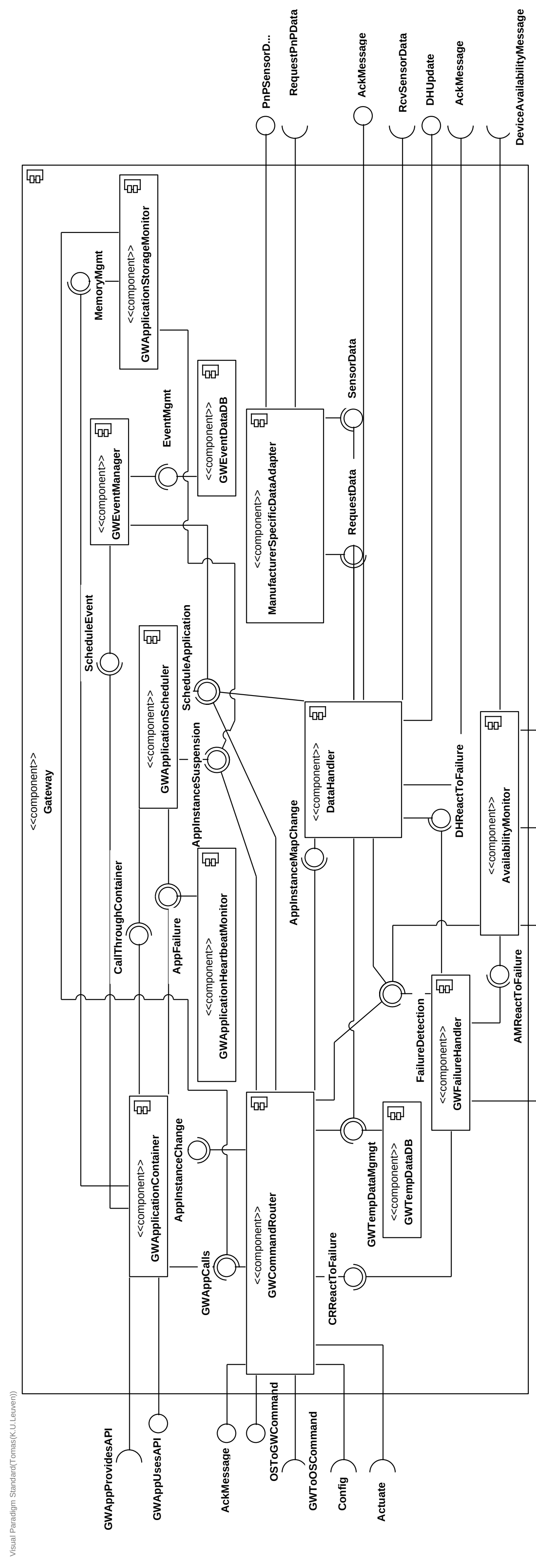


Figure B.1: Decomposition of Gateway.

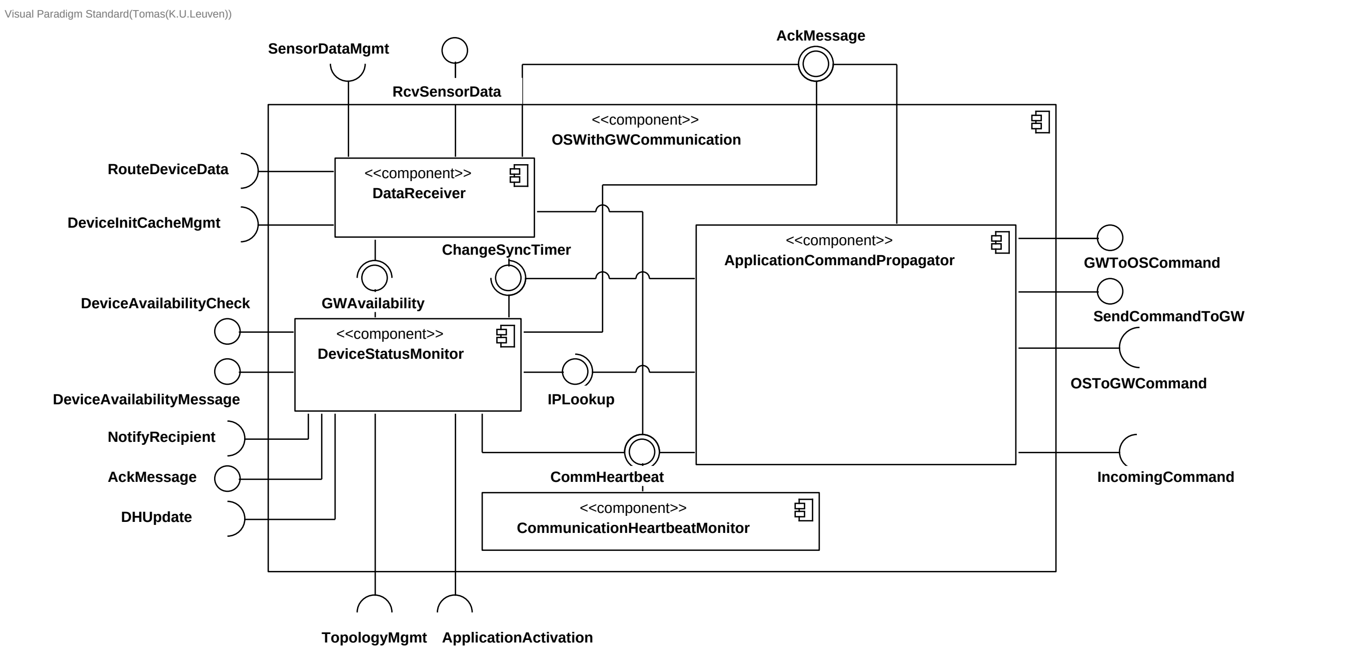


Figure B.2: Decomposition of `OSWithGWCommunication`.

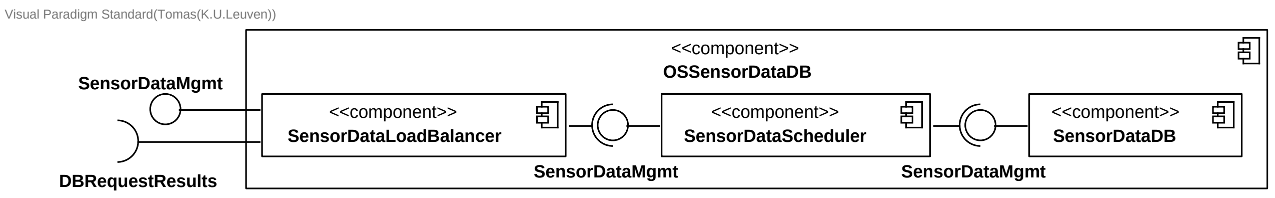


Figure B.3: Decomposition of `OSSensorDataDB`.

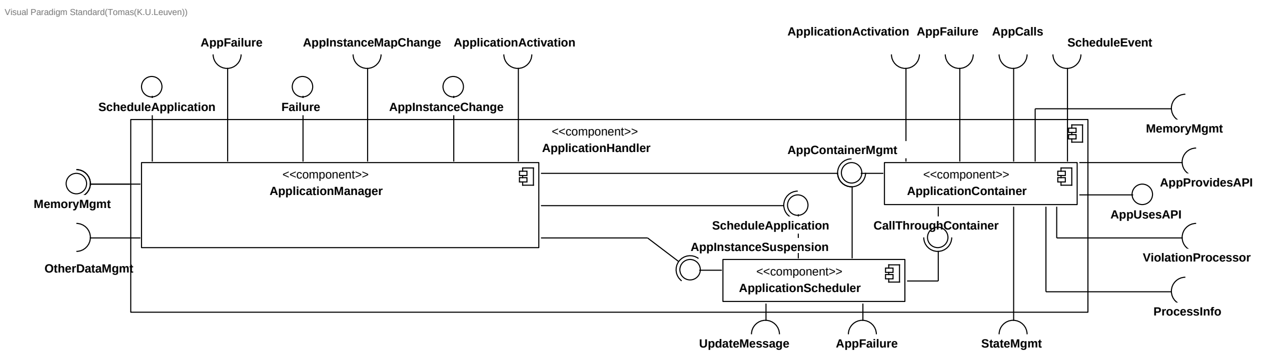


Figure B.4: Decomposition of `ApplicationHandler`. Dangling internal interfaces at the stand-in components represent available interfaces which are not connected at all times.

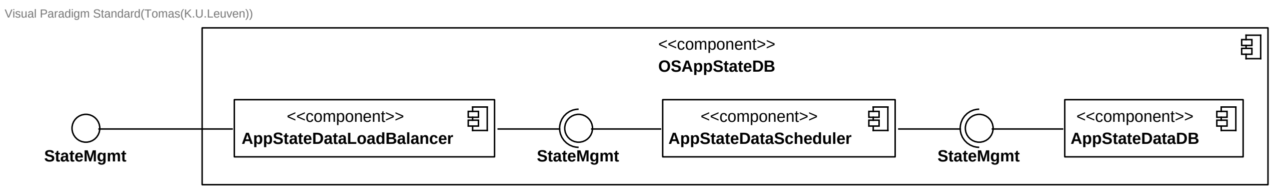


Figure B.5: dec_OSAppStateDB

C. Deployment view (UML Deployment Diagram)

Figures

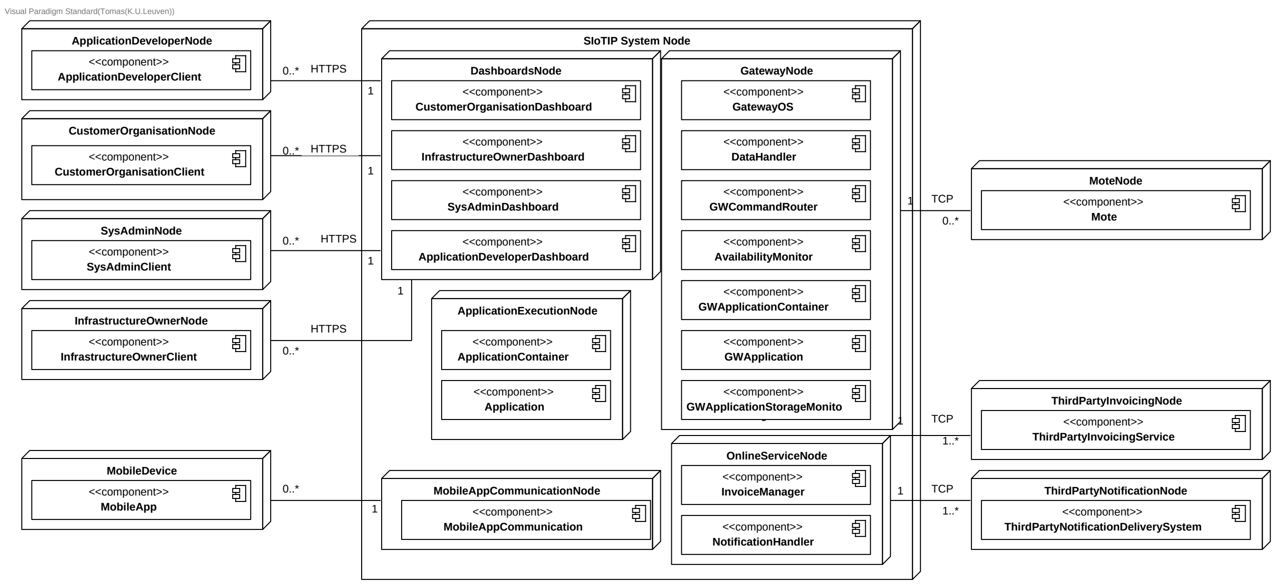

Visual Paradigm Standard(Tomas(K.U.Leuven))
ApplicationDeveloperNode
<<component>> ApplicationDeveloperClient
CustomerOrganisationNode
<<component>> CustomerOrganisationClient
SysAdminNode
<<component>> SysAdminClient
InfrastructureOwnerNode
<<component>> InfrastructureOwnerClient
MobileDevice
<<component>> MobileApp
SIoTIP System Node
DashboardsNode
<<component>> CustomerOrganisationDashboard
<<component>> InfrastructureOwnerDashboard
<<component>> SysAdminDashboard
<<component>> ApplicationDeveloperDashboard
ApplicationExecutionNode
<<component>> ApplicationContainer
<<component>> Application
MobileAppCommunicationNode
<<component>> MobileAppCommunication
GatewayNode
<<component>> GatewayOS
<<component>> DataHandler
<<component>> GWCommandRouter
<<component>> AvailabilityMonitor
<<component>> GWApplicationContainer
<<component>> GWApplication
<<component>> GWApplicationStorageMonito
OnlineServiceNode
<<component>> InvoiceManager
<<component>> NotificationHandler
MoteNode
<<component>> Mote
ThirdPartyInvoicingNode
<<component>> ThirdPartyInvoicingService
ThirdPartyNotificationNode
<<component>> ThirdPartyNotificationDeliverySystem
HTTPS
0..*
1
TCP
1..*

Figure C.1: Context diagram for the deployment view. Note that TCP is used for communication with the mote since loss of packets (and thus device data) is unacceptable. The various clients connect to the system via a web browser and thus HTTPS is used for those connections. The rest of the connections are done over TCP to avoid packet loss.

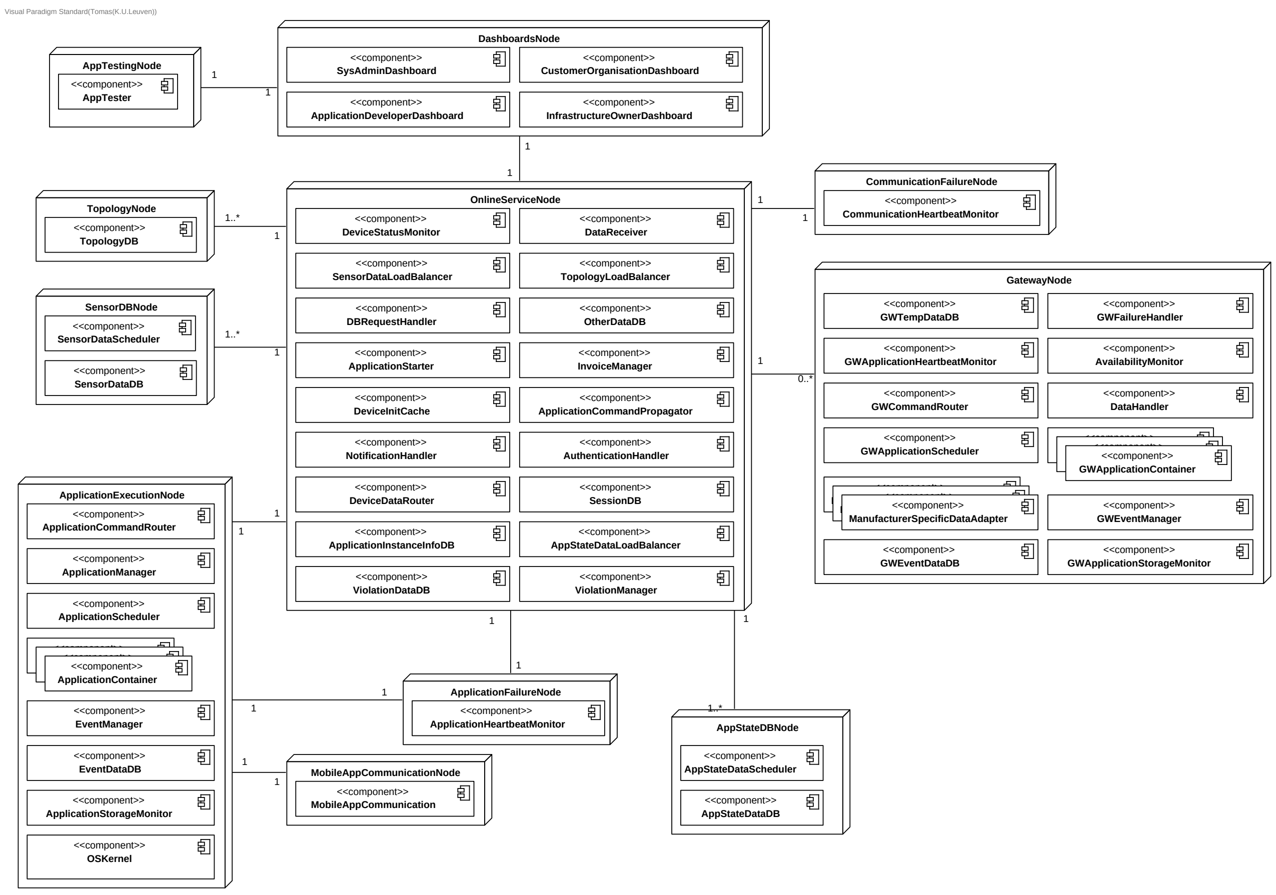

Visual Paradigm Standard(Tomas(K.U.Leuven))
DashboardsNode
<<component>> SysAdminDashboard
<<component>> CustomerOrganisationDashboard
<<component>> ApplicationDeveloperDashboard
<<component>> InfrastructureOwnerDashboard
AppTestingNode
<<component>> AppTester
TopologyNode
<<component>> TopologyDB
SensorDBNode
<<component>> SensorDataScheduler
<<component>> SensorDataDB
OnlineServiceNode
<<component>> DeviceStatusMonitor
<<component>> DataReceiver
<<component>> SensorDataLoadBalancer
<<component>> TopologyLoadBalancer
<<component>> DBRequestHandler
<<component>> OtherDataDB
<<component>> ApplicationStarter
<<component>> InvoiceManager
<<component>> DeviceInitCache
<<component>> ApplicationCommandPropagator
<<component>> NotificationHandler
<<component>> AuthenticationHandler
<<component>> DeviceDataRouter
<<component>> SessionDB
<<component>> ApplicationInstanceInfoDB
<<component>> AppStateDataLoadBalancer
<<component>> ViolationDataDB
<<component>> ViolationManager
CommunicationFailureNode
<<component>> CommunicationHeartbeatMonitor
GatewayNode
<<component>> GWTempDataDB
<<component>> GWFailureHandler
<<component>> GWApplicationHeartbeatMonitor
<<component>> AvailabilityMonitor
<<component>> GWCommandRouter
<<component>> DataHandler
<<component>> GWApplicationScheduler
<<component>> GWApplicationContainer
<<component>> ManufacturerSpecificDataAdapter
<<component>> GWEventManager
<<component>> GWEventDataDB
<<component>> GWApplicationStorageMonitor
ApplicationExecutionNode
<<component>> ApplicationCommandRouter
<<component>> ApplicationManager
<<component>> ApplicationScheduler
<<component>> ApplicationContainer
<<component>> EventManager
<<component>> EventDataDB
<<component>> ApplicationStorageMonitor
<<component>> OSKernel
ApplicationFailureNode
<<component>> ApplicationHeartbeatMonitor
MobileAppCommunicationNode
<<component>> MobileAppCommunication
AppStateDBNode
<<component>> AppStateDataScheduler
<<component>> AppStateDataDB
1
1..*
0..*

Figure C.2: Primary diagram for de deployment view.

D. Scenarios (UML Sequence Diagram)

Figures

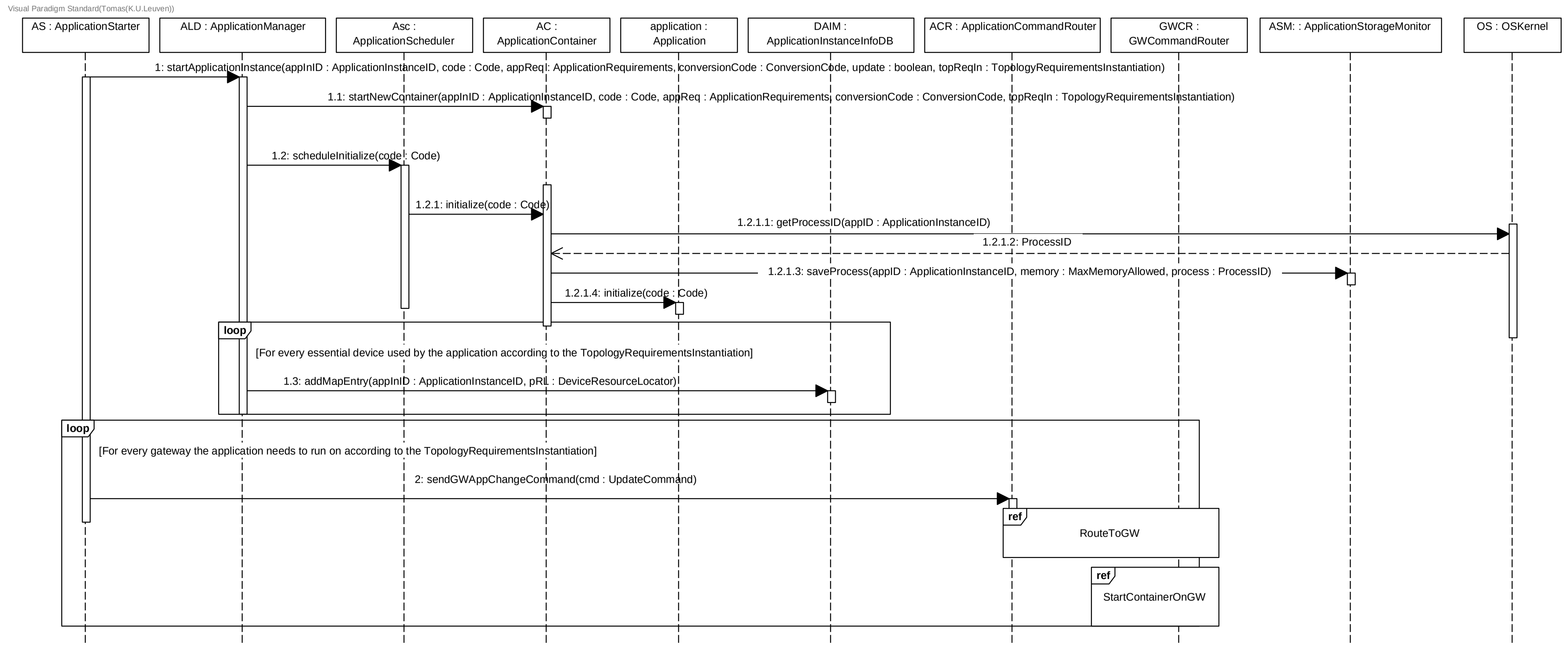

Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationStarter
ALD : ApplicationManager
Asc : ApplicationScheduler
AC : ApplicationContainer
application : Application
DAIM : ApplicationInstanceInfoDB
ACR : ApplicationCommandRouter
GWCR : GWCommandRouter
ASM: : ApplicationStorageMonitor
OS : OSKernel
1: startApplicationInstance(appInID : ApplicationInstanceID, code : Code, appReq : ApplicationRequirements, conversionCode : ConversionCode, update : boolean, topReqIn : TopologyRequirementsInstantiation)
1.1: startNewContainer(appInID : ApplicationInstanceID, code : Code, appReq : ApplicationRequirements, conversionCode : ConversionCode, topReqIn : TopologyRequirementsInstantiation)
1.2: scheduleInitialize(code : Code)
1.2.1: initialize(code : Code)
1.2.1.1: getProcessID(appID : ApplicationInstanceID)
1.2.1.2: ProcessID
1.2.1.3: saveProcess(appID : ApplicationInstanceID, memory : MaxMemoryAllowed, process : ProcessID)
1.2.1.4: initialize(code : Code)
loop
[For every essential device used by the application according to the TopologyRequirementsInstantiation]
1.3: addMapEntry(appInID : ApplicationInstanceID, pRL : DeviceResourceLocator)
loop
[For every gateway the application needs to run on according to the TopologyRequirementsInstantiation]
2: sendGWAppChangeCommand(cmd : UpdateCommand)
ref
RouteToGW
ref
StartContainerOnGW

Figure D.1: Note the addition of entries to a map used for routing sensor data to the correct application instances.

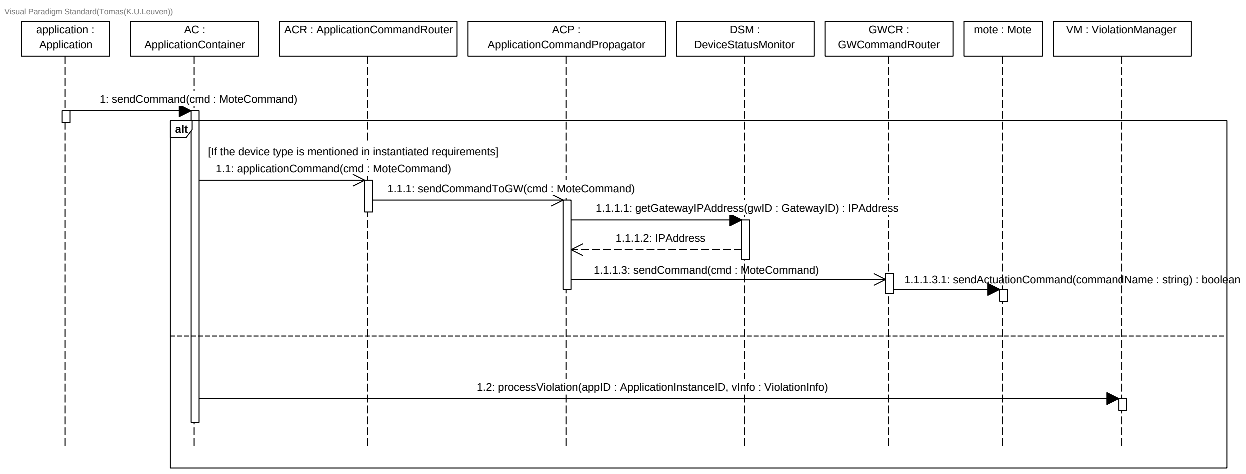

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
AC : ApplicationContainer
ACR : ApplicationCommandRouter
ACP : ApplicationCommandPropagator
DSM : DeviceStatusMonitor
GWCR : GWCommandRouter
mote : Mote
VM : ViolationManager
1: sendCommand(cmd : MoteCommand)
alt
[If the device type is mentioned in instantiated requirements]
1.1: applicationCommand(cmd : MoteCommand)
1.1.1: sendCommandToGW(cmd : MoteCommand)
1.1.1.1: getGatewayIPAddress(gwID : GatewayID) : IPAddress
1.1.1.2: IPAddress
1.1.1.3: sendCommand(cmd : MoteCommand)
1.1.1.3.1: sendActuationCommand(commandName : string) : boolean
1.2: processViolation(appID : ApplicationInstanceID, vInfo : ViolationInfo)

Figure D.2: Make sure an application command for an actuator makes it to that actuator.

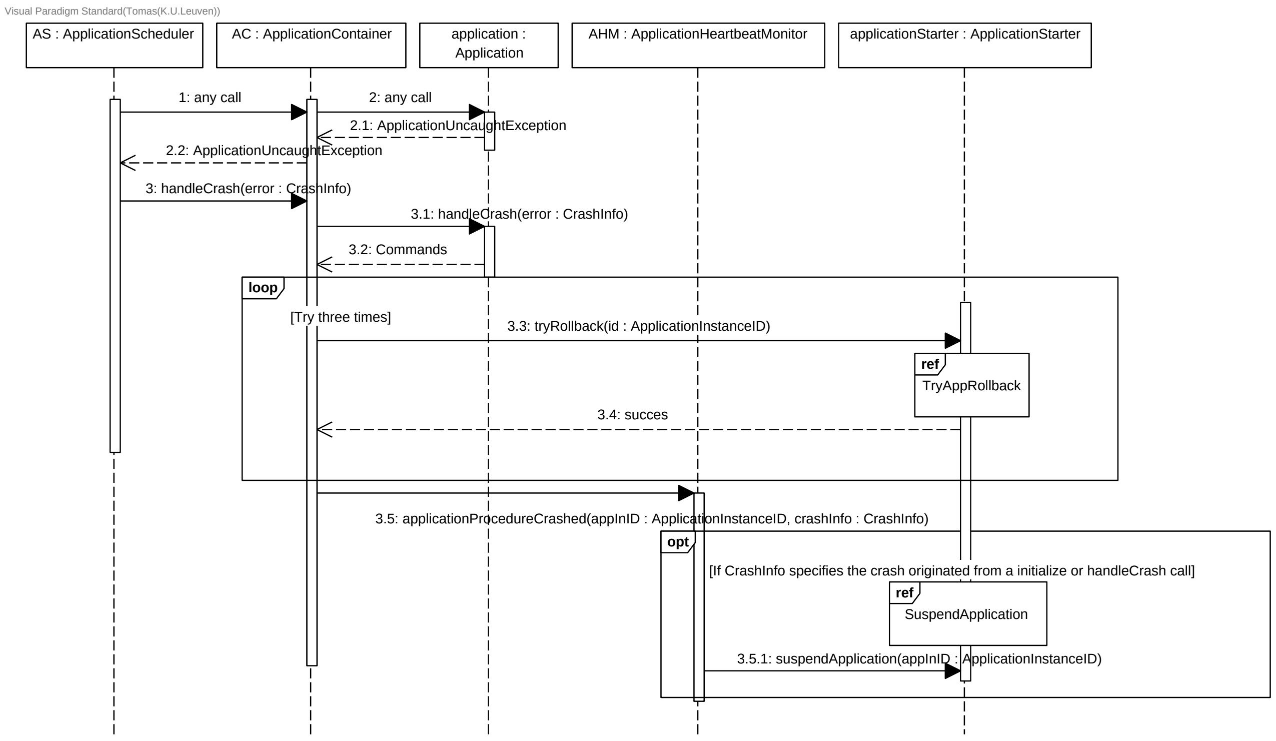

Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationScheduler
AC : ApplicationContainer
application : Application
AHM : ApplicationHeartbeatMonitor
applicationStarter : ApplicationStarter
1: any call
2: any call
2.1: ApplicationUncaughtException
2.2: ApplicationUncaughtException
3: handleCrash(error : CrashInfo)
3.1: handleCrash(error : CrashInfo)
3.2: Commands
loop
[Try three times]
3.3: tryRollback(id : ApplicationInstanceID)
ref
TryAppRollback
3.4: succes
3.5: applicationProcedureCrashed(appInID : ApplicationInstanceID, crashInfo : CrashInfo)
opt
[If CrashInfo specifies the crash originated from a initialize or handleCrash call]
ref
SuspendApplication
3.5.1: suspendApplication(appInID : ApplicationInstanceID)

Figure D.3: ApplicationProcedureCrash

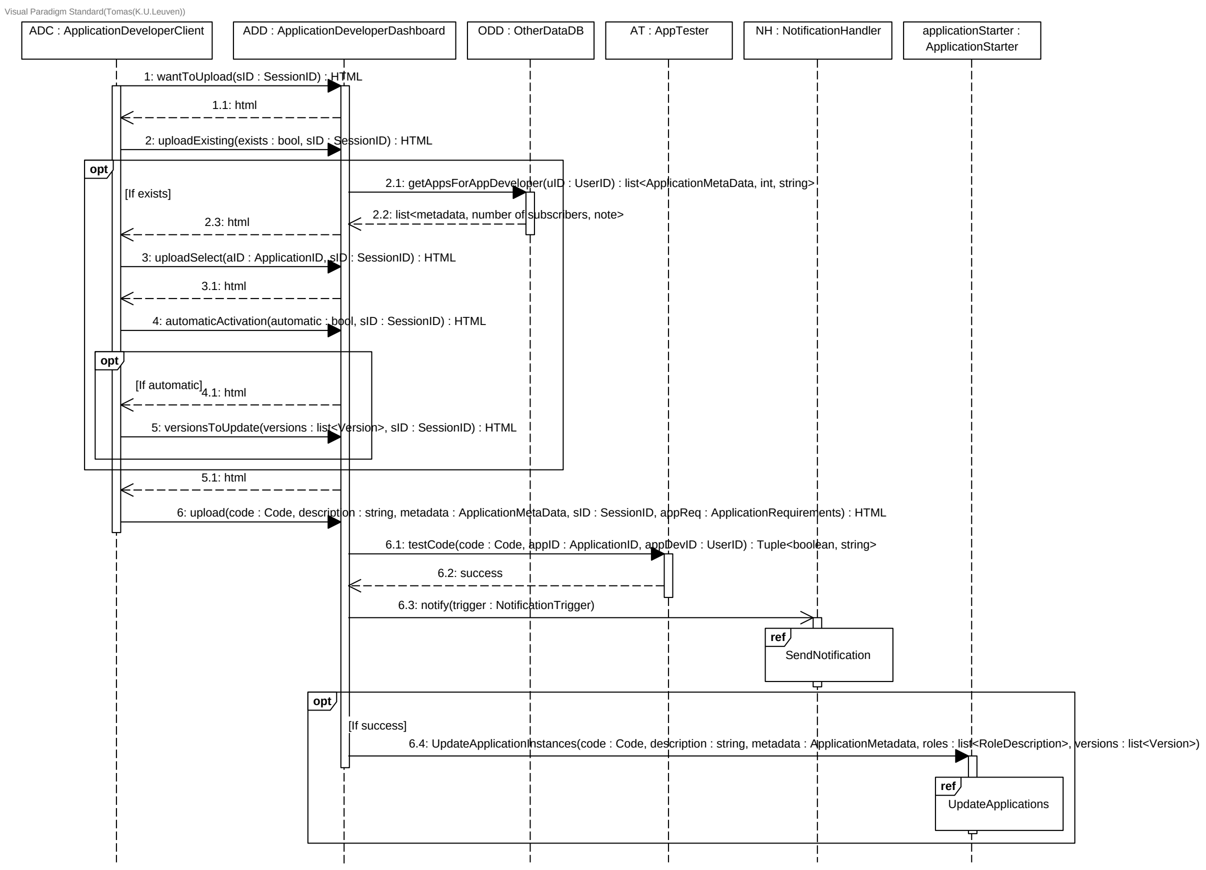

Visual Paradigm Standard(Tomas(K.U.Leuven))
ADC : ApplicationDeveloperClient
ADD : ApplicationDeveloperDashboard
ODD : OtherDataDB
AT : AppTester
NH : NotificationHandler
applicationStarter : ApplicationStarter
1: wantToUpload(sID : SessionID) : HTML
1.1: html
2: uploadExisting(exists : bool, sID : SessionID) : HTML
opt
[If exists]
2.1: getAppsForAppDeveloper(uID : UserID) : list<ApplicationMetaData, int, string>
2.2: list<metadata, number of subscribers, note>
2.3: html
3: uploadSelect(aID : ApplicationID, sID : SessionID) : HTML
3.1: html
4: automaticActivation(automatic : bool, sID : SessionID) : HTML
opt
[If automatic]
4.1: html
5: versionsToUpdate(versions : list<Version>, sID : SessionID) : HTML
5.1: html
6: upload(code : Code, description : string, metadata : ApplicationMetaData, sID : SessionID, appReq : ApplicationRequirements) : HTML
6.1: testCode(code : Code, appID : ApplicationID, appDevID : UserID) : Tuple<boolean, string>
6.2: success
6.3: notify(trigger : NotificationTrigger)
ref
SendNotification
opt
[If success]
6.4: UpdateApplicationInstances(code : Code, description : string, metadata : ApplicationMetadata, roles : list<RoleDescription>, versions : list<Version>)
ref
UpdateApplications

Figure D.4: An application developer uploads his or her application, indicating in the process whether this is a new version of an existing application and, if so, which previous versions should be automatically updated.

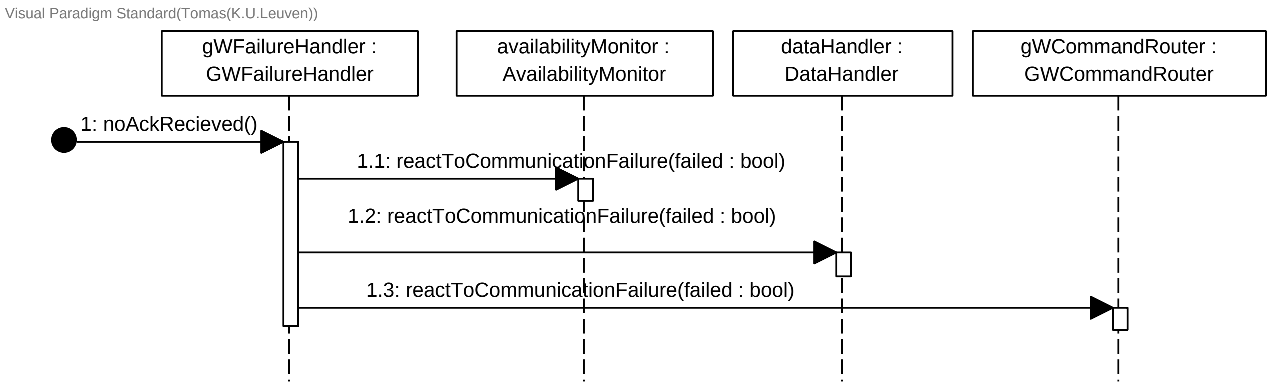

Visual Paradigm Standard(Tomas(K.U.Leuven))
gWFailureHandler : GWFailureHandler
availabilityMonitor : AvailabilityMonitor
dataHandler : DataHandler
gWCommandRouter : GWCommandRouter
1: noAckRecieved()
1.1: reactToCommunicationFailure(failed : bool)
1.2: reactToCommunicationFailure(failed : bool)
1.3: reactToCommunicationFailure(failed : bool)

Figure D.5: CommunicationChannelFailure(GW)

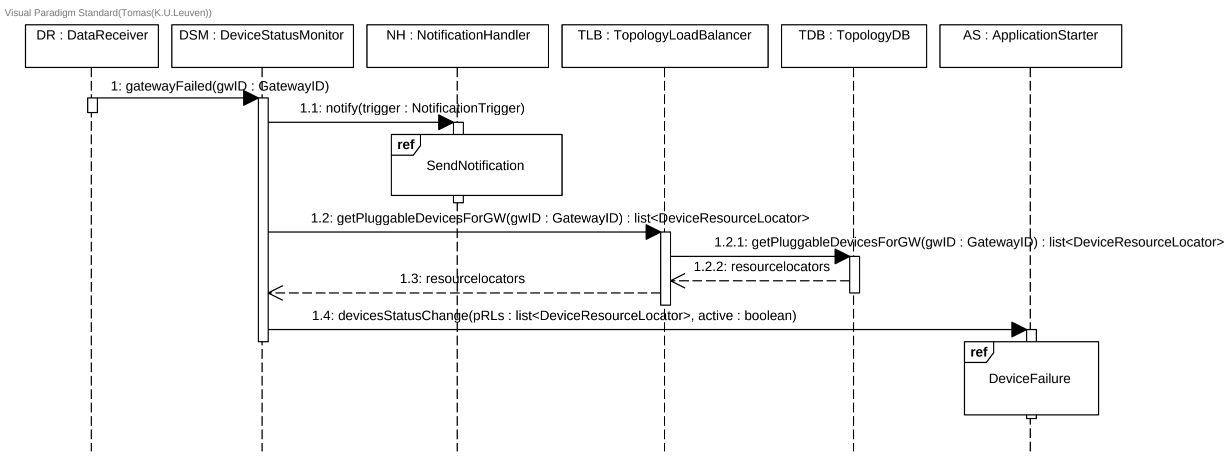

Visual Paradigm Standard(Tomas(K.U.Leuven))
DR : DataReceiver
DSM : DeviceStatusMonitor
NH : NotificationHandler
TLB : TopologyLoadBalancer
TDB : TopologyDB
AS : ApplicationStarter
1: gatewayFailed(gwID : GatewayID)
1.1: notify(trigger : NotificationTrigger)
ref
SendNotification
1.2: getPluggableDevicesForGW(gwID : GatewayID) : list<DeviceResourceLocator>
1.2.1: getPluggableDevicesForGW(gwID : GatewayID) : list<DeviceResourceLocator>
1.2.2: resourcelocators
1.3: resourcelocators
1.4: devicesStatusChange(pRLs : list<DeviceResourceLocator>, active : boolean)
ref
DeviceFailure

Figure D.6: CommunicationChannelFailure(OS)

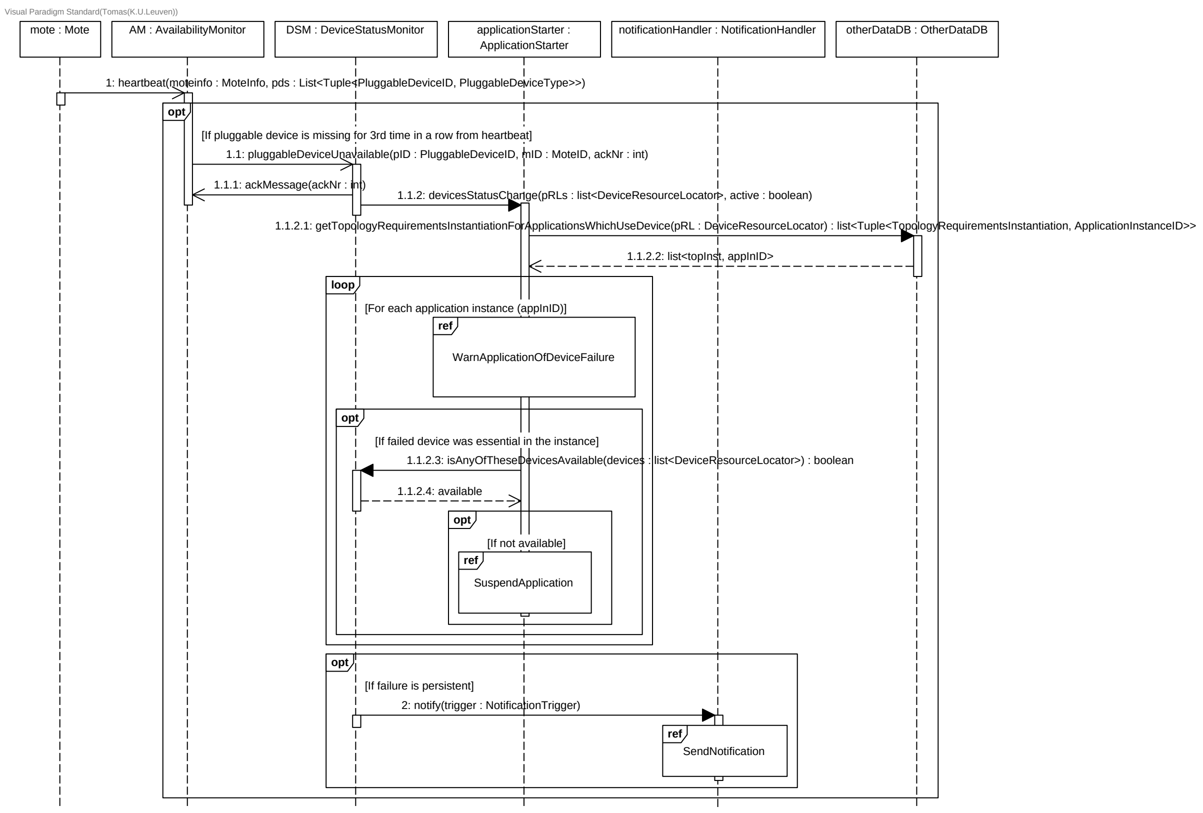

Visual Paradigm Standard(Tomas(K.U.Leuven))
mote : Mote
AM : AvailabilityMonitor
DSM : DeviceStatusMonitor
applicationStarter : ApplicationStarter
notificationHandler : NotificationHandler
otherDataDB : OtherDataDB
1: heartbeat(moteinfo : MoteInfo, pds : List<Tuple<PluggableDeviceID, PluggableDeviceType>>)
opt
[If pluggable device is missing for 3rd time in a row from heartbeat]
1.1: pluggableDeviceUnavailable(pID : PluggableDeviceID, mID : MoteID, ackNr : int)
1.1.1: ackMessage(ackNr : int)
1.1.2: devicesStatusChange(pRLs : list<DeviceResourceLocator>, active : boolean)
1.1.2.1: getTopologyRequirementsInstantiationForApplicationsWhichUseDevice(pRL : DeviceResourceLocator) : list<Tuple<TopologyRequirementsInstantiation, ApplicationInstanceID>>
1.1.2.2: list<topInst, appInID>
loop
[For each application instance (appInID)]
ref
WarnApplicationOfDeviceFailure
opt
[If failed device was essential in the instance]
1.1.2.3: isAnyOfTheseDevicesAvailable(devices : list<DeviceResourceLocator>) : boolean
1.1.2.4: available
opt
[If not available]
ref
SuspendApplication
opt
[If failure is persistent]
2: notify(trigger : NotificationTrigger)
ref
SendNotification

Figure D.7: A device is no longer sending out heartbeats. The system responds by figuring out which applications are affected by this failure, checking whether they can still run without the failed device, and suspending them if they can't. If they can still run, the application is informed of the failure so it can leverage any redundancies if possible (by asking sensor data from other accessible pluggable devices).

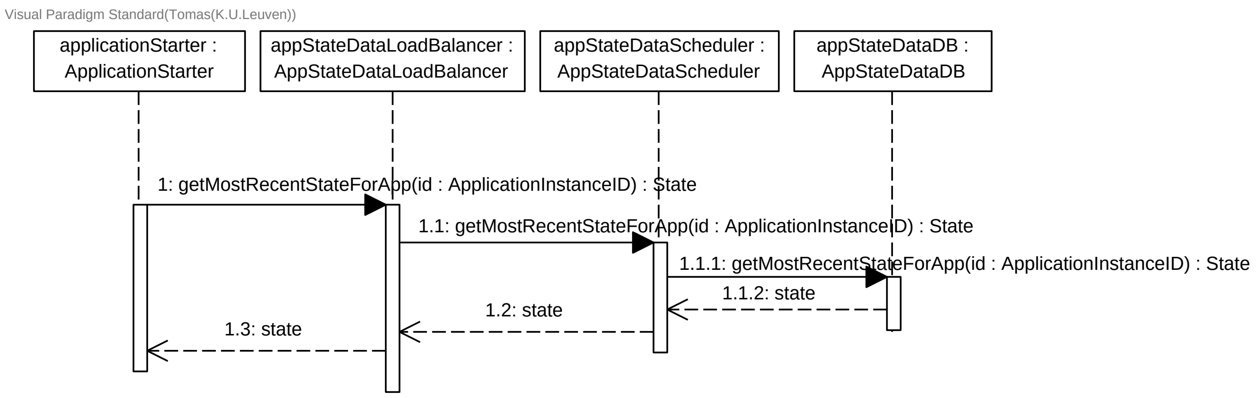

Visual Paradigm Standard(Tomas(K.U.Leuven))
applicationStarter : ApplicationStarter
appStateDataLoadBalancer : AppStateDataLoadBalancer
appStateDataScheduler : AppStateDataScheduler
appStateDataDB : AppStateDataDB
1: getMostRecentStateForApp(id : ApplicationInstanceID) : State
1.1: getMostRecentStateForApp(id : ApplicationInstanceID) : State
1.1.1: getMostRecentStateForApp(id : ApplicationInstanceID) : State
1.1.2: state
1.2: state
1.3: state

Figure D.8: GetMostRecentState

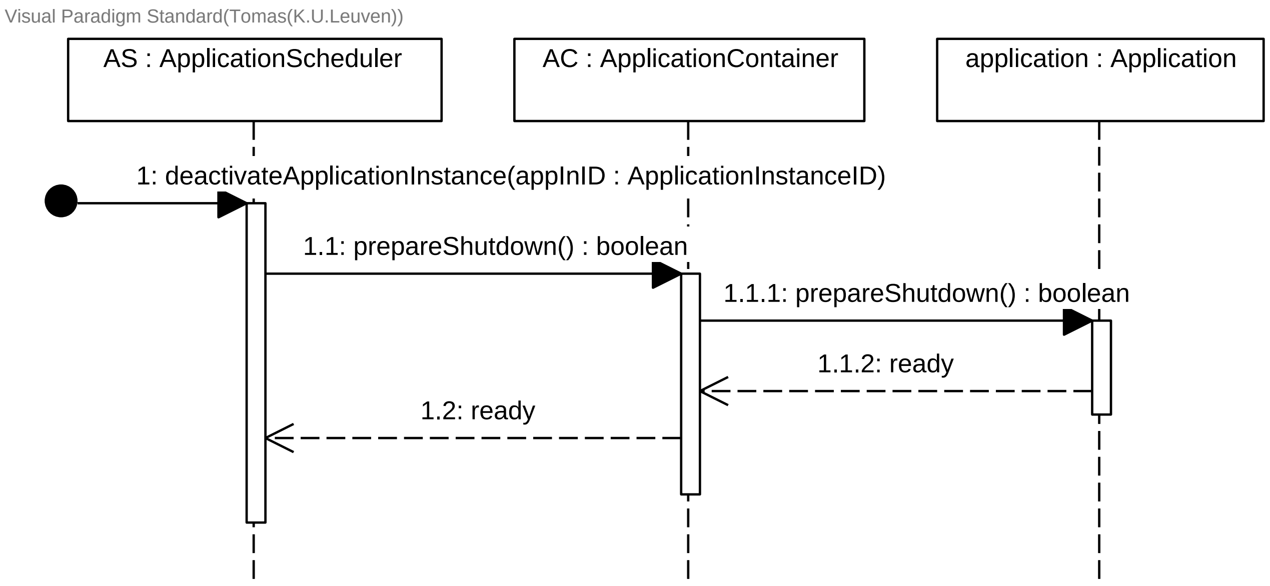

Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationScheduler
AC : ApplicationContainer
application : Application
1: deactivateApplicationInstance(appInID : ApplicationInstanceID)
1.1: prepareShutdown() : boolean
1.1.1: prepareShutdown() : boolean
1.1.2: ready
1.2: ready

Figure D.9: KillContainer

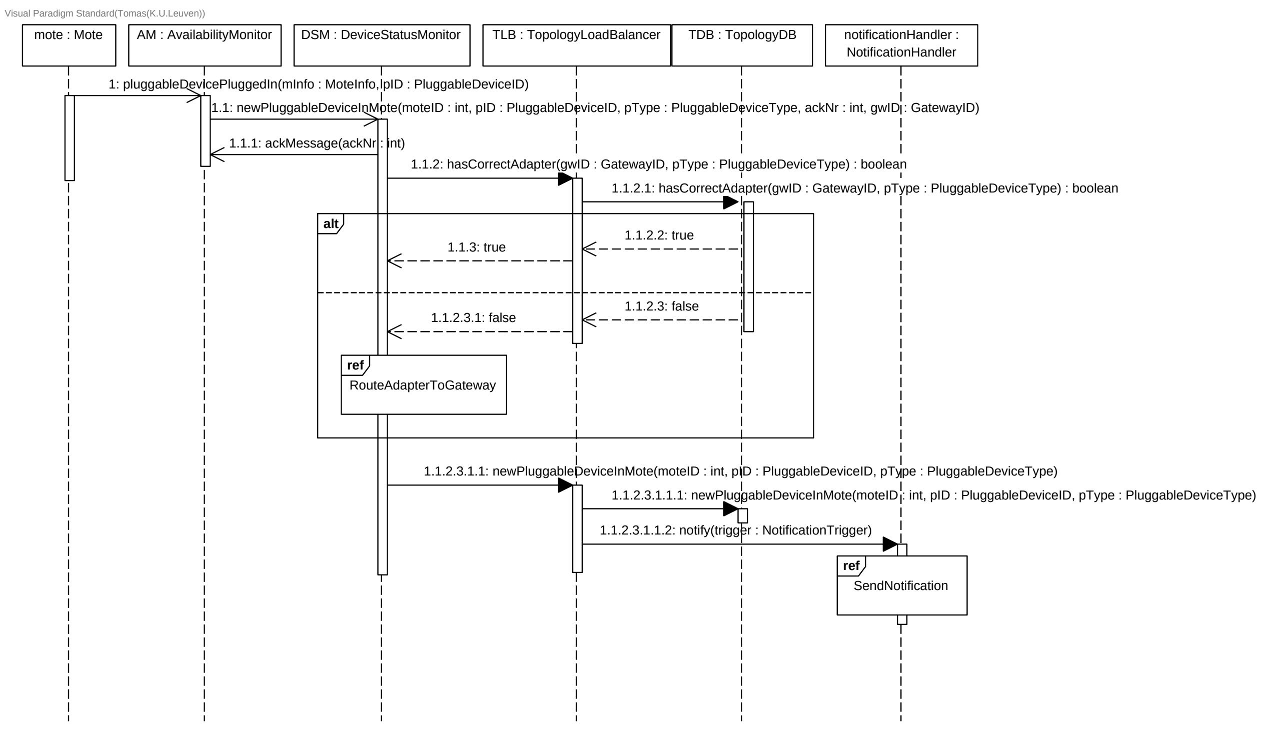

Visual Paradigm Standard(Tomas(K.U.Leuven))
mote : Mote
AM : AvailabilityMonitor
DSM : DeviceStatusMonitor
TLB : TopologyLoadBalancer
TDB : TopologyDB
notificationHandler : NotificationHandler
1: pluggableDevicePluggedIn(mInfo : MoteInfo, pID : PluggableDeviceID)
1.1: newPluggableDeviceInMote(moteID : int, pID : PluggableDeviceID, pType : PluggableDeviceType, ackNr : int, gwID : GatewayID)
1.1.1: ackMessage(ackNr : int)
1.1.2: hasCorrectAdapter(gwID : GatewayID, pType : PluggableDeviceType) : boolean
1.1.2.1: hasCorrectAdapter(gwID : GatewayID, pType : PluggableDeviceType) : boolean
alt
1.1.2.2: true
1.1.3: true
1.1.2.3: false
1.1.2.3.1: false
ref
RouteAdapterToGateway
1.1.2.3.1.1: newPluggableDeviceInMote(moteID : int, pID : PluggableDeviceID, pType : PluggableDeviceType)
1.1.2.3.1.1.1: newPluggableDeviceInMote(moteID : int, pID : PluggableDeviceID, pType : PluggableDeviceType)
1.1.2.3.1.1.2: notify(trigger : NotificationTrigger)
ref
SendNotification

Figure D.10: A new pluggable device is plugged into a mote.

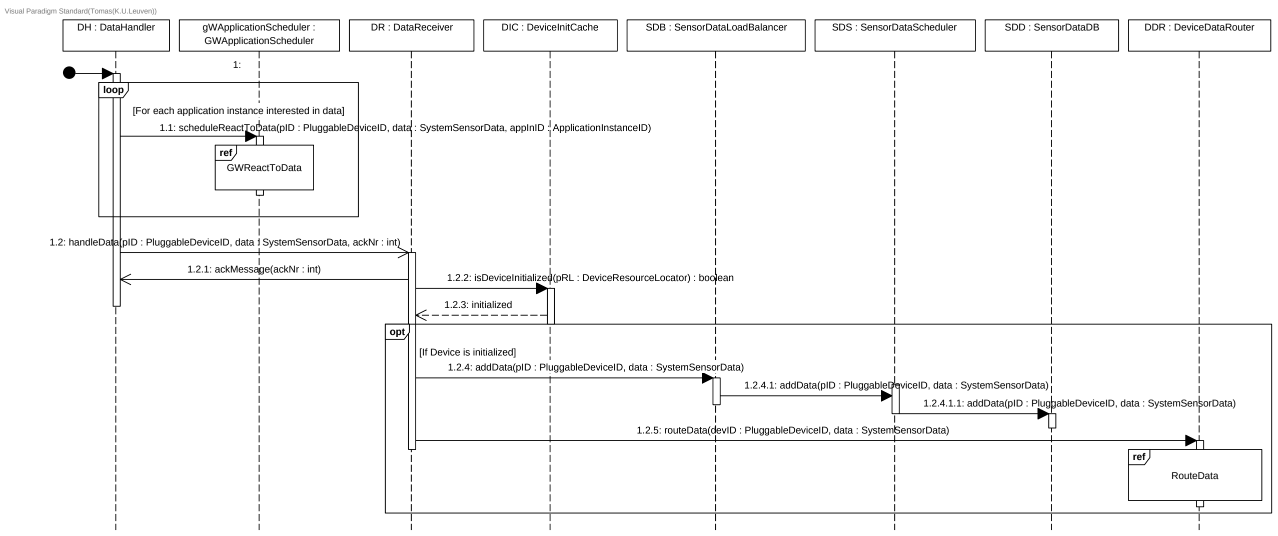

Visual Paradigm Standard(Tomas(K.U.Leuven))
DH : DataHandler
gWApplicationScheduler : GWApplicationScheduler
DR : DataReceiver
DIC : DeviceInitCache
SDB : SensorDataLoadBalancer
SDS : SensorDataScheduler
SDD : SensorDataDB
DDR : DeviceDataRouter
1:
loop
[For each application instance interested in data]
1.1: scheduleReactToData(pID : PluggableDeviceID, data : SystemSensorData, appInID : ApplicationInstanceID)
ref
GWReactToData
1.2: handleData(pID : PluggableDeviceID, data : SystemSensorData, ackNr : int)
1.2.1: ackMessage(ackNr : int)
1.2.2: isDeviceInitialized(pRL : DeviceResourceLocator) : boolean
1.2.3: initialized
opt
[If Device is initialized]
1.2.4: addData(pID : PluggableDeviceID, data : SystemSensorData)
1.2.4.1: addData(pID : PluggableDeviceID, data : SystemSensorData)
1.2.4.1.1: addData(pID : PluggableDeviceID, data : SystemSensorData)
1.2.5: routeData(devID : PluggableDeviceID, data : SystemSensorData)
ref
RouteData

Figure D.11: How a single pluggable device reading is handled. Checking whether a device is initialized is only needed at the Online Service as no application on the `Gateway` (or the Online Service) will be scheduled to receive data from an uninitialized device (cannot be part of a **TopologyRequirementsInstantiation**).

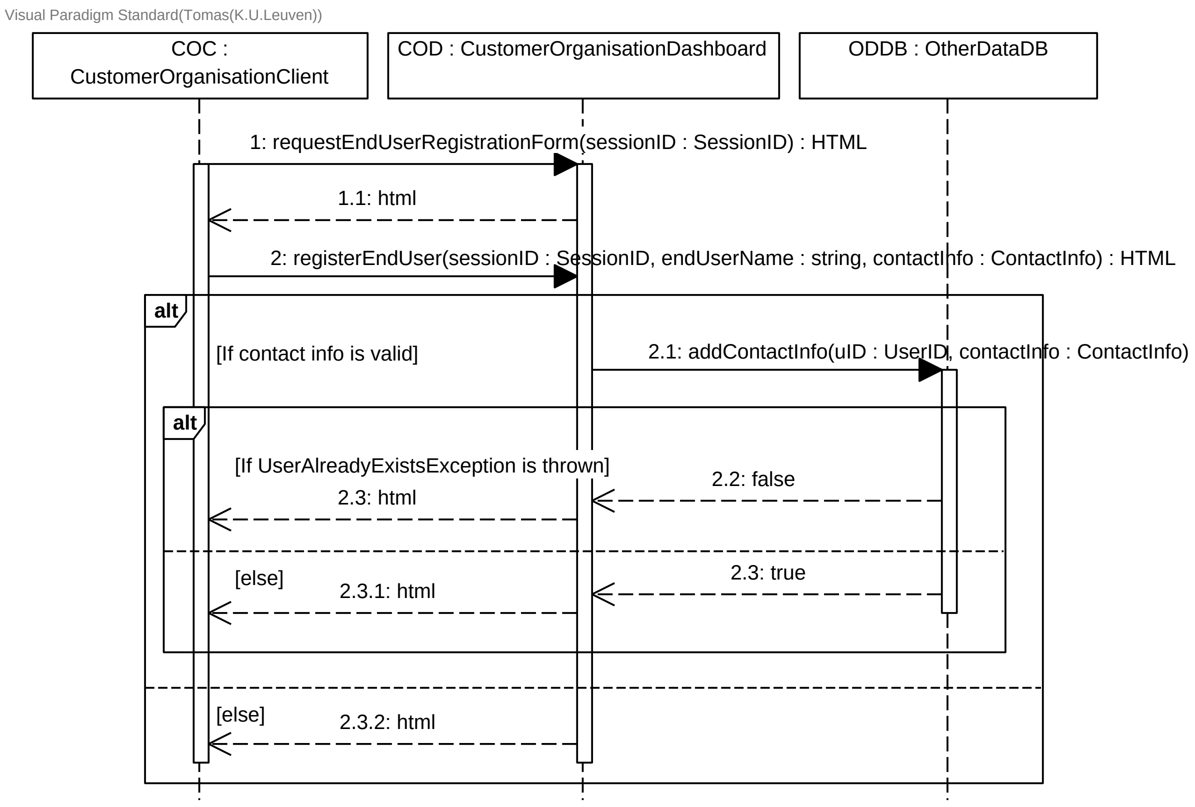

Visual Paradigm Standard(Tomas(K.U.Leuven))
COC : CustomerOrganisationClient
COD : CustomerOrganisationDashboard
ODDB : OtherDataDB
1: requestEndUserRegistrationForm(sessionID : SessionID) : HTML
1.1: html
2: registerEndUser(sessionID : SessionID, endUserName : string, contactInfo : ContactInfo) : HTML
alt
[If contact info is valid]
2.1: addContactInfo(uID : UserID, contactInfo : ContactInfo)
alt
[If UserAlreadyExistsException is thrown]
2.2: false
2.3: html
[else]
2.3: true
2.3.1: html
[else]
2.3.2: html

Figure D.12: Register a possible end user for a Customer Organisation.

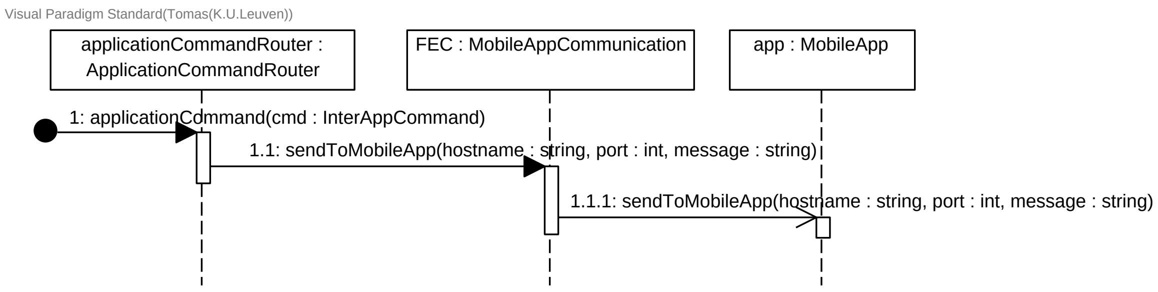

Visual Paradigm Standard(Tomas(K.U.Leuven))
applicationCommandRouter : ApplicationCommandRouter
FEC : MobileAppCommunication
app : MobileApp
1: applicationCommand(cmd : InterAppCommand)
1.1: sendToMobileApp(hostname : string, port : int, message : string)
1.1.1: sendToMobileApp(hostname : string, port : int, message : string)

Figure D.13: Send a command from an application instance on the Online Service to a smartphone app.

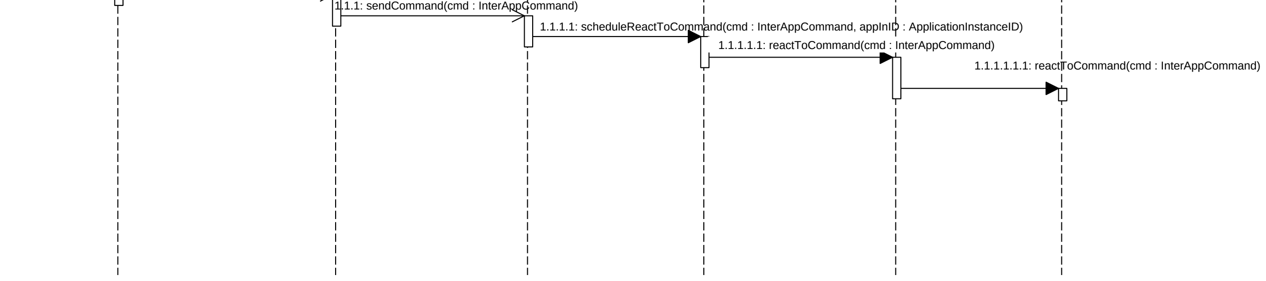

Visual Paradigm Standard(Tomas(K.U.Leuven))
applicationCommandRouter : ApplicationCommandRouter
applicationCommandPropagator : ApplicationCommandPropagator
gWCommandRouter : GWCommandRouter
gWApplicationScheduler : GWApplicationScheduler
gWApplicationContainer : GWApplicationContainer
gWApplication : GWApplication
1: applicationCommand(cmd : InterAppCommand)
1.1: sendCommandToGW(cmd : InterAppCommand)
1.1.1: sendCommand(cmd : InterAppCommand)
1.1.1.1: scheduleReactToCommand(cmd : InterAppCommand, appInID : ApplicationInstanceID)
1.1.1.1.1: reactToCommand(cmd : InterAppCommand)
1.1.1.1.1.1: reactToCommand(cmd : InterAppCommand)

Figure D.14: A command intended for the `Gateway` part of an application instance has arrived on the `ApplicationCommandRouter`, which forwards it to the correct `Gateway`.

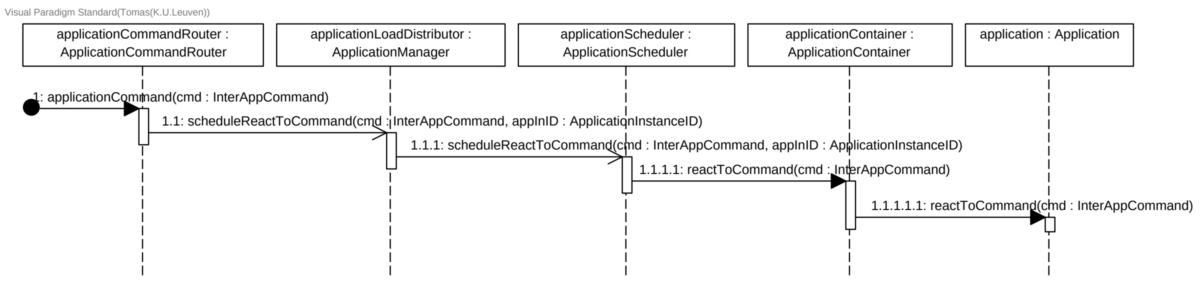

Visual Paradigm Standard(Tomas(K.U.Leuven))
applicationCommandRouter : ApplicationCommandRouter
applicationLoadDistributor : ApplicationManager
applicationScheduler : ApplicationScheduler
applicationContainer : ApplicationContainer
application : Application
1: applicationCommand(cmd : InterAppCommand)
1.1: scheduleReactToCommand(cmd : InterAppCommand, appInID : ApplicationInstanceID)
1.1.1: scheduleReactToCommand(cmd : InterAppCommand, appInID : ApplicationInstanceID)
1.1.1.1: reactToCommand(cmd : InterAppCommand)
1.1.1.1.1: reactToCommand(cmd : InterAppCommand)

Figure D.15: A command intended for an Online Service part of an `Application` instance has arrived on the `ApplicationCommandRouter` and is forwarded to the correct `Application` instance (via its container).

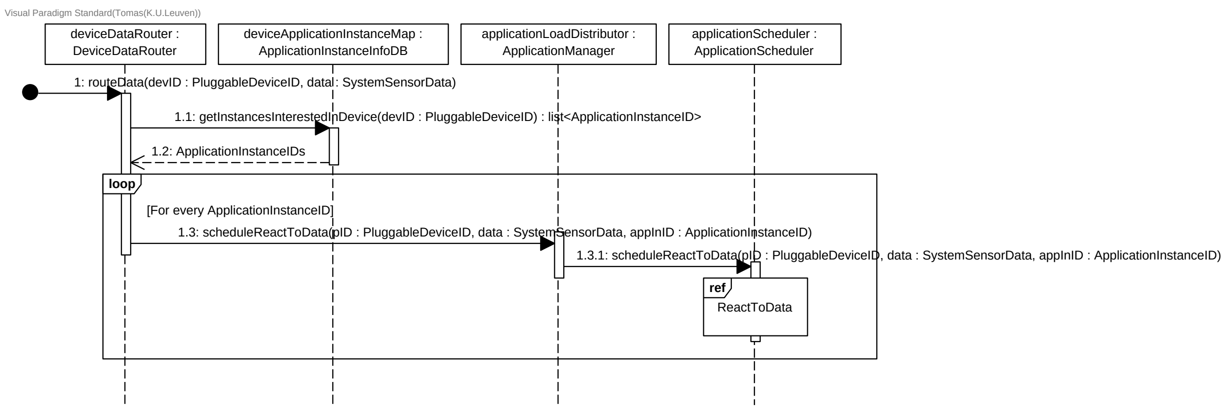

Visual Paradigm Standard(Tomas(K.U.Leuven))
deviceDataRouter : DeviceDataRouter
deviceApplicationInstanceMap : ApplicationInstanceInfoDB
applicationLoadDistributor : ApplicationManager
applicationScheduler : ApplicationScheduler
1: routeData(devID : PluggableDeviceID, data : SystemSensorData)
1.1: getInstancesInterestedInDevice(devID : PluggableDeviceID) : list<ApplicationInstanceID>
1.2: ApplicationInstanceIDs
loop
[For every ApplicationInstanceID]
1.3: scheduleReactToData(pID : PluggableDeviceID, data : SystemSensorData, appInID : ApplicationInstanceID)
1.3.1: scheduleReactToData(pID : PluggableDeviceID, data : SystemSensorData, appInID : ApplicationInstanceID)
ref
ReactToData

Figure D.16: RouteData

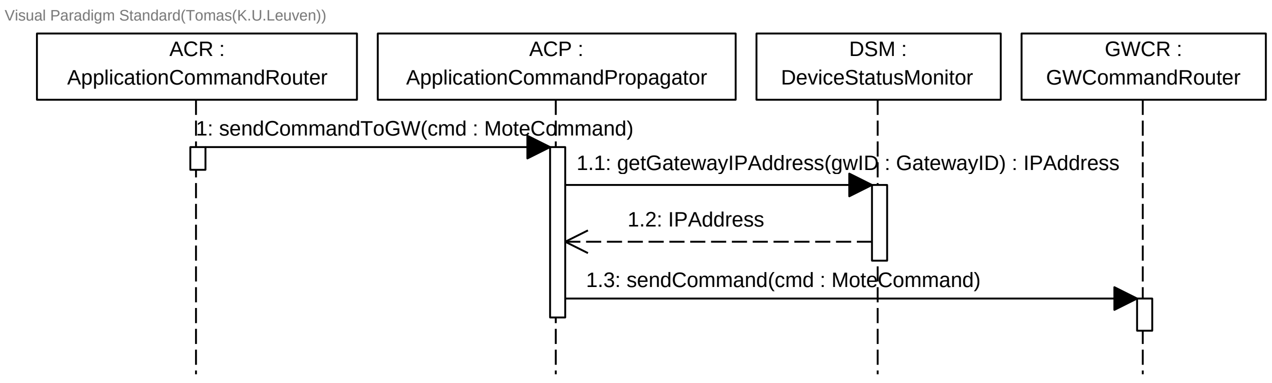

Visual Paradigm Standard(Tomas(K.U.Leuven))
ACR : ApplicationCommandRouter
ACP : ApplicationCommandPropagator
DSM : DeviceStatusMonitor
GWCR : GWCommandRouter
1: sendCommandToGW(cmd : MoteCommand)
1.1: getGatewayIPAddress(gwID : GatewayID) : IPAddress
1.2: IPAddress
1.3: sendCommand(cmd : MoteCommand)

Figure D.17: Forward a command to a `Gateway`, this can be an actuation command or a command for an `Gateway Application` instance.

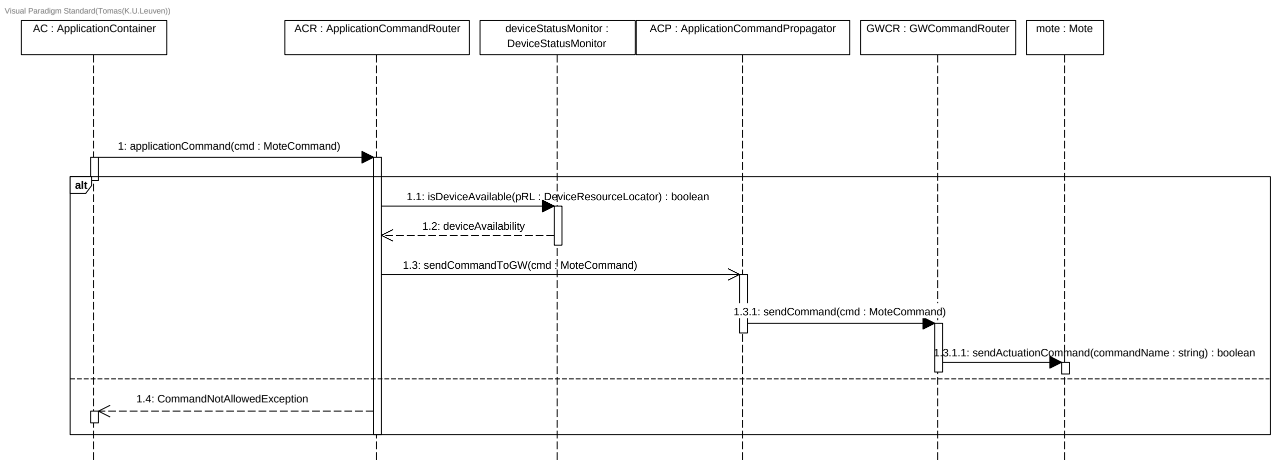

Visual Paradigm Standard(Tomas(K.U.Leuven))
AC : ApplicationContainer
ACR : ApplicationCommandRouter
deviceStatusMonitor : DeviceStatusMonitor
ACP : ApplicationCommandPropagator
GWCR : GWCommandRouter
mote : Mote
1: applicationCommand(cmd : MoteCommand)
alt
1.1: isDeviceAvailable(pRL : DeviceResourceLocator) : boolean
1.2: deviceAvailability
1.3: sendCommandToGW(cmd : MoteCommand)
1.3.1: sendCommand(cmd : MoteCommand)
1.3.1.1: sendActuationCommand(commandName : string) : boolean
1.4: CommandNotAllowedException

Figure D.18: Routing of an actuation command to `Mote`.

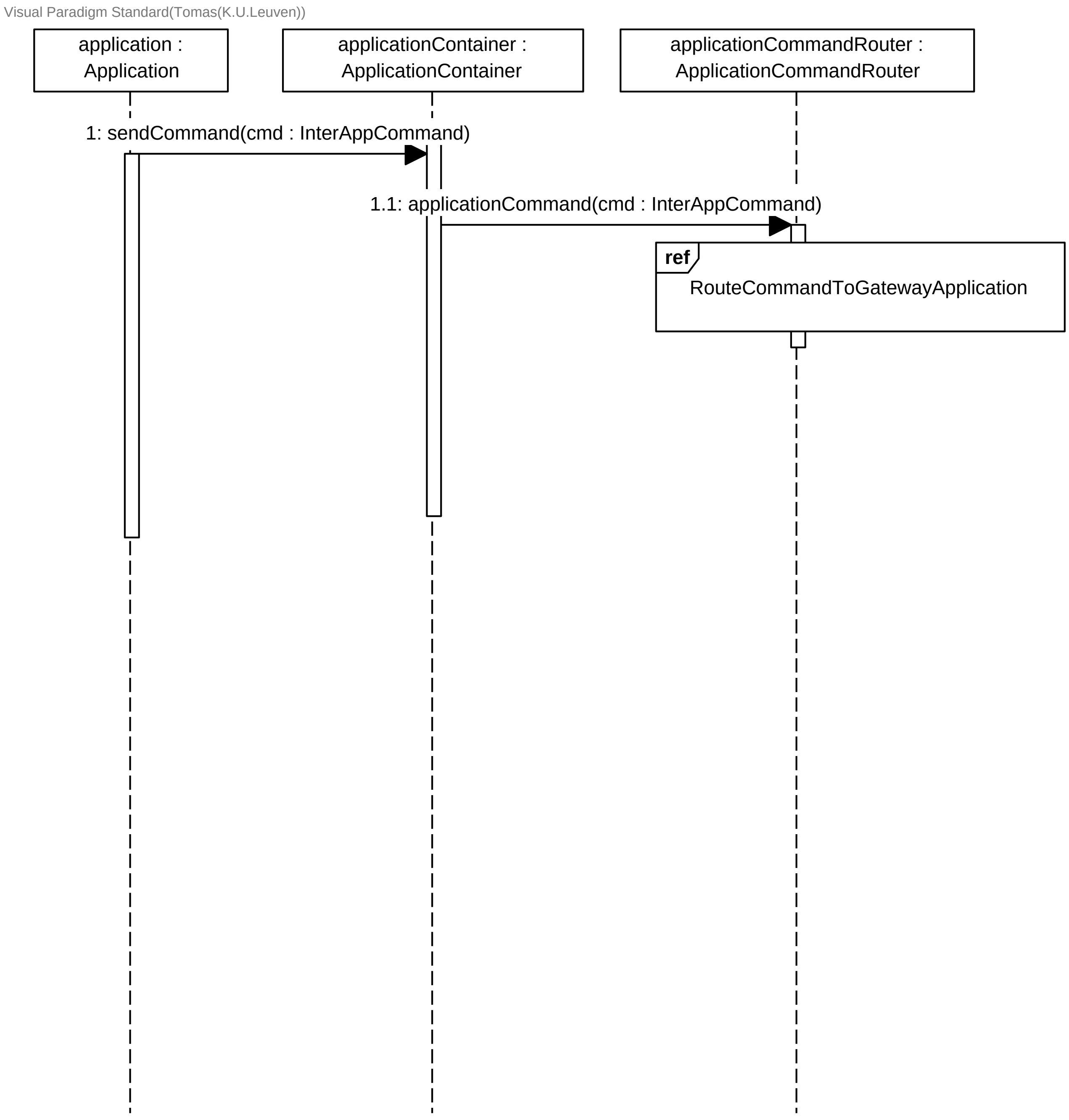

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
applicationContainer : ApplicationContainer
applicationCommandRouter : ApplicationCommandRouter
1: sendCommand(cmd : InterAppCommand)
1.1: applicationCommand(cmd : InterAppCommand)
ref
RouteCommandToGatewayApplication

Figure D.19: Forward commands received from an application instance on the Online Service to `Gateway`.

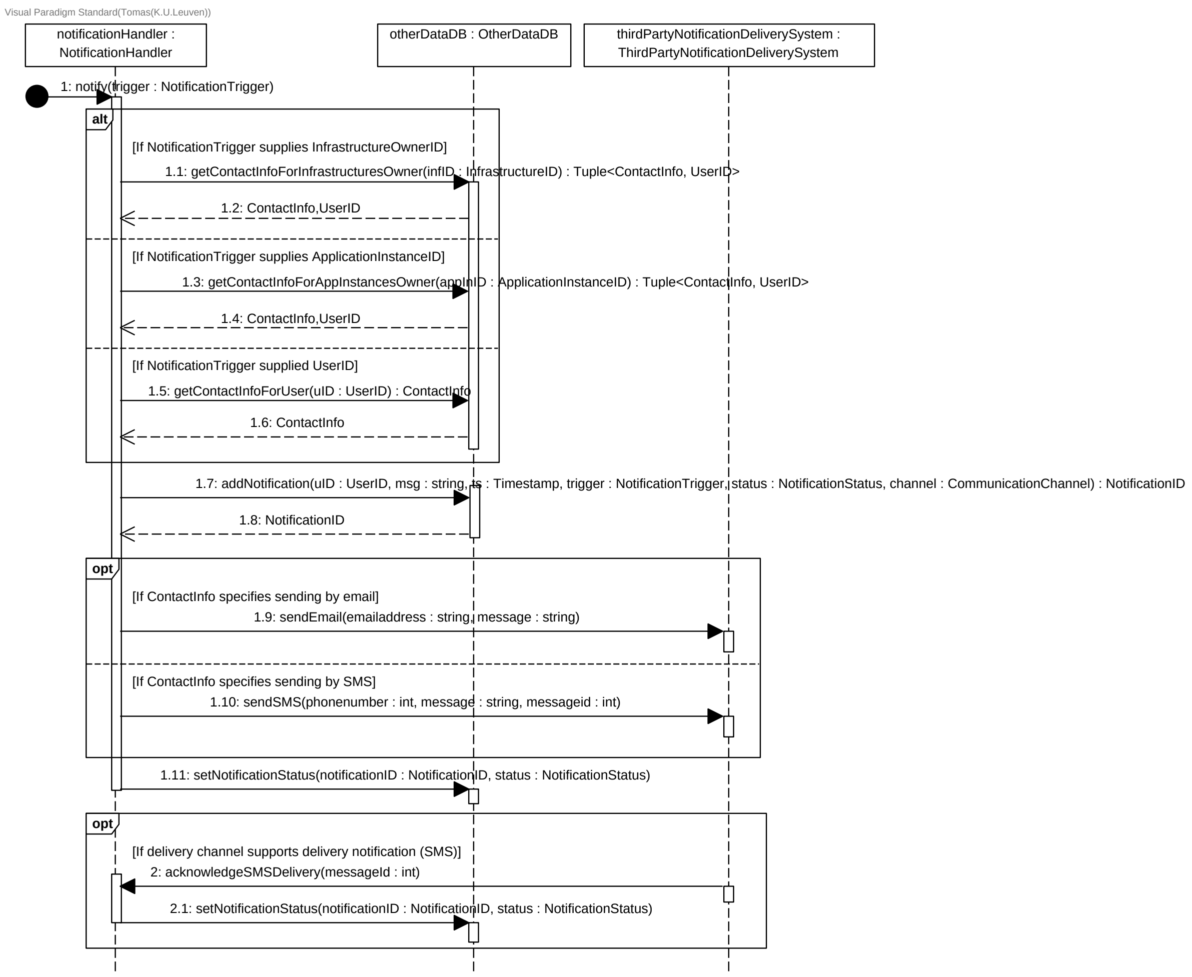

Visual Paradigm Standard(Tomas(K.U.Leuven))
notificationHandler : NotificationHandler
otherDataDB : OtherDataDB
thirdPartyNotificationDeliverySystem : ThirdPartyNotificationDeliverySystem
1: notify(trigger : NotificationTrigger)
alt
[If NotificationTrigger supplies InfrastructureOwnerID]
1.1: getContactInfoForInfrastructuresOwner(infID : InfrastructureID) : Tuple<ContactInfo, UserID>
1.2: ContactInfo,UserID
[If NotificationTrigger supplies ApplicationInstanceID]
1.3: getContactInfoForAppInstancesOwner(appInID : ApplicationInstanceID) : Tuple<ContactInfo, UserID>
1.4: ContactInfo,UserID
[If NotificationTrigger supplied UserID]
1.5: getContactInfoForUser(uID : UserID) : ContactInfo
1.6: ContactInfo
1.7: addNotification(uID : UserID, msg : string, ts : Timestamp, trigger : NotificationTrigger, status : NotificationStatus, channel : CommunicationChannel) : NotificationID
1.8: NotificationID
opt
[If ContactInfo specifies sending by email]
1.9: sendEmail(emailaddress : string, message : string)
[If ContactInfo specifies sending by SMS]
1.10: sendSMS(phonenumber : int, message : string, messageid : int)
1.11: setNotificationStatus(notificationID : NotificationID, status : NotificationStatus)
opt
[If delivery channel supports delivery notification (SMS)]
2: acknowledgeSMSDelivery(messageId : int)
2.1: setNotificationStatus(notificationID : NotificationID, status : NotificationStatus)

Figure D.20: Sending a notification because one of the many possible triggers happened. Note that a **NotificationTrigger** can contain a variety of data causing different lookups to the `OtherDataDB`. The first setNotification call will mark a notification as “sent”, the optional second one as “delivered”.

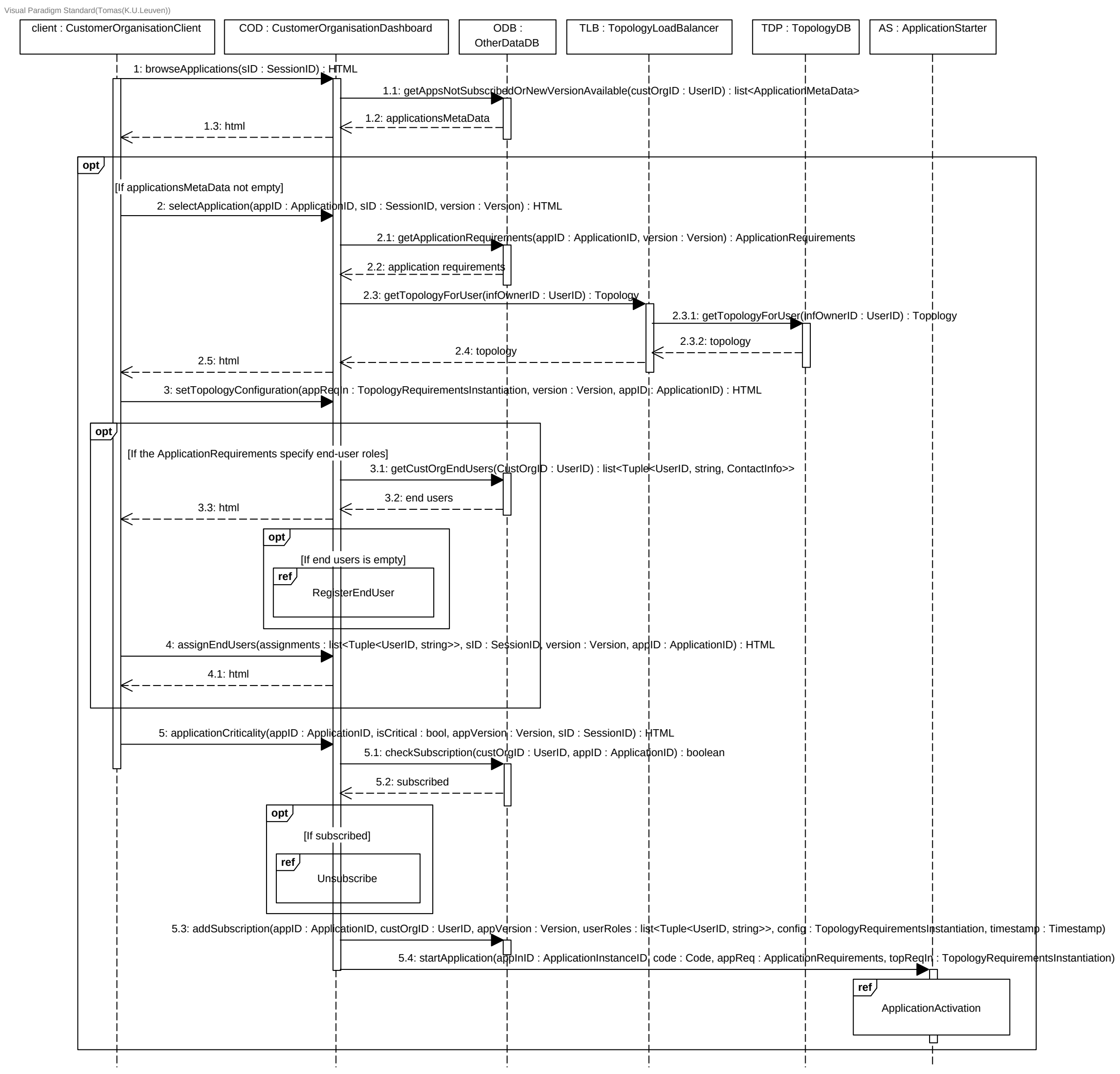

Visual Paradigm Standard(Tomas(K.U.Leuven))
client : CustomerOrganisationClient
COD : CustomerOrganisationDashboard
ODB : OtherDataDB
TLB : TopologyLoadBalancer
TDP : TopologyDB
AS : ApplicationStarter
1: browseApplications(sID : SessionID) : HTML
1.1: getAppsNotSubscribedOrNewVersionAvailable(custOrgID : UserID) : list<ApplicationMetaData>
1.2: applicationsMetaData
1.3: html
opt
[If applicationsMetaData not empty]
2: selectApplication(appID : ApplicationID, sID : SessionID, version : Version) : HTML
2.1: getApplicationRequirements(appID : ApplicationID, version : Version) : ApplicationRequirements
2.2: application requirements
2.3: getTopologyForUser(infOwnerID : UserID) : Topology
2.3.1: getTopologyForUser(infOwnerID : UserID) : Topology
2.3.2: topology
2.4: topology
2.5: html
3: setTopologyConfiguration(appReqIn : TopologyRequirementsInstantiation, version : Version, appID : ApplicationID) : HTML
opt
[If the ApplicationRequirements specify end-user roles]
3.1: getCustOrgEndUsers(CustOrgID : UserID) : list<Tuple<UserID, string, ContactInfo>>
3.2: end users
3.3: html
opt
[If end users is empty]
ref
RegisterEndUser
4: assignEndUsers(assignments : list<Tuple<UserID, string>>, sID : SessionID, version : Version, appID : ApplicationID) : HTML
4.1: html
5: applicationCriticality(appID : ApplicationID, isCritical : bool, appVersion : Version, sID : SessionID) : HTML
5.1: checkSubscription(custOrgID : UserID, appID : ApplicationID) : boolean
5.2: subscribed
opt
[If subscribed]
ref
Unsubscribe
5.3: addSubscription(appID : ApplicationID, custOrgID : UserID, appVersion : Version, userRoles : list<Tuple<UserID, string>>, config : TopologyRequirementsInstantiation, timestamp : Timestamp)
5.4: startApplication(appInID : ApplicationInstanceID, code : Code, appReq : ApplicationRequirements, topReqIn : TopologyRequirementsInstantiation)
ref
ApplicationActivation

Figure D.21: A Customer Organisation subscribes to an application.

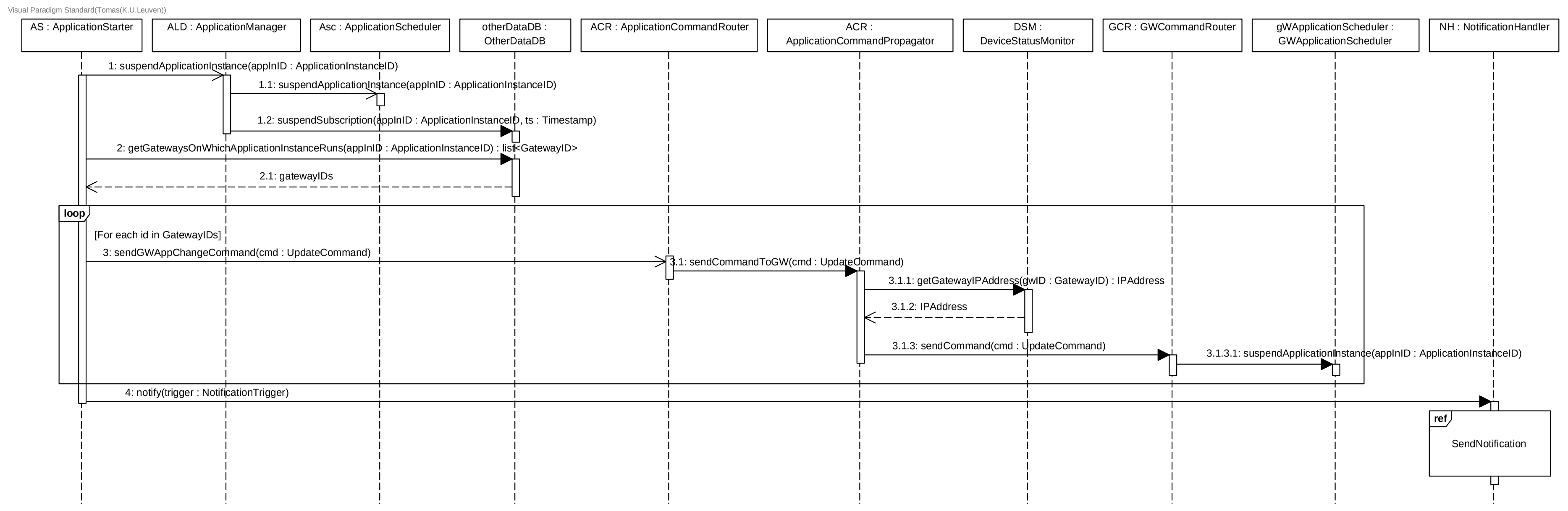

Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationStarter
ALD : ApplicationManager
Asc : ApplicationScheduler
otherDataDB : OtherDataDB
ACR : ApplicationCommandRouter
ACR : ApplicationCommandPropagator
DSM : DeviceStatusMonitor
GCR : GWCommandRouter
gWApplicationScheduler : GWApplicationScheduler
NH : NotificationHandler
1: suspendApplicationInstance(appInID : ApplicationInstanceID)
1.1: suspendApplicationInstance(appInID : ApplicationInstanceID)
1.2: suspendSubscription(appInID : ApplicationInstanceID, ts : Timestamp)
2: getGatewaysOnWhichApplicationInstanceRuns(appInID : ApplicationInstanceID) : list<GatewayID>
2.1: gatewayIDs
loop
[For each id in GatewayIDs]
3: sendGWAppChangeCommand(cmd : UpdateCommand)
3.1: sendCommandToGW(cmd : UpdateCommand)
3.1.1: getGatewayIPAddress(gwID : GatewayID) : IPAddress
3.1.2: IPAddress
3.1.3: sendCommand(cmd : UpdateCommand)
3.1.3.1: suspendApplicationInstance(appInID : ApplicationInstanceID)
4: notify(trigger : NotificationTrigger)
ref
SendNotification

Figure D.22: The sent "**Command**" in this case specifies the crashed application instance.

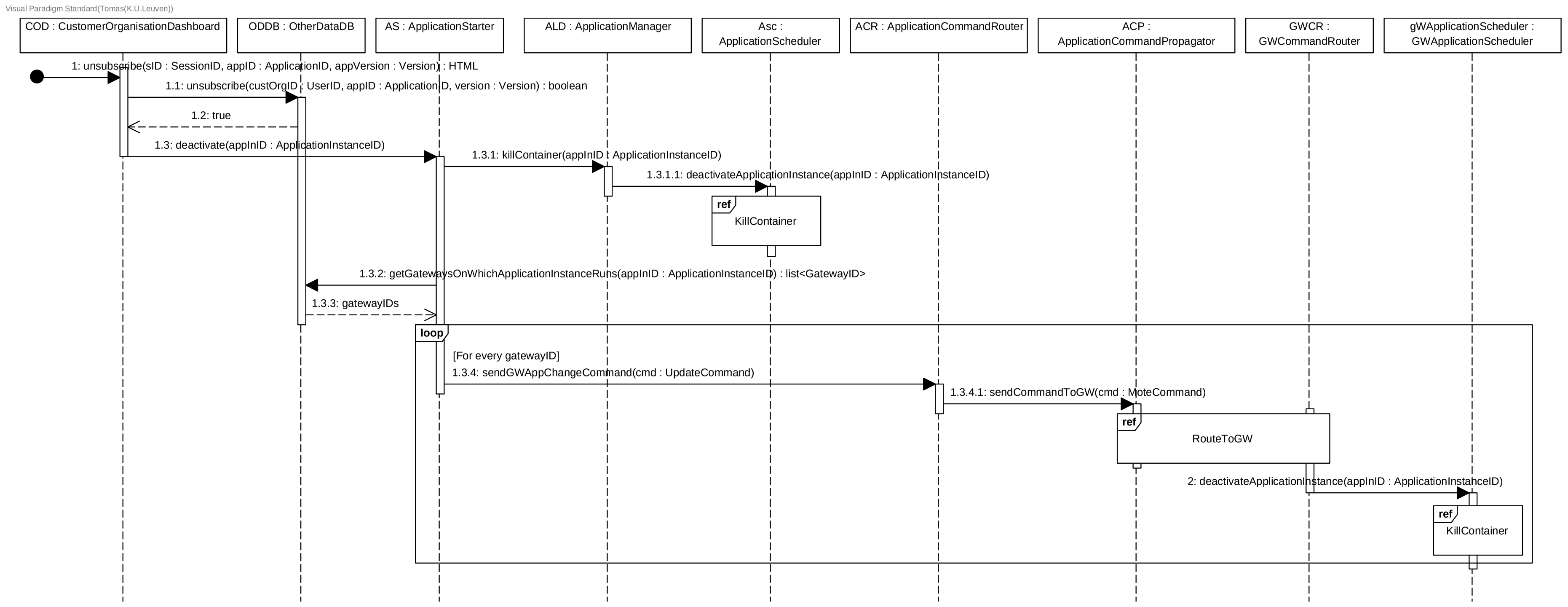
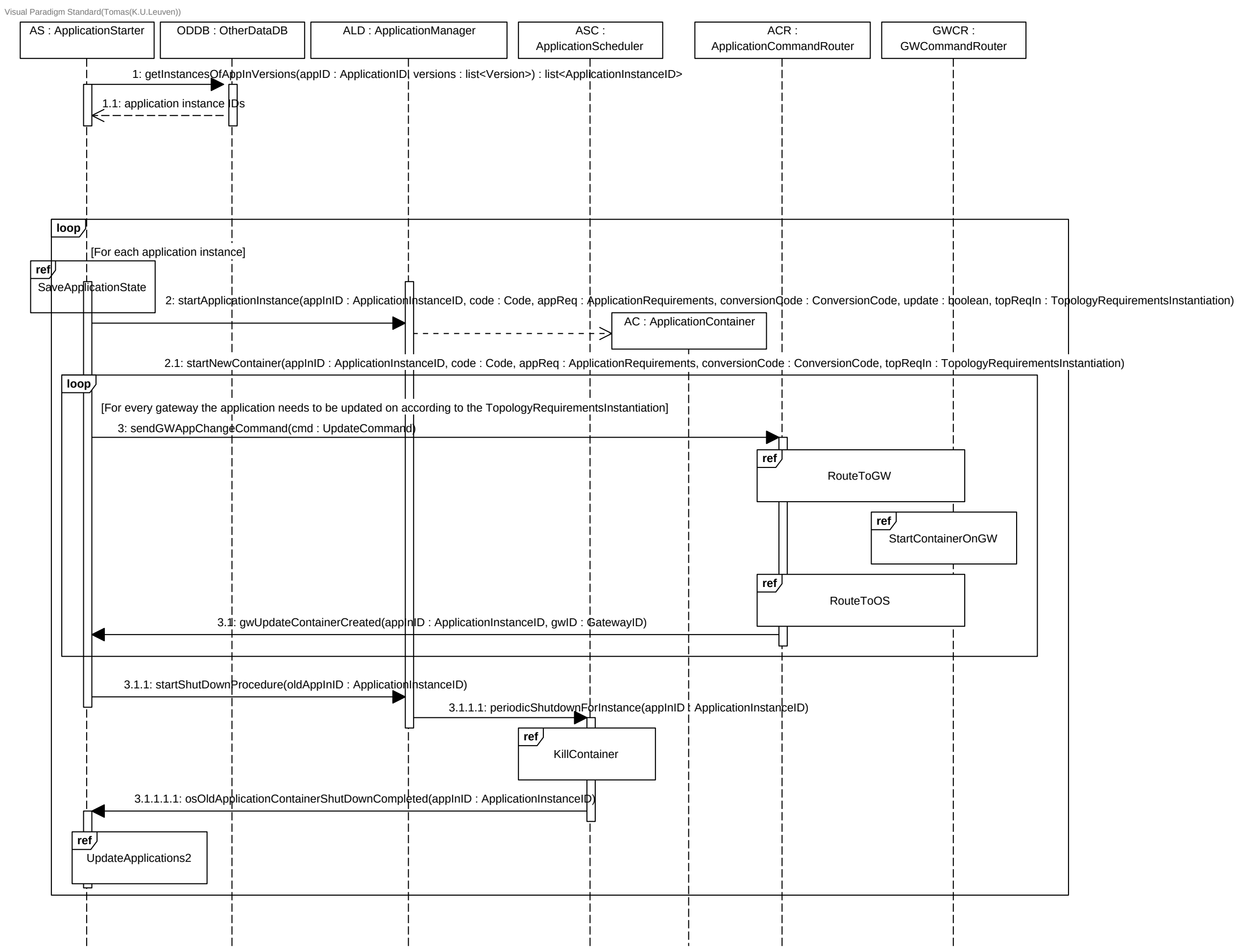

Visual Paradigm Standard(Tomas(K.U.Leuven))
COD : CustomerOrganisationDashboard
ODDB : OtherDataDB
AS : ApplicationStarter
ALD : ApplicationManager
Asc : ApplicationScheduler
ACR : ApplicationCommandRouter
ACP : ApplicationCommandPropagator
GWCR : GWCommandRouter
gWApplicationScheduler : GWApplicationScheduler
1: unsubscribe(sID : SessionID, appID : ApplicationID, appVersion : Version) : HTML
1.1: unsubscribe(custOrgID : UserID, appID : ApplicationID, version : Version) : boolean
1.2: true
1.3: deactivate(appInID : ApplicationInstanceID)
1.3.1: killContainer(appInID : ApplicationInstanceID)
1.3.1.1: deactivateApplicationInstance(appInID : ApplicationInstanceID)
ref
KillContainer
1.3.2: getGatewaysOnWhichApplicationInstanceRuns(appInID : ApplicationInstanceID) : list<GatewayID>
1.3.3: gatewayIDs
loop
[For every gatewayID]
1.3.4: sendGWAppChangeCommand(cmd : UpdateCommand)
1.3.4.1: sendCommandToGW(cmd : MoteCommand)
ref
RouteToGW
2: deactivateApplicationInstance(appInID : ApplicationInstanceID)
ref
KillContainer

Figure D.23: A Customer Organisation unsubscribes from an application. The **Command** being sent to the `Gateway` asks the `Gateway` to kill the container running the specified application instance.



Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationStarter
ODDB : OtherDataDB
ALD : ApplicationManager
ASC : ApplicationScheduler
ACR : ApplicationCommandRouter
GWCR : GWCommandRouter
1: getInstancesOfAppInVersions(appID : ApplicationID, versions : list<Version>) : list<ApplicationInstanceID>
1.1: application instance IDs
loop
[For each application instance]
ref
SaveApplicationState
2: startApplicationInstance(appInID : ApplicationInstanceID, code : Code, appReq : ApplicationRequirements, conversionCode : ConversionCode, update : boolean, topReqIn : TopologyRequirementsInstantiation)
AC : ApplicationContainer
2.1: startNewContainer(appInID : ApplicationInstanceID, code : Code, appReq : ApplicationRequirements, conversionCode : ConversionCode, topReqIn : TopologyRequirementsInstantiation)
loop
[For every gateway the application needs to be updated on according to the TopologyRequirementsInstantiation]
3: sendGWAppChangeCommand(cmd : UpdateCommand)
ref
RouteToGW
ref
StartContainerOnGW
ref
RouteToOS
3.1: gwUpdateContainerCreated(appInID : ApplicationInstanceID, gwID : GatewayID)
3.1.1: startShutDownProcedure(oldAppInID : ApplicationInstanceID)
3.1.1.1: periodicShutdownForInstance(appInID : ApplicationInstanceID)
ref
KillContainer
3.1.1.1.1: osOldApplicationContainerShutDownCompleted(appInID : ApplicationInstanceID)
ref
UpdateApplications2

Figure D.24: An application is eligible for an update. The old version is safely shut down and the new one booted up. Starting a container on the `Gateway` is similar as on the Online Service and is not explicitly shown.

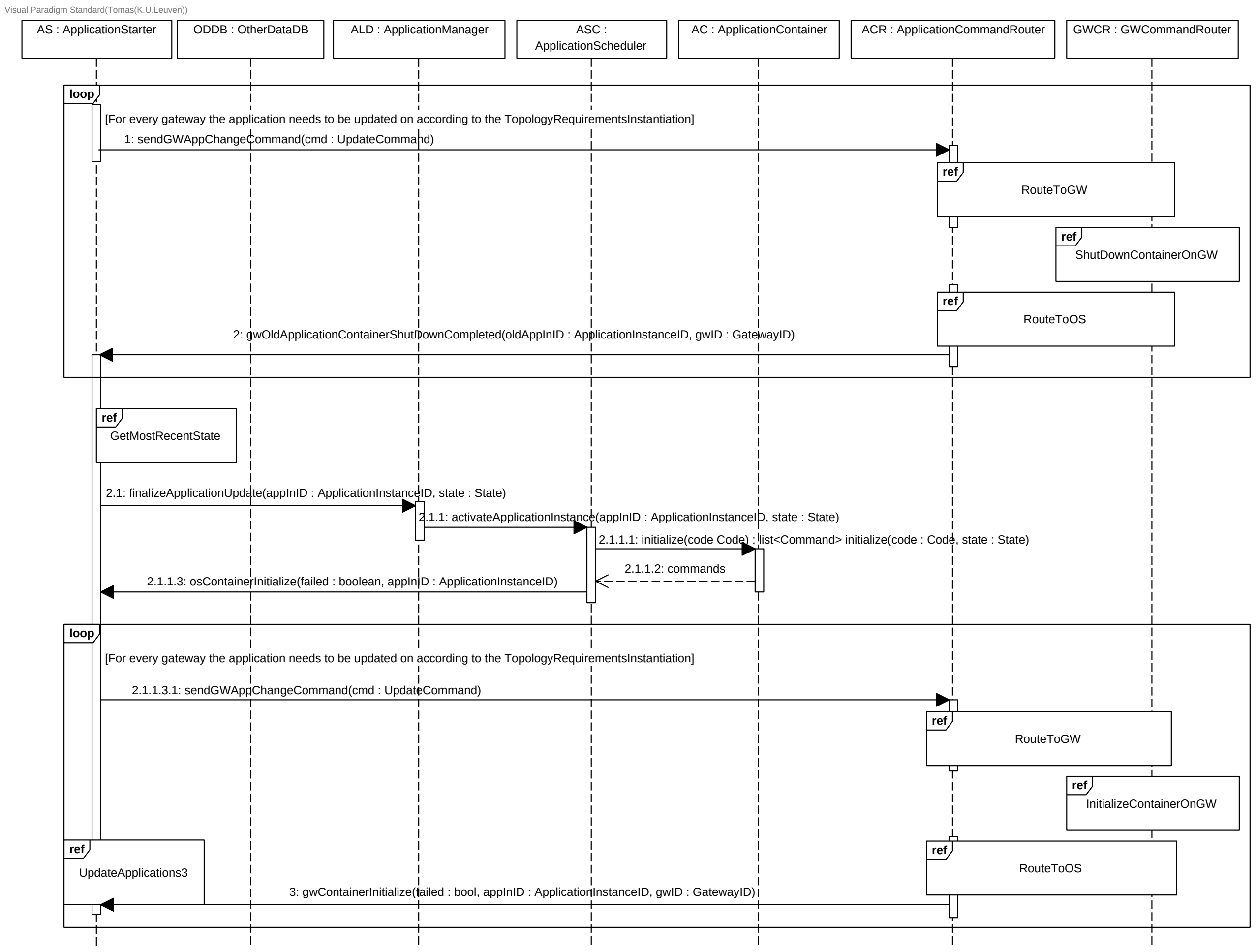
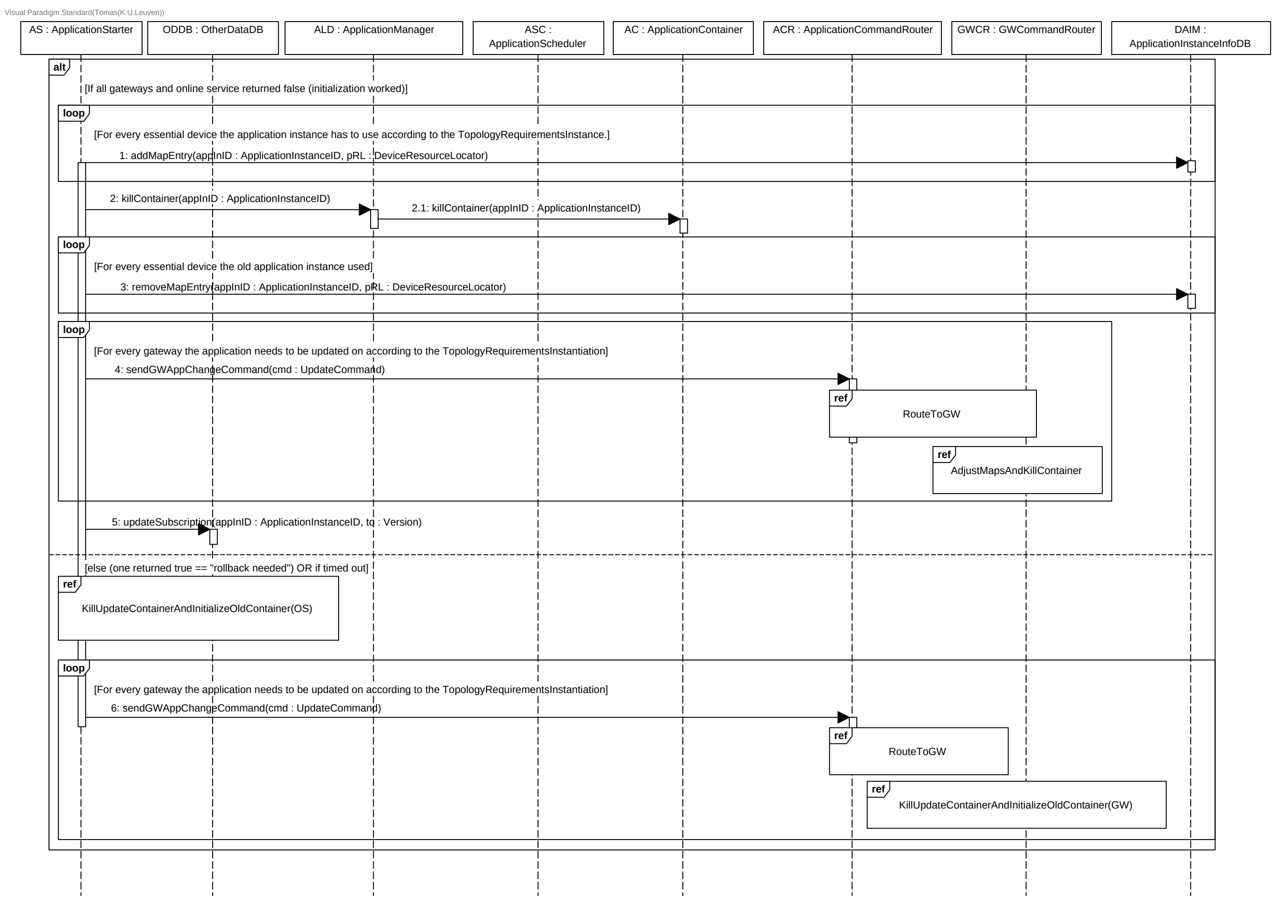
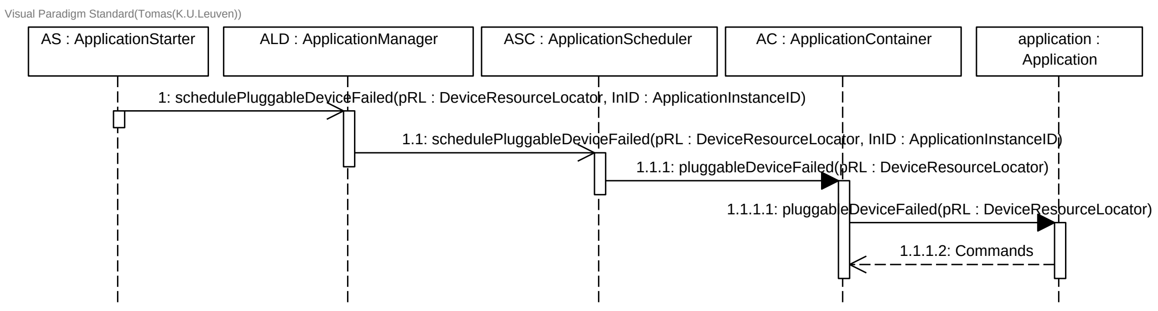

Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationStarter
ODDB : OtherDataDB
ALD : ApplicationManager
ASC : ApplicationScheduler
AC : ApplicationContainer
ACR : ApplicationCommandRouter
GWCR : GWCommandRouter
loop
[For every gateway the application needs to be updated on according to the TopologyRequirementsInstantiation]
1: sendGWAppChangeCommand(cmd : UpdateCommand)
ref
RouteToGW
ref
ShutDownContainerOnGW
ref
RouteToOS
2: gwOldApplicationContainerShutDownCompleted(oldAppInID : ApplicationInstanceID, gwID : GatewayID)
ref
GetMostRecentState
2.1: finalizeApplicationUpdate(appInID : ApplicationInstanceID, state : State)
2.1.1: activateApplicationInstance(appInID : ApplicationInstanceID, state : State)
2.1.1.1: initialize(code Code) : list<Command> initialize(code : Code, state : State)
2.1.1.2: commands
2.1.1.3: osContainerInitialize(failed : boolean, appInID : ApplicationInstanceID)
loop
[For every gateway the application needs to be updated on according to the TopologyRequirementsInstantiation]
2.1.1.3.1: sendGWAppChangeCommand(cmd : UpdateCommand)
ref
RouteToGW
ref
InitializeContainerOnGW
ref
UpdateApplications3
ref
RouteToOS
3: gwContainerInitialize(failed : bool, appInID : ApplicationInstanceID, gwID : GatewayID)

Figure D.25: UpdateApplications2



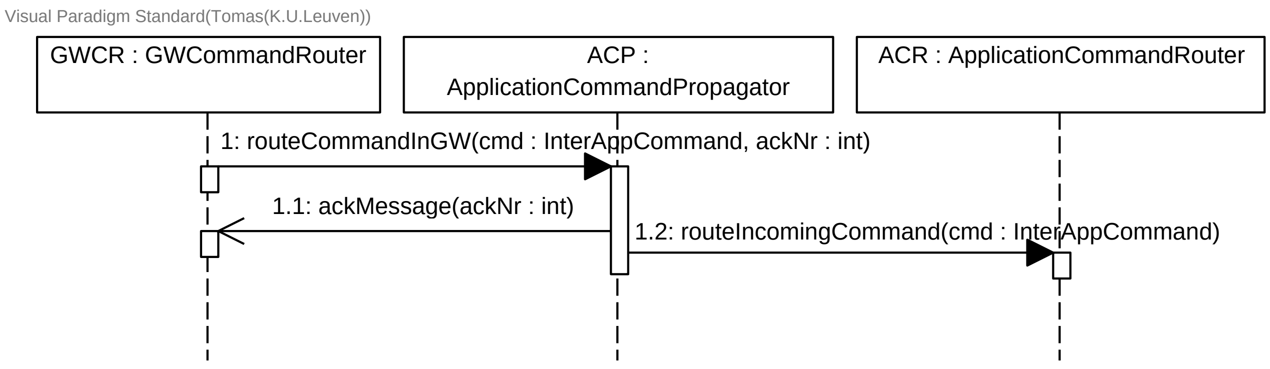
Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationStarter
ODDB : OtherDataDB
ALD : ApplicationManager
ASC : ApplicationScheduler
AC : ApplicationContainer
ACR : ApplicationCommandRouter
GWCR : GWCommandRouter
DAIM : ApplicationInstanceInfoDB
alt
[If all gateways and online service returned false (initialization worked)]
loop
[For every essential device the application instance has to use according to the TopologyRequirementsInstance.]
1: addMapEntry(appInID : ApplicationInstanceID, pRL : DeviceResourceLocator)
2: killContainer(appInID : ApplicationInstanceID)
2.1: killContainer(appInID : ApplicationInstanceID)
loop
[For every essential device the old application instance used]
3: removeMapEntry(appInID : ApplicationInstanceID, pRL : DeviceResourceLocator)
loop
[For every gateway the application needs to be updated on according to the TopologyRequirementsInstantiation]
4: sendGWAppChangeCommand(cmd : UpdateCommand)
ref
RouteToGW
ref
AdjustMapsAndKillContainer
5: updateSubscription(appInID : ApplicationInstanceID, to : Version)
[else (one returned true == "rollback needed") OR if timed out]
ref
KillUpdateContainerAndInitializeOldContainer(OS)
loop
[For every gateway the application needs to be updated on according to the TopologyRequirementsInstantiation]
6: sendGWAppChangeCommand(cmd : UpdateCommand)
ref
RouteToGW
ref
KillUpdateContainerAndInitializeOldContainer(GW)

Figure D.26: Add required pluggable devices to the new (updated) application instances, remove them from the previous, now outdated, instances. The containers executing the old application instances are killed.



Visual Paradigm Standard(Tomas(K.U.Leuven))
AS : ApplicationStarter
ALD : ApplicationManager
ASC : ApplicationScheduler
AC : ApplicationContainer
application : Application
1: schedulePluggableDeviceFailed(pRL : DeviceResourceLocator, InID : ApplicationInstanceID)
1.1: schedulePluggableDeviceFailed(pRL : DeviceResourceLocator, InID : ApplicationInstanceID)
1.1.1: pluggableDeviceFailed(pRL : DeviceResourceLocator)
1.1.1.1: pluggableDeviceFailed(pRL : DeviceResourceLocator)
1.1.1.2: Commands

Figure D.27: Inform an application that a pluggable device it used has failed.



Visual Paradigm Standard(Tomas(K.U.Leuven))
GWCR : GWCommandRouter
ACP : ApplicationCommandPropagator
ACR : ApplicationCommandRouter
1: routeCommandInGW(cmd : InterAppCommand, ackNr : int)
1.1: ackMessage(ackNr : int)
1.2: routeIncomingCommand(cmd : InterAppCommand)

Figure D.28: RouteToOS: Forward a command from the `Gateway` to the Online Service.

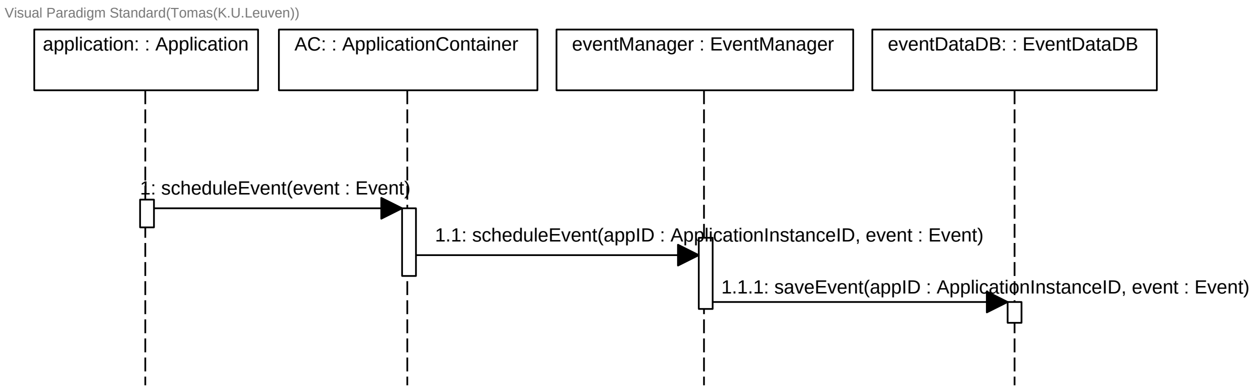

Visual Paradigm Standard(Tomas(K.U.Leuven))
application: : Application
AC: : ApplicationContainer
eventManager : EventManager
eventDataDB: : EventDataDB
1: scheduleEvent(event : Event)
1.1: scheduleEvent(appID : ApplicationInstanceID, event : Event)
1.1.1: saveEvent(appID : ApplicationInstanceID, event : Event)

Figure D.29: The application registers an event in `EventDataDB`.

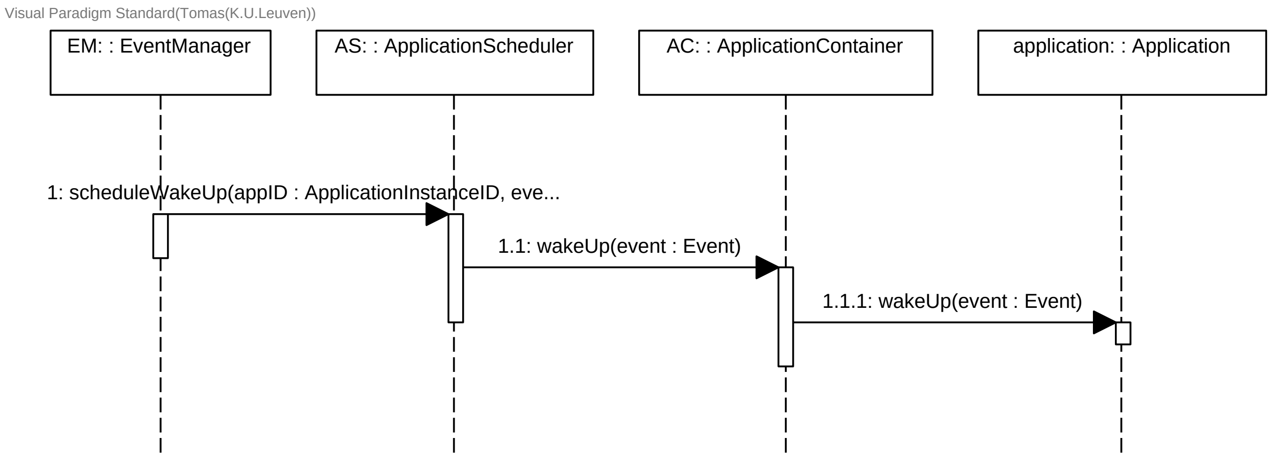

Visual Paradigm Standard(Tomas(K.U.Leuven))
EM: : EventManager
AS: : ApplicationScheduler
AC: : ApplicationContainer
application: : Application
1: scheduleWakeUp(appID : ApplicationInstanceID, eve...
1.1: wakeUp(event : Event)
1.1.1: wakeUp(event : Event)

Figure D.30: The `Application` is notified of a time-based event. Timed events on the `Gateway` happen similar

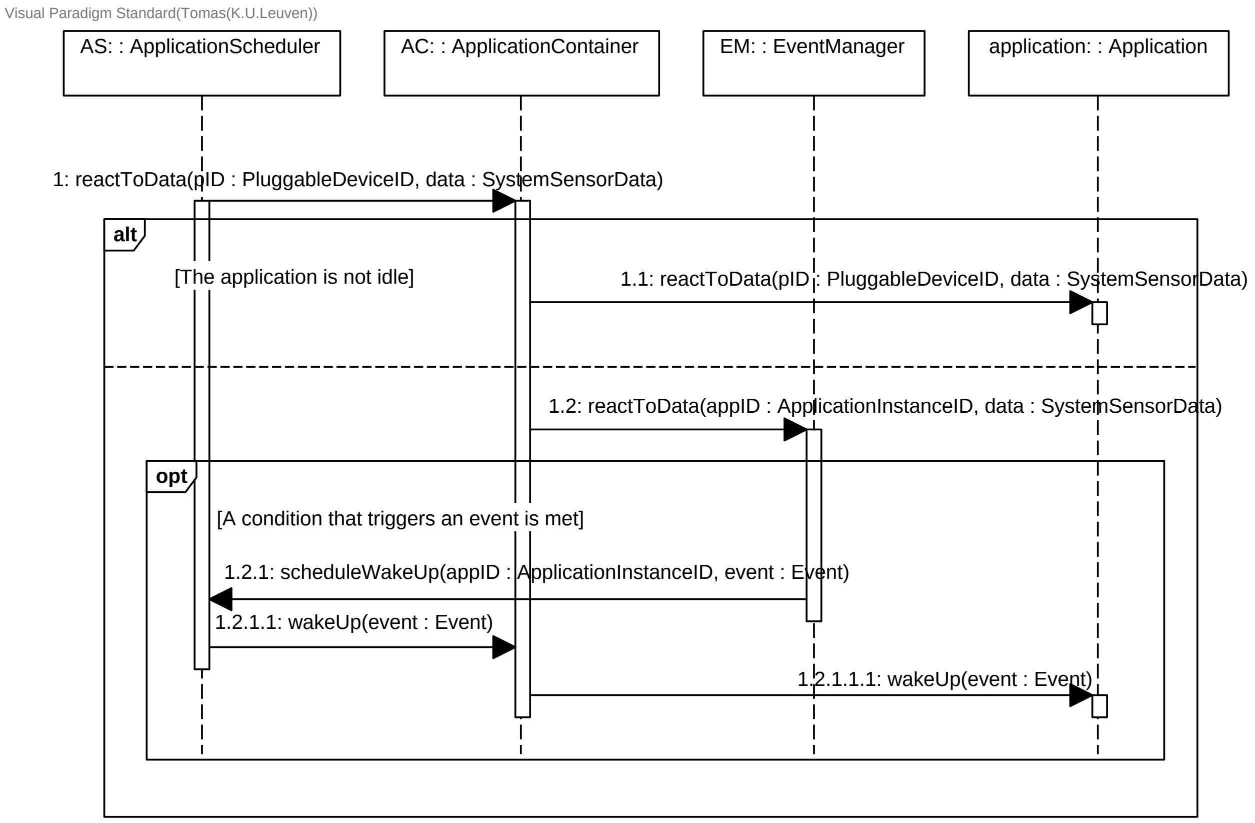

Visual Paradigm Standard(Tomas(K.U.Leuven))
AS: : ApplicationScheduler
AC: : ApplicationContainer
EM: : EventManager
application: : Application
1: reactToData(pID : PluggableDeviceID, data : SystemSensorData)
alt
[The application is not idle]
1.1: reactToData(pID : PluggableDeviceID, data : SystemSensorData)
1.2: reactToData(appID : ApplicationInstanceID, data : SystemSensorData)
opt
[A condition that triggers an event is met]
1.2.1: scheduleWakeUp(appID : ApplicationInstanceID, event : Event)
1.2.1.1: wakeUp(event : Event)
1.2.1.1.1: wakeUp(event : Event)

Figure D.31: When certain conditions, set by the application, are met, the application has to be woken up

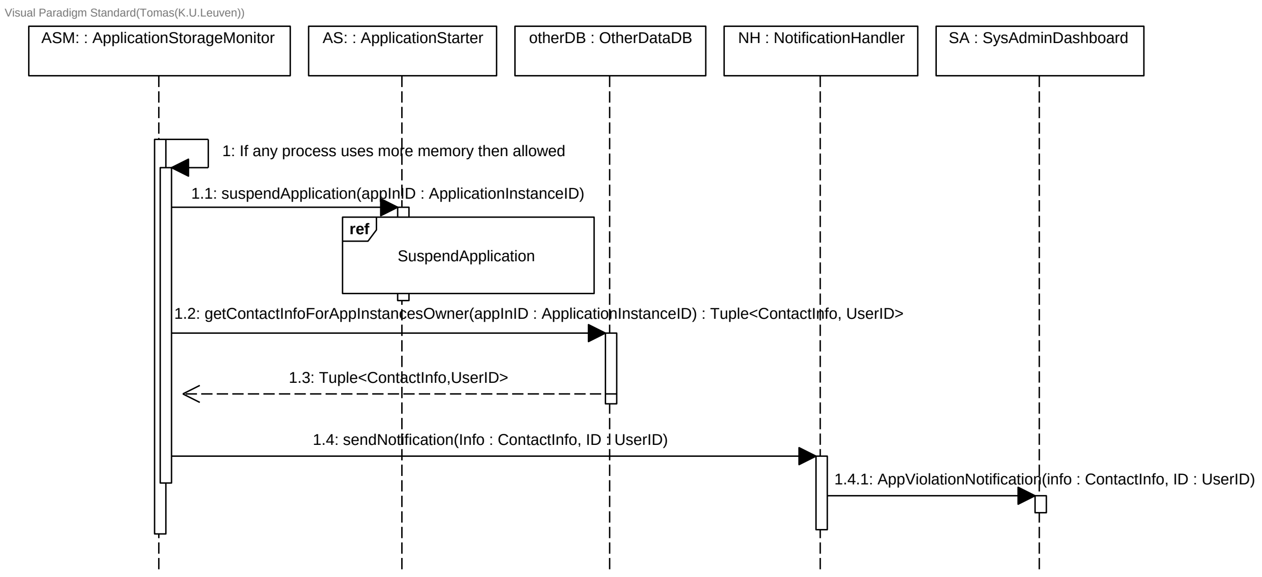

Visual Paradigm Standard(Tomas(K.U.Leuven))
ASM: : ApplicationStorageMonitor
AS: : ApplicationStarter
otherDB : OtherDataDB
NH : NotificationHandler
SA : SysAdminDashboard
1: If any process uses more memory then allowed
1.1: suspendApplication(appInID : ApplicationInstanceID)
ref
SuspendApplication
1.2: getContactInfoForAppInstancesOwner(appInID : ApplicationInstanceID) : Tuple<ContactInfo, UserID>
1.3: Tuple<ContactInfo,UserID>
1.4: sendNotification(Info : ContactInfo, ID : UserID)
1.4.1: AppViolationNotification(info : ContactInfo, ID : UserID)

Figure D.32: The application process will be suspended if it uses more memory then allocated by it's container. This limit is defined by the application developer when uploading the application. Sequence diagram for the gateway part works in same way and online service will be notified if it's counter part gets suspended.

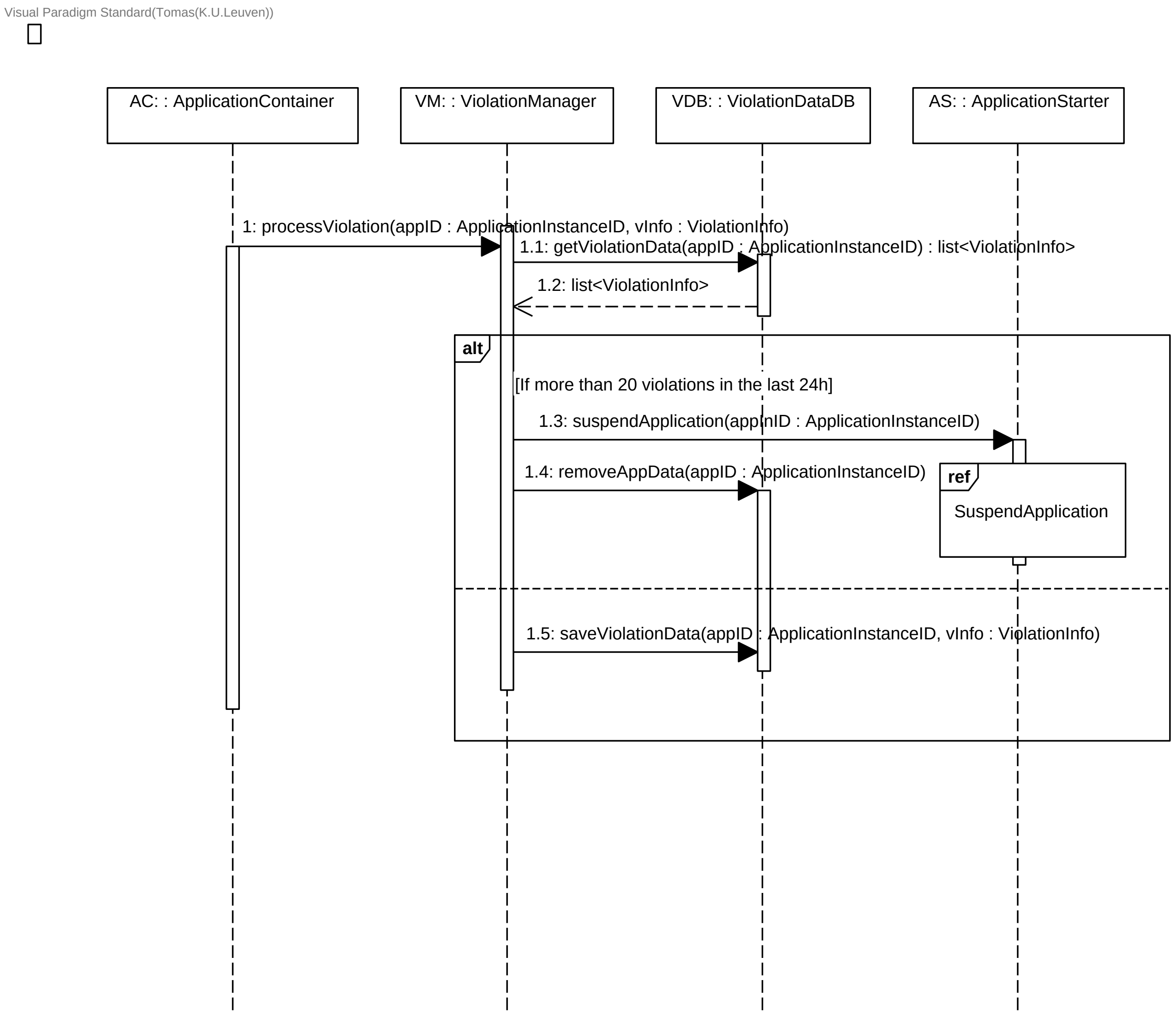

Visual Paradigm Standard(Tomas(K.U.Leuven))
AC: : ApplicationContainer
VM: : ViolationManager
VDB: : ViolationDataDB
AS: : ApplicationStarter
1: processViolation(appID : ApplicationInstanceID, vInfo : ViolationInfo)
1.1: getViolationData(appID : ApplicationInstanceID) : list<ViolationInfo>
1.2: list<ViolationInfo>
alt
[If more than 20 violations in the last 24h]
1.3: suspendApplication(appInID : ApplicationInstanceID)
1.4: removeAppData(appID : ApplicationInstanceID)
ref
SuspendApplication
1.5: saveViolationData(appID : ApplicationInstanceID, vInfo : ViolationInfo)

Figure D.33: The application will be suspended if it has more than 20 violations in database. If the violations are less than 20 then these will be updated in database. For each application Instance there is only one record in database and each record is saved along with the timestamp which is generated at the database.

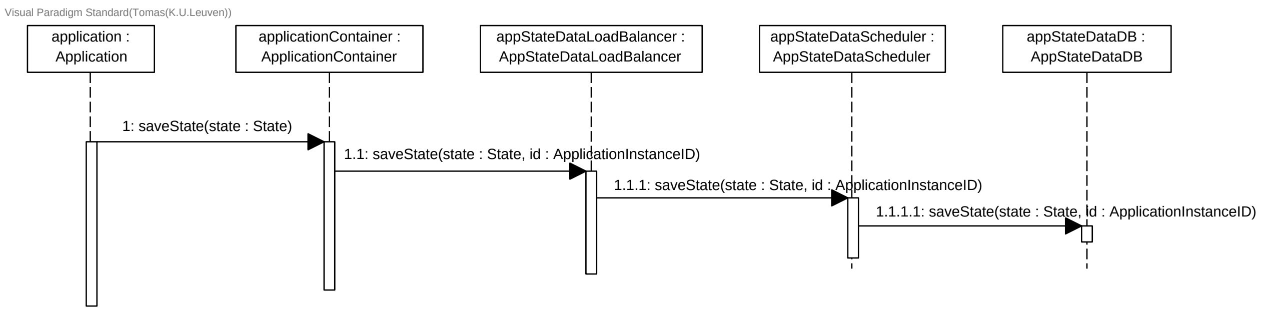

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
applicationContainer : ApplicationContainer
appStateDataLoadBalancer : AppStateDataLoadBalancer
appStateDataScheduler : AppStateDataScheduler
appStateDataDB : AppStateDataDB
1: saveState(state : State)
1.1: saveState(state : State, id : ApplicationInstanceID)
1.1.1: saveState(state : State, id : ApplicationInstanceID)
1.1.1.1: saveState(state : State, id : ApplicationInstanceID)

Figure D.34: SaveApplicationState

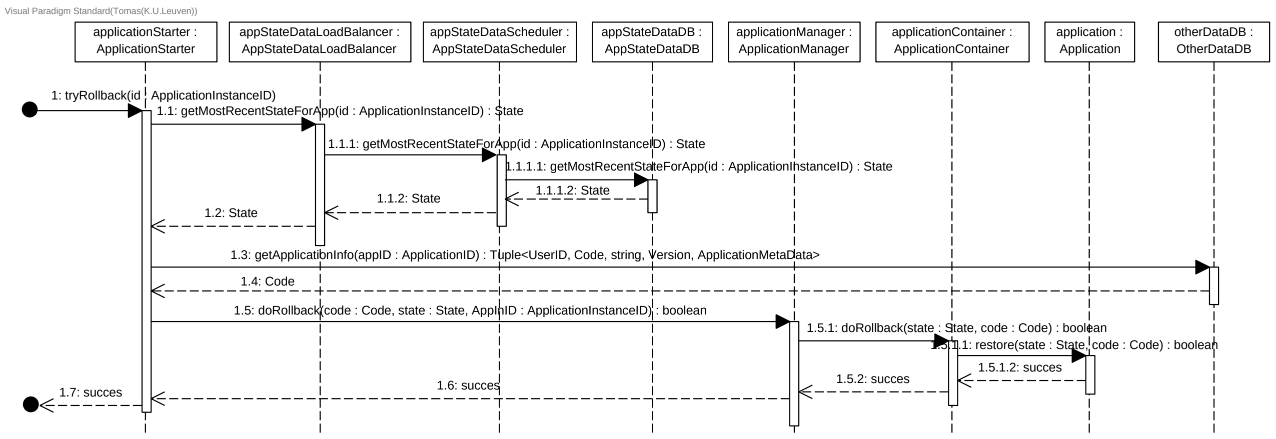

Visual Paradigm Standard(Tomas(K.U.Leuven))
applicationStarter : ApplicationStarter
appStateDataLoadBalancer : AppStateDataLoadBalancer
appStateDataScheduler : AppStateDataScheduler
appStateDataDB : AppStateDataDB
applicationManager : ApplicationManager
applicationContainer : ApplicationContainer
application : Application
otherDataDB : OtherDataDB
1: tryRollback(id : ApplicationInstanceID)
1.1: getMostRecentStateForApp(id : ApplicationInstanceID) : State
1.1.1: getMostRecentStateForApp(id : ApplicationInstanceID) : State
1.1.1.1: getMostRecentStateForApp(id : ApplicationInstanceID) : State
1.1.1.2: State
1.1.2: State
1.2: State
1.3: getApplicationInfo(appID : ApplicationID) : Tuple<UserID, Code, string, Version, ApplicationMetaData>
1.4: Code
1.5: doRollback(code : Code, state : State, ApplnID : ApplicationInstanceID) : boolean
1.5.1: doRollback(state : State, code : Code) : boolean
1.5.1.1: restore(state : State, code : Code) : boolean
1.5.1.2: succes
1.5.2: succes
1.6: succes
1.7: succes

Figure D.35: TryAppRollback

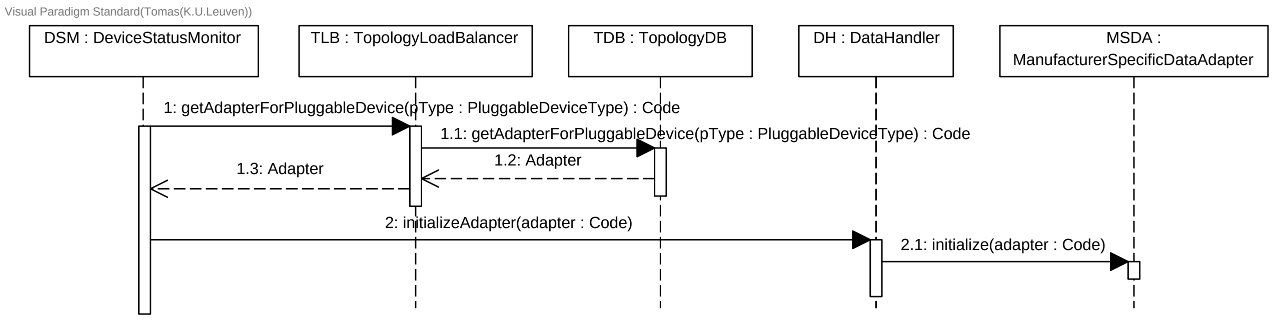

Visual Paradigm Standard(Tomas(K.U.Leuven))
DSM : DeviceStatusMonitor
TLB : TopologyLoadBalancer
TDB : TopologyDB
DH : DataHandler
MSDA : ManufacturerSpecificDataAdapter
1: getAdapterForPluggableDevice(pType : PluggableDeviceType) : Code
1.1: getAdapterForPluggableDevice(pType : PluggableDeviceType) : Code
1.2: Adapter
1.3: Adapter
2: initializeAdapter(adapter : Code)
2.1: initialize(adapter : Code)

Figure D.36: RouteAdapterToGateway

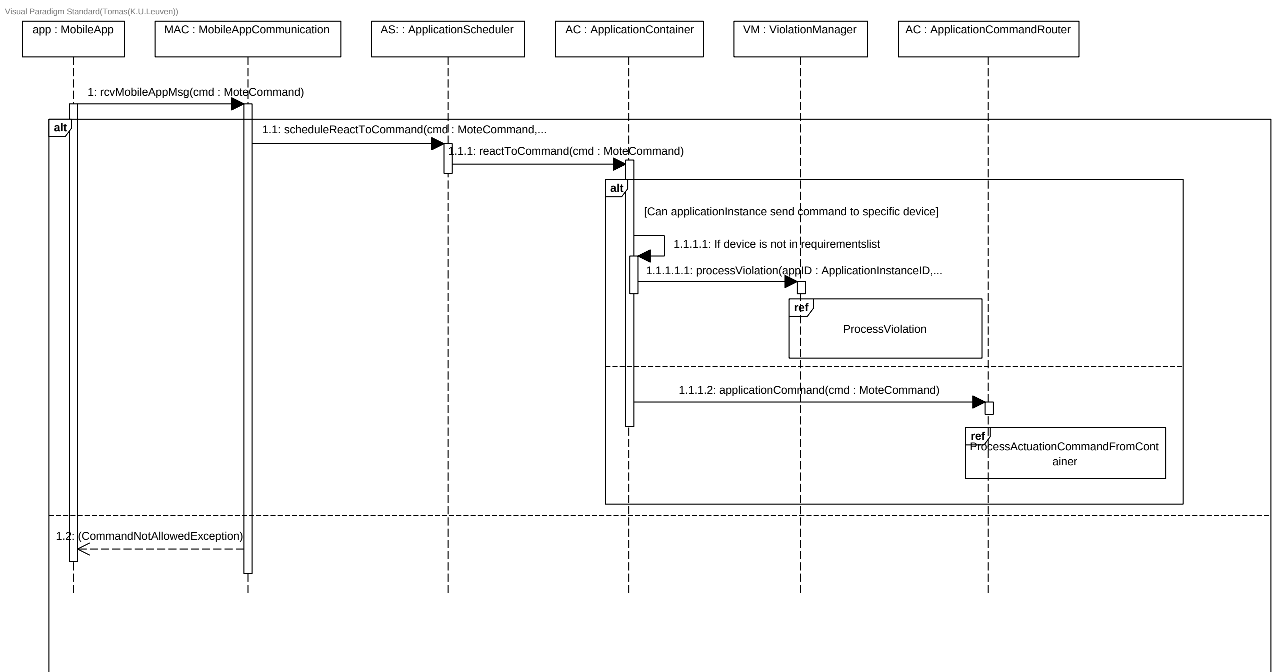

Visual Paradigm Standard(Tomas(K.U.Leuven))
app : MobileApp
MAC : MobileAppCommunication
AS: : ApplicationScheduler
AC : ApplicationContainer
VM : ViolationManager
AC : ApplicationCommandRouter
1: rcvMobileAppMsg(cmd : MoteCommand)
alt
1.1: scheduleReactToCommand(cmd : MoteCommand,...
1.1.1: reactToCommand(cmd : MoteCommand)
alt
[Can applicationInstance send command to specific device]
1.1.1.1: If device is not in requirementslist
1.1.1.1.1: processViolation(appID : ApplicationInstanceID,...
ref
ProcessViolation
1.1.1.2: applicationCommand(cmd : MoteCommand)
ref
ProcessActuationCommandFromContainer
1.2: (CommandNotAllowedException)

Figure D.37: AcuationCommandFromMobileApp

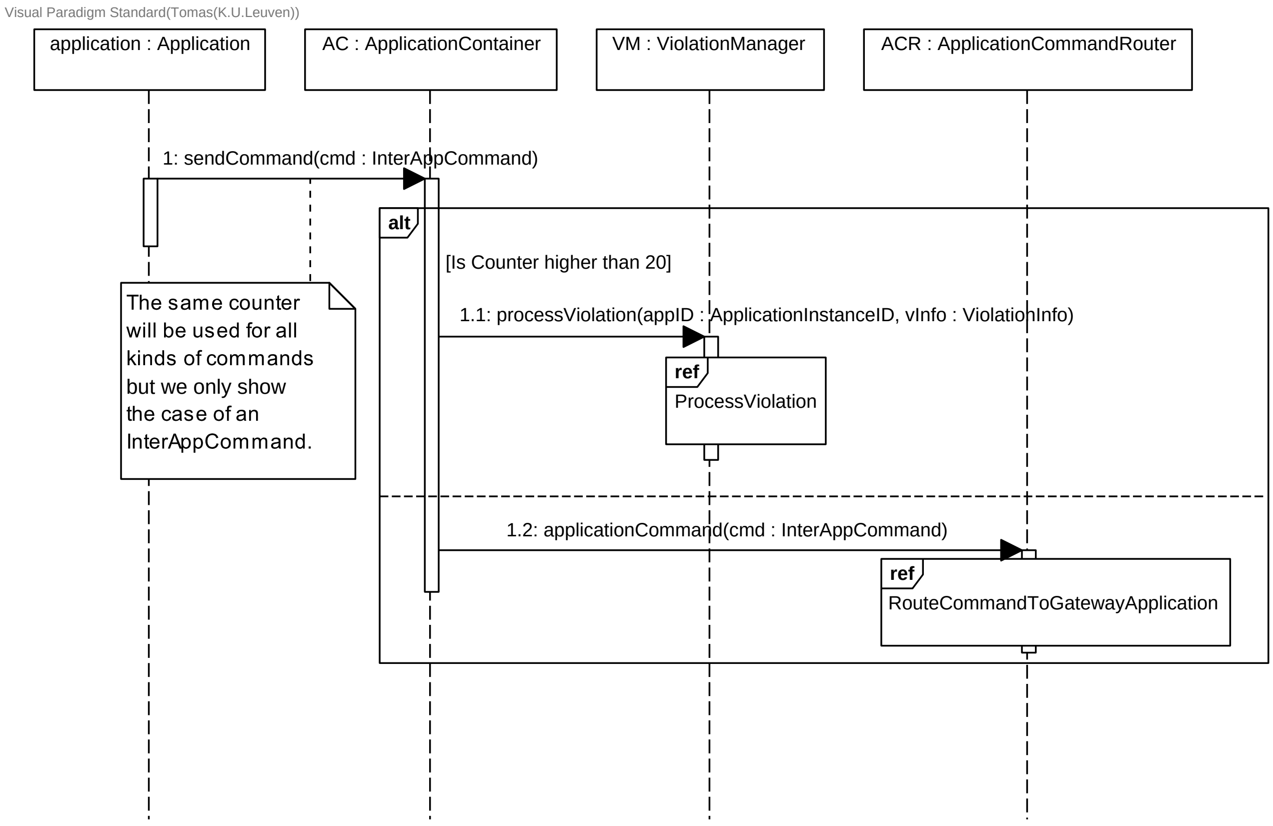

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
AC : ApplicationContainer
VM : ViolationManager
ACR : ApplicationCommandRouter
1: sendCommand(cmd : InterAppCommand)
alt
[Is Counter higher than 20]
1.1: processViolation(appID : ApplicationInstanceID, vInfo : ViolationInfo)
ref
ProcessViolation
The same counter will be used for all kinds of commands but we only show the case of an InterAppCommand.
1.2: applicationCommand(cmd : InterAppCommand)
ref
RouteCommandToGatewayApplication

Figure D.38: RegisterTooManyRequestViolation

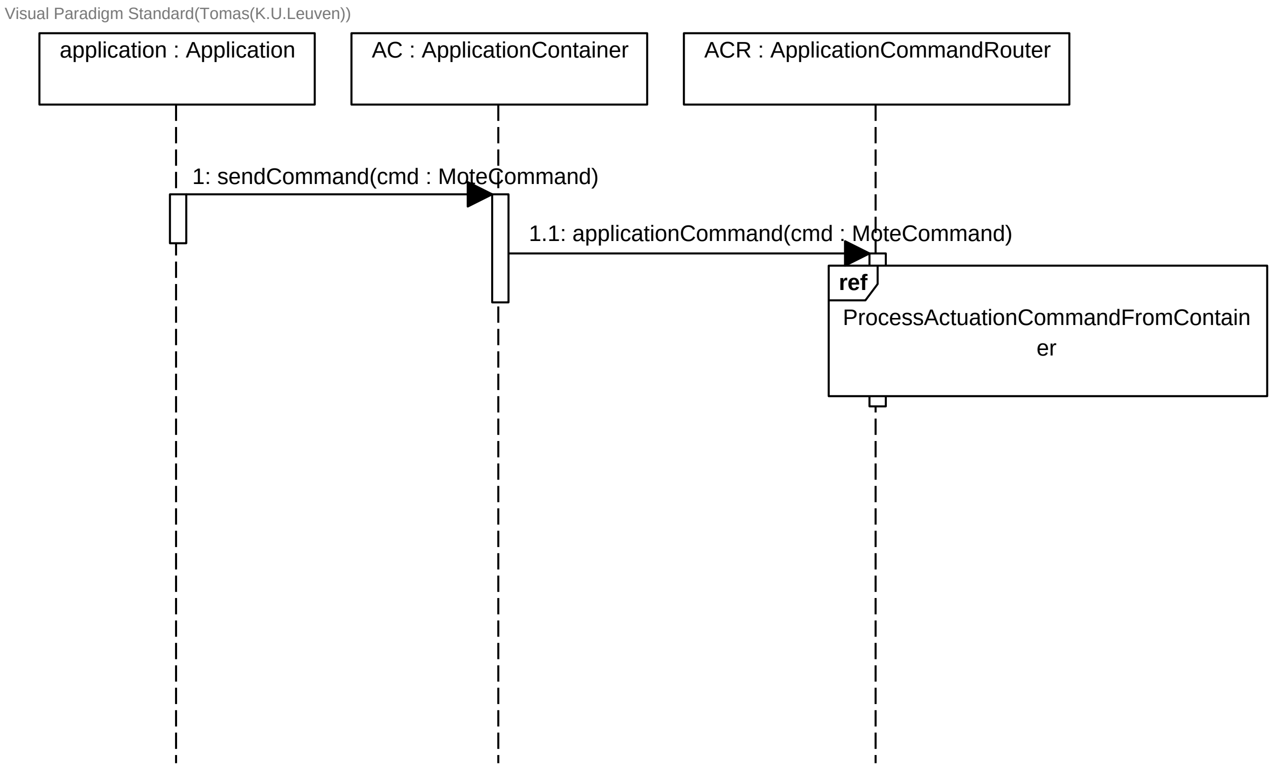

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
AC : ApplicationContainer
ACR : ApplicationCommandRouter
1: sendCommand(cmd : MoteCommand)
1.1: applicationCommand(cmd : MoteCommand)
ref
ProcessActuationCommandFromContainer

Figure D.39: RouteMoteCommand

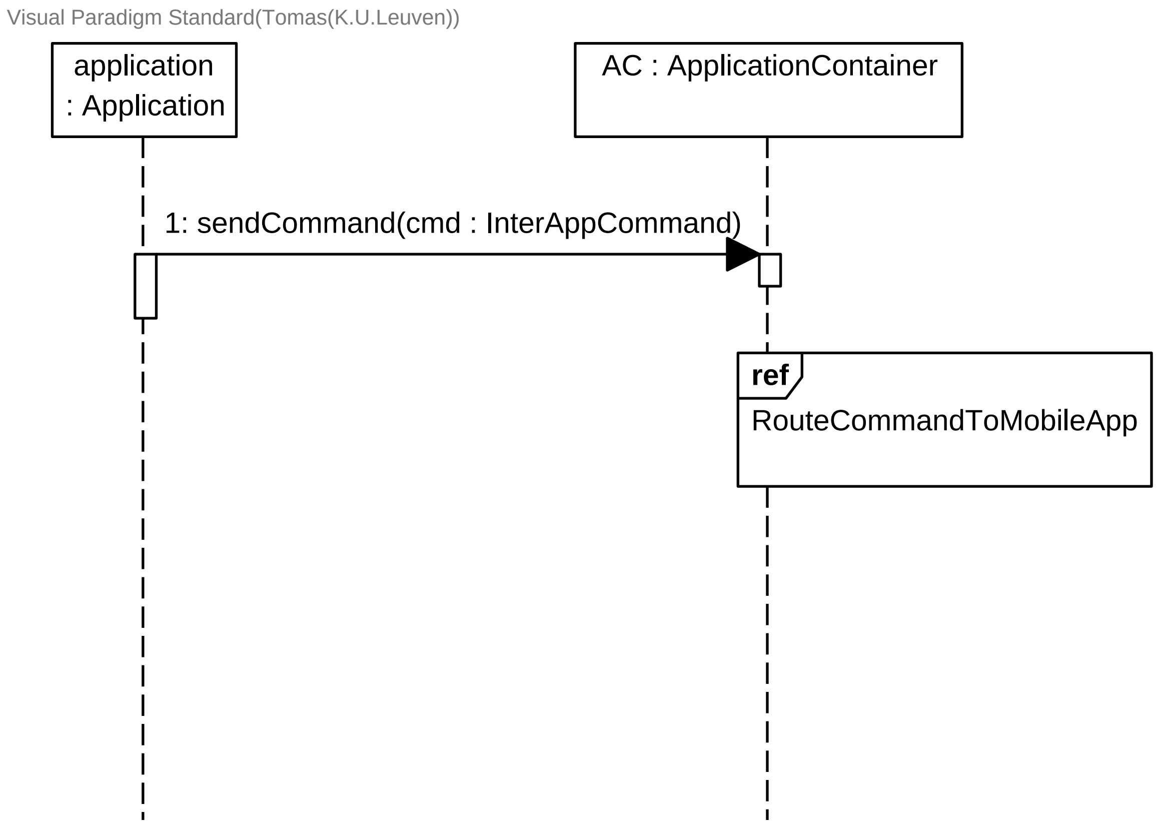

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
AC : ApplicationContainer
1: sendCommand(cmd : InterAppCommand)
ref
RouteCommandToMobileApp

Figure D.40: RouteCommandFromAppToMobileApp

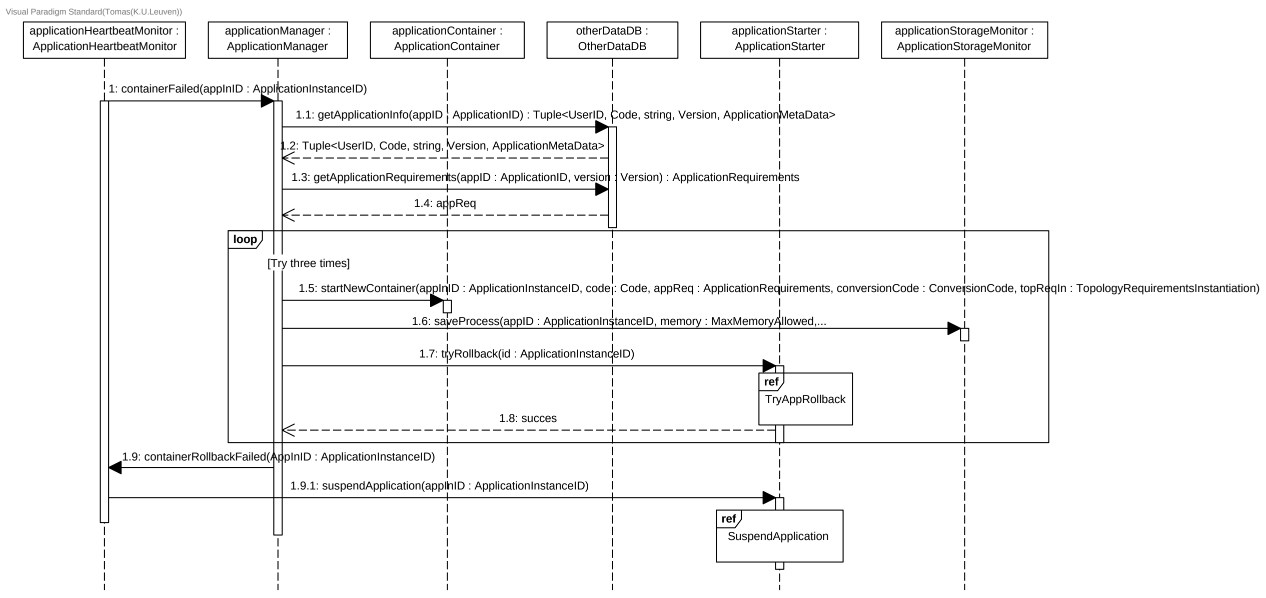

Visual Paradigm Standard(Tomas(K.U.Leuven))
applicationHeartbeatMonitor : ApplicationHeartbeatMonitor
applicationManager : ApplicationManager
applicationContainer : ApplicationContainer
otherDataDB : OtherDataDB
applicationStarter : ApplicationStarter
applicationStorageMonitor : ApplicationStorageMonitor
1: containerFailed(appInID : ApplicationInstanceID)
1.1: getApplicationInfo(appID : ApplicationID) : Tuple<UserID, Code, string, Version, ApplicationMetaData>
1.2: Tuple<UserID, Code, string, Version, ApplicationMetaData>
1.3: getApplicationRequirements(appID : ApplicationID, version : Version) : ApplicationRequirements
1.4: appReq
loop
[Try three times]
1.5: startNewContainer(appInID : ApplicationInstanceID, code : Code, appReq : ApplicationRequirements, conversionCode : ConversionCode, topReqIn : TopologyRequirementsInstantiation)
1.6: saveProcess(appID : ApplicationInstanceID, memory : MaxMemoryAllowed,...
1.7: tryRollback(id : ApplicationInstanceID)
ref
TryAppRollback
1.8: succes
1.9: containerRollbackFailed(AppInID : ApplicationInstanceID)
1.9.1: suspendApplication(appInID : ApplicationInstanceID)
ref
SuspendApplication

Figure D.41: ApplicationContainerCrash

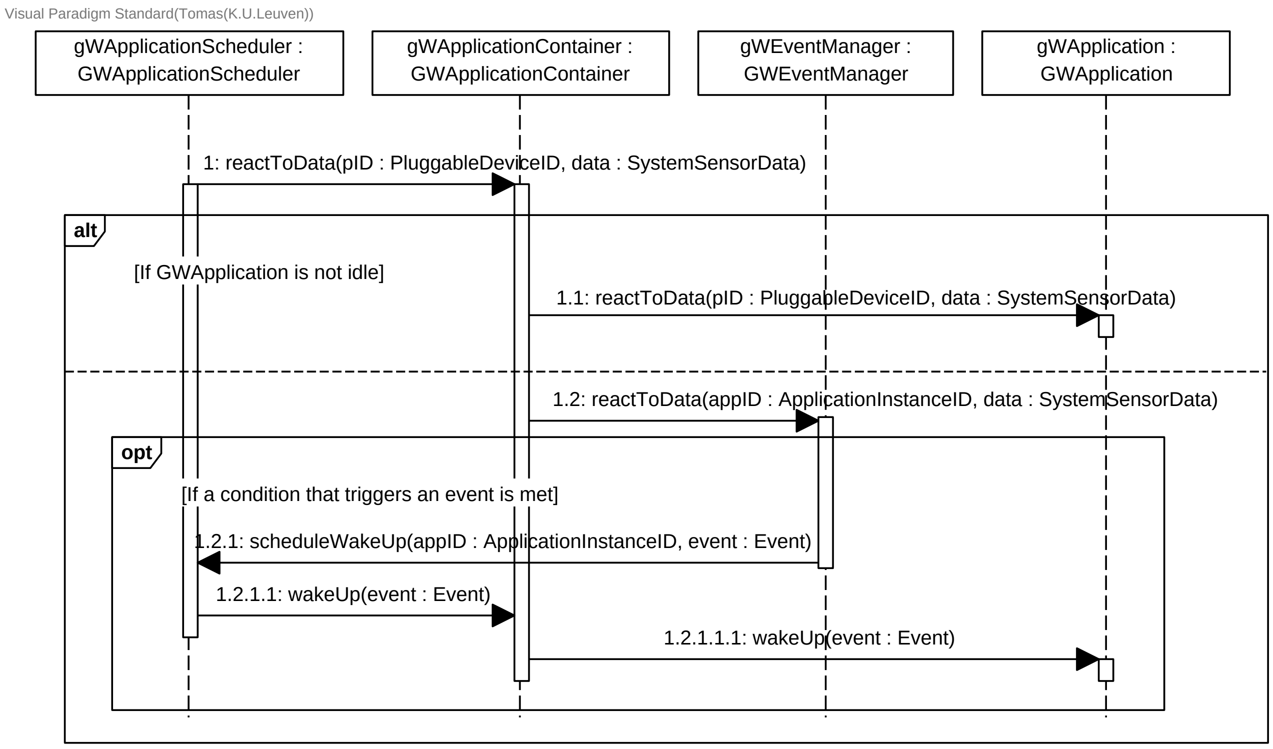

Visual Paradigm Standard(Tomas(K.U.Leuven))
gWApplicationScheduler : GWApplicationScheduler
gWApplicationContainer : GWApplicationContainer
gWEventManager : GWEventManager
gWApplication : GWApplication
1: reactToData(pID : PluggableDeviceID, data : SystemSensorData)
alt
[If GWApplication is not idle]
1.1: reactToData(pID : PluggableDeviceID, data : SystemSensorData)
1.2: reactToData(appID : ApplicationInstanceID, data : SystemSensorData)
opt
[If a condition that triggers an event is met]
1.2.1: scheduleWakeUp(appID : ApplicationInstanceID, event : Event)
1.2.1.1: wakeUp(event : Event)
1.2.1.1.1: wakeUp(event : Event)

Figure D.42: GWReactToData

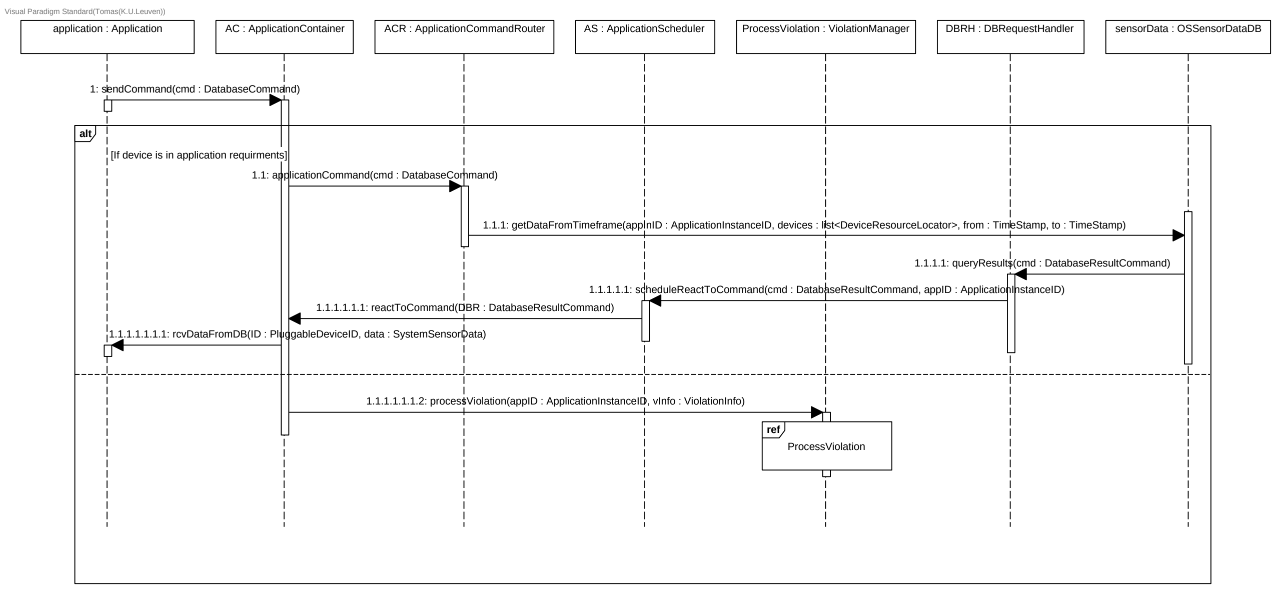

Visual Paradigm Standard(Tomas(K.U.Leuven))
application : Application
AC : ApplicationContainer
ACR : ApplicationCommandRouter
AS : ApplicationScheduler
ProcessViolation : ViolationManager
DBRH : DBRequestHandler
sensorData : OSSensorDataDB
1: sendCommand(cmd : DatabaseCommand)
alt
[If device is in application requirments]
1.1: applicationCommand(cmd : DatabaseCommand)
1.1.1: getDataFromTimeframe(appInID : ApplicationInstanceID, devices : list<DeviceResourceLocator>, from : TimeStamp, to : TimeStamp)
1.1.1.1: queryResults(cmd : DatabaseResultCommand)
1.1.1.1.1: scheduleReactToCommand(cmd : DatabaseResultCommand, appID : ApplicationInstanceID)
1.1.1.1.1.1: reactToCommand(DBR : DatabaseResultCommand)
1.1.1.1.1.1.1: rcvDataFromDB(ID : PluggableDeviceID, data : SystemSensorData)
1.1.1.1.1.1.2: processViolation(appID : ApplicationInstanceID, vInfo : ViolationInfo)
ref
ProcessViolation

Figure D.43: RequestSensorData

E. Element catalog

E.1 Components

E.1.1 Application

Responsibility: Represents an application instance of a certain Customer Organisation for an application subscription. Although this physically runs on the `SIoTIP system` it is still communicated with as an external entity.
Super-components: None
Sub-components: None
Provided interfaces: -○ `AppProvidesAPI`
Required interfaces: -< `AppUsesAPI`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.1, C.1, D.1, D.2, D.3, D.9, D.15, D.19, D.27, D.29, D.30, D.31, D.34, D.35, D.38, D.39, D.40 and D.43

E.1.2 ApplicationCommandPropagator

Responsibility: The `ApplicationCommandPropagator` is responsible for propagating commands from the Online Service to the correct `Gateway`, as well as routing incoming commands to the correct subcomponent to process the command.
Super-components: ⎘ `SIoTIP system` ▷ ⎘ `OSWithGWCommunication`
Sub-components: None
Provided interfaces: -○ `GWToOSCommand`, -○ `SendCommandToGW`
Required interfaces: -< `AckMessage`, -< `ChangeSyncTimer`, -< `CommHeartbeat`, -< `IncomingCommand`, -< `IPLookup`, -< `OSToGWCommand`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. B.2, C.2, D.2, D.14, D.17, D.18, D.22, D.23 and D.28

E.1.3 ApplicationCommandRouter

Responsibility: The `ApplicationCommandRouter` is responsible for routing commands from application instances and mobile apps to the correct location in the system, i.e. another part of the same application instance, an actuator or database. It will also check whether the issued command is legitimate, i.e. is the application instance allowed to issue that command. See the AppCalls interface for more information on the routing.
Super-components: ⎘ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `AppCalls`, -○ `GWAppChange`, -○ `IncomingCommand`
Required interfaces: -< `AppFailure`, -< `AppInstanceMapChange`, -< `DeliverMsgMobileApp`, -< `DeviceAvailabilityCheck`
-< `HandleDBRequest`, -< `OtherDataMgmt`, -< `ScheduleApplication`, -< `SendCommandToGW`, -< `SensorDataMgmt`, -< `TopologyMgmt`, -< `UpdateMessage`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.2, C.2, D.1, D.2, D.13, D.14, D.15, D.17, D.18, D.19, D.22, D.23, D.24, D.25, D.26, D.28, D.37, D.38, D.39 and D.43

E.1.4 ApplicationContainer

Responsibility: The `ApplicationContainer` is responsible for internally running and monitoring an application instance (or the part of it which is running on the Online Service). All interaction with the application instance is done through the `ApplicationContainer`. It will keep track of the local state of the instance and provide it whenever an interface of the container is used. An application uses the container, which has the requirements of the application, to check whether a configuration/other command is valid. If an application expects a sensor data unit different than

the one provided (e.g. degrees Celsius instead of degrees Fahrenheit, found out by checking the **ApplicationRequirements**), the container will convert the unit before handing it to the application by using **ConversionCode**.
Super-components: `ApplicationHandler` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AppContainerMgmt`, `AppUsesAPI`, `CallThroughContainer`
Required interfaces: `AppCalls`, `AppFailure`, `ApplicationActivation`, `AppProvidesAPI`, `MemoryMgmt`, `ProcessInfo`, `ScheduleEvent`, `StateMgmt`, `ViolationProcessor`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.1, B.4, C.1, C.2, D.1, D.2, D.3, D.9, D.15, D.18, D.19, D.24, D.25, D.26, D.27, D.29, D.30, D.31, D.33, D.34, D.35, D.37, D.38, D.39, D.40, D.41 and D.43

E.1.5 ApplicationDeveloperClient

Responsibility: The `ApplicationDeveloperClient` is external to the `SIoTIP system` and represents the client of the application developer that communicates with the `SIoTIP system`.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: `ApplicationInfo`, `ConsultNotifications`, `UploadApplication`, `UserAuthentication`
Deployed on: ApplicationDeveloperNode
Visible on diagrams: figs. A.1, C.1 and D.4

E.1.6 ApplicationDeveloperDashboard

Responsibility: The `ApplicationDeveloperDashboard` provides the main interface to all application developers. It allows them to upload new applications as well as update old ones, gather statistical data and consult notifications.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ApplicationInfo`, `ConsultNotifications`, `UploadApplication`, `UserAuthentication`
Required interfaces: `ApplicationActivation`, `NotifyRecipient`, `OtherDataMgmt`, `TestApplication`, `UserAuthentication`, `VerifySession`
Deployed on: DashboardsNode
Visible on diagrams: figs. A.1, A.2, C.1, C.2 and D.4

E.1.7 ApplicationHandler

Responsibility: The `ApplicationHandler` is responsible for the execution of application instances on the Online Service.
Super-components: `SIoTIP system`
Sub-components: `ApplicationManager`, `ApplicationScheduler`, `ApplicationContainer`
Provided interfaces: `AppInstanceChange`, `AppUsesAPI`, `Failure`, `ScheduleApplication`
Required interfaces: `AppCalls`, `AppFailure`, `AppInstanceMapChange`, `ApplicationActivation`, `AppProvidesAPI`, `MemoryMgmt`, `OtherDataMgmt`, `ProcessInfo`, `ScheduleEvent`, `StateMgmt`, `UpdateMessage`, `ViolationProcessor`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.2 and B.4

E.1.8 ApplicationHeartbeatMonitor

Responsibility: The `ApplicationHeartbeatMonitor` is responsible for detecting failures of application execution components. When an `ApplicationContainer` on a server crashes, the application instance is restarted in a new container, when an `ApplicationScheduler` or `ApplicationCommandRouter` crashes, the entire server is reset and the `ApplicationInstanceInfoDB` lookup map and `DeviceDataRouter` can be reconstructed from the data in the database.

Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `AppFailure`
Required interfaces: `Failure`, `InternalAppActivation`, `NotifyRecipient`
Deployed on: ApplicationFailureNode
Visible on diagrams: figs. A.2, C.2, D.3 and D.41

E.1.9 ApplicationInstanceInfoDB

Responsibility: This component is responsible to keep track of which application instances are interested in which pluggable devices.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `AppInstanceMapChange`, `ApplicationInstanceLookup`
Required interfaces: None
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2, D.1, D.16 and D.26

E.1.10 ApplicationManager

Responsibility: The `ApplicationManager` is responsible to manage calls concerning application instances. This includes starting new instances, stopping existing ones and forwarding incoming requests to the `ApplicationScheduler`.
Super-components: `ApplicationHandler` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AppInstanceChange`, `Failure`, `ScheduleApplication`
Required interfaces: `AppContainerMgmt`, `AppFailure`, `AppInstanceMapChange`, `AppInstanceSuspension`, `ApplicationActivation`, `MemoryMgmt`, `OtherDataMgmt`, `ScheduleApplication`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. B.4, C.2, D.1, D.15, D.16, D.22, D.23, D.24, D.25, D.26, D.27, D.35 and D.41

E.1.11 ApplicationScheduler

Responsibility: The `ApplicationScheduler` is responsible for scheduling all information/commands to the various application instances. This includes answers from databases, commands, synchronization data from Gateways and wake up calls in case the application wishes asynchronous updates. It will route any of those requests to the corresponding `ApplicationContainer` where they will be relayed to the `Application` instance.
Super-components: `ApplicationHandler` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AppInstanceSuspension`, `ScheduleApplication`
Required interfaces: `AppContainerMgmt`, `AppFailure`, `CallThroughContainer`, `UpdateMessage`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. B.4, C.2, D.1, D.3, D.9, D.15, D.16, D.22, D.23, D.24, D.25, D.26, D.27, D.30, D.31, D.37 and D.43

E.1.12 ApplicationStarter

Responsibility: The `ApplicationStarter` is responsible for activating, deactivating and updating application instances. Whenever it gets a request to start a new application of some type, it will spread the required information to do so in a correct order to various other parts of the system (e.g. starting a new `ApplicationContainer` with the given code). Certain events (end-user role assignment, pluggable device failure, etc.) might trigger the `ApplicationStarter` to check wether any of its three responsibilities need to be executed.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ApplicationActivation`, `InternalAppActivation`, `UpdateMessage`

Required interfaces: ≺ `AppInstanceChange`, ≺ `AppInstanceMapChange`, ≺ `DeviceAvailabilityCheck`, ≺ `GWAppChange`, ≺ `NotifyRecipient`, ≺ `OtherDataMgmt`, ≺ `ScheduleApplication`, ≺ `StateMgmt`, ≺ `TopologyMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2, D.1, D.3, D.4, D.6, D.7, D.8, D.21, D.22, D.23, D.24, D.25, D.26, D.27, D.32, D.33, D.35 and D.41

E.1.13 ApplicationStorageMonitor

Responsibility: The `ApplicationStorageMonitor` stores the processID(from the operating system or file system perspective) and runs a daemon process which monitors the memory taken by each process. It is the responsibility of the application developer to set the max limit for any process. If any process tries to use more than max allowed memory then applicationStarter will be asked to fully suspend the application.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: ⊸ `MemoryMgmt`
Required interfaces: ≺ `AppNotifications`, ≺ `InternalAppActivation`, ≺ `OtherDataMgmt`
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.2, C.2, D.1, D.32 and D.41

E.1.14 AppStateDataDB

Responsibility: The `AppStateDataDB` is responsible for actually storing the states of applications.
Super-components: `OSAppStateDB` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: ⊸ `StateMgmt`
Required interfaces: None
Deployed on: AppStateDBNode
Visible on diagrams: figs. B.5, C.2, D.8, D.34 and D.35

E.1.15 AppStateDataLoadBalancer

Responsibility: The `AppStateDataLoadBalancer` is responsible for sending database queries (reads/writes) to the correct server on which part of the application state data is located.
Super-components: `OSAppStateDB` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: ⊸ `StateMgmt`
Required interfaces: ≺ `StateMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. B.5, C.2, D.8, D.34 and D.35

E.1.16 AppStateDataScheduler

Responsibility: The `AppStateDataScheduler` is responsible for handling all read/write requests to the `AppStateDataDB`. It should prioritize requests from critical applications. Furthermore, it should automatically remove states older than two weeks.
Super-components: `OSAppStateDB` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: ⊸ `StateMgmt`
Required interfaces: ≺ `StateMgmt`
Deployed on: AppStateDBNode
Visible on diagrams: figs. B.5, C.2, D.8, D.34 and D.35

E.1.17 AppTester

Responsibility: Responsible for running various tests on uploaded applications. If the tests fail, notifications will have to be sent out to certain parties, if they succeed, the application is made

available to Customer Organisations and (if needed) updates to running application instances are initiated. If the tests fail the application developer is notified.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `TestApplication`
Required interfaces: None
Deployed on: AppTestingNode
Visible on diagrams: figs. A.2, C.2 and D.4

E.1.18 AuthenticationHandler

Responsibility: The `AuthenticationHandler` is responsible for authenticating all end-users of the `SIoTIP system` dashboards. The authentication is done by username and password.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `UserAuthentication`, `VerifySession`
Required interfaces: `OtherDataMgmt`, `SessionMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2 and C.2

E.1.19 AvailabilityMonitor

Responsibility: The `AvailabilityMonitor` is responsible for keeping track of the motes connected to the gateway and their pluggable devices. Whenever the status of either of those changes (e.g. new pluggable added or a mote fails), it sends a message to the Online Service.
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AckMessage`, `AMReactToFailure`, `DeviceManagement`, `Heartbeat`
Required interfaces: `DeviceAvailabilityMessage`, `FailureDetection`
Deployed on: GatewayNode
Visible on diagrams: figs. A.1, B.1, C.1, C.2, D.5, D.7 and D.10

E.1.20 CommunicationHeartbeatMonitor

Responsibility: The `CommunicationHeartbeatMonitor` is responsible for heartbeating the various components on the Online Service which communicate with the Gateways to make sure they (the components) have not gone down.
Super-components: `SIoTIP system` ▷ `OSWithGWCommunication`
Sub-components: None
Provided interfaces: `CommHeartbeat`
Required interfaces: None
Deployed on: CommunicationFailureNode
Visible on diagrams: figs. B.2 and C.2

E.1.21 CustomerOrganisationClient

Responsibility: The `CustomerOrganisationClient` is external to the `SIoTIP system` and represents the client of the Customer Organisation that communicates with the `SIoTIP system`.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: `ConsultNotifications`, `Registration`, `Subscription`, `UserAuthentication`
Deployed on: CustomerOrganisationNode
Visible on diagrams: figs. A.1, C.1, D.12 and D.21

E.1.22 CustomerOrganisationDashboard

Responsibility: The `CustomerOrganisationDashboard` provides the main interface to Customer Organisations, it can be used to register a new organisation or by a registered organisation to

subscribe to applications. Notifications for Customer Organisations can also be requested. Timers are used to timeout registration links.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ConsultNotifications`, `Registration`, `Subscription`, `UserAuthentication`
Required interfaces: `ApplicationActivation`, `OtherDataMgmt`, `TopologyMgmt`, `UserAuthentication`, `VerifySession`
Deployed on: DashboardsNode
Visible on diagrams: figs. A.1, A.2, C.1, C.2, D.12, D.21 and D.23

E.1.23 DataHandler

Responsibility: The `DataHandler` is responsible for translating between the data format the pluggable devices use and one used inside of the `SIoTIP system` (in both directions). It will propagate any incoming sensor data from the sensors to the Online Service (and if needed to a part of the application instance running on the `Gateway`).
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AckMessage`, `AppInstanceMapChange`, `DHReactToFailure`, `DHUpdate`, `SensorData`
Required interfaces: `AckMessage`, `FailureDetection`, `GWTempDataMgmgt`, `RcvSensorData`, `RequestData`, `ScheduleApplication`
Deployed on: GatewayNode
Visible on diagrams: figs. A.1, B.1, C.1, C.2, D.5, D.11 and D.36

E.1.24 DataReceiver

Responsibility: The `DataReceiver` is responsible for propagating sensor data received from a gateway to the `OSSensorDataDB`, as well as route it to the application execution subsystem. It will warn the `DeviceStatusMonitor` if no data has been received for a while as these are used as heartbeats. It uses timers to detect whether a `Gateway` has failed or when a new gateway starts sending out hearbeats.
Super-components: `SIoTIP system` ▷ `OSWithGWCommunication`
Sub-components: None
Provided interfaces: `ChangeSyncTimer`, `RcvSensorData`
Required interfaces: `AckMessage`, `CommHeartbeat`, `DeviceInitCacheMgmt`, `GWAvailability`, `RouteDeviceData`, `SensorDataMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. B.2, C.2, D.6 and D.11

E.1.25 DBRequestHandler

Responsibility: The `DBRequestHandler` will forward requests by application instances to the `OSSensorDataDB` and `TopologyLoadBalancer`. Once It recieves an answer, it forwards that information to the `ApplicationScheduler`. These requests concern historical data and overview of the topology.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `DBRequestResults`, `HandleDBRequest`
Required interfaces: `ScheduleApplication`, `SensorDataMgmt`, `TopologyMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2 and D.43

E.1.26 DeviceDataRouter

Responsibility: Responsible for routing incoming data to the correct application instances by scheduling the receival of this data by the interested applications via the `ApplicationManager`.
Super-components: `SIoTIP system`

Sub-components: None
Provided interfaces: `RouteDeviceData`
Required interfaces: `ApplicationInstanceLookup`, `ScheduleApplication`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2, D.11 and D.16

E.1.27 DeviceInitCache

Responsibility: The `DeviceInitCache` provides a cache of the status of pluggable devices which can be used to determine whether a pluggable device has been initialized yet. Useful to reduce load on the `TopologyDB` by avoiding doing a database call every time sensor data arrives.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `DeviceInitCacheMgmt`
Required interfaces: None
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2 and D.11

E.1.28 DeviceStatusMonitor

Responsibility: The `DeviceStatusMonitor` is responsible for monitoring the status of the gateways, motes and their pluggable devices. It will recieve both direct messages from the `Gateway` informing of failures as well as information from the `DataReceiver` whose messages act as heartbeats to detect `Gateway` failure. It will propagate which devices have become unavailable as a result to the `ApplicationStarter`. Other communication components use this component to lookup `Gateway` IP addresses based on `Gateway` IDs.
Super-components: `SIoTIP system` ▷ `OSWithGWCommunication`
Sub-components: None
Provided interfaces: `AckMessage`, `DeviceAvailabilityCheck`, `DeviceAvailabilityMessage`, `GWAvailability`, `IPLookup`
Required interfaces: `AckMessage`, `ApplicationActivation`, `ChangeSyncTimer`, `CommHeartbeat`, `DHUpdate`, `NotifyRecipient`, `TopologyMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. B.2, C.2, D.2, D.6, D.7, D.10, D.17, D.18, D.22 and D.36

E.1.29 EventDataDB

Responsibility: The `EventDataDB` stores different kinds of events for application parts on the online service. Events can be conditional events (e.g. temperature exceeding 30C) or timed events set by the application. Note that this should be a fairly small database.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `EventMgmt`
Required interfaces: None
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.2, C.2 and D.29

E.1.30 EventManager

Responsibility: The `EventManager` is responsible for handling different kind of events for application parts on the online service. Events can be conditional events (e.g. temperature exceeding 30C) or timed events set by the application. For events that happen when an application has not received data for a set amount of time it keeps a timer that is refreshed every time data comes in.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ScheduleEvent`
Required interfaces: `EventMgmt`, `ScheduleApplication`
Deployed on: ApplicationExecutionNode

Visible on diagrams: figs. A.2, C.2, D.29, D.30 and D.31

E.1.31 Gateway

Responsibility: The `Gateway` is responsible for all the parts of the system that are expected to be executed on a gateway. This includes running parts of application instances, communication with the Online Service and communication with motes and their pluggable devices.
Super-components: `SIoTIP system`
Sub-components: `GWApplicationContainer`, `GWApplicationScheduler`, `GWApplicationHeartbeatMonitor`, `GWCommandRouter`, `GWFailureHandler`, `GWTempDataDB`, `AvailabilityMonitor`, `DataHandler`, `ManufacturerSpecificDataAdapter`, `GWEventDataDB`, `GWEventManager`, `GWApplicationStorageMonitor`
Provided interfaces: `AckMessage`, `DeviceManagement`, `DHUpdate`, `GWAppUsesAPI`, `Heartbeat`, `OSToGWCommand`, `PnPSensorData`
Required interfaces: `Actuate`, `Config`, `DeviceAvailabilityMessage`, `GWAppProvidesAPI`, `GWToOSCommand`, `RcvSensorData`, `RebootGW`, `RequestPnPData`
Deployed on: GatewayNode
Visible on diagrams: figs. A.2 and B.1

E.1.32 GatewayOS

Responsibility: Represents the operating system of the `Gateway`.
Super-components: None
Sub-components: None
Provided interfaces: `RebootGW`
Required interfaces: None
Deployed on: GatewayNode
Visible on diagrams: figs. A.1 and C.1

E.1.33 GWApplication

Responsibility: Represents an application instance of a certain customer organisation for an application part running on the `Gateway`. This physically runs on the `SIoTIP system` but is still communicated with as an external entity.
Super-components: None
Sub-components: None
Provided interfaces: `GWAppProvidesAPI`
Required interfaces: `GWAppUsesAPI`
Deployed on: GatewayNode
Visible on diagrams: figs. A.1, C.1, D.14 and D.42

E.1.34 GWApplicationContainer

Responsibility: The `GWApplicationContainer` is responsible for internally running and monitoring a `GWApplication` instance. All interaction with the application instance is done through the `GWApplicationContainer`. It will keep track of the local state of the instance and provide it whenever an interface of the container is used. An application uses the container (which has the applications' requirements) to check whether a configuration/other command is valid. If an application expects a sensor data unit different than the one provided (e.g. degrees Celsius instead of degress Fahrenheit, found out by checking the **ApplicationRequirements**), the container will convert the unit before handing it to the application by using **ConversionCode**.
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AppInstanceChange`, `CallThroughContainer`, `GWAppUsesAPI`
Required interfaces: `AppFailure`, `GWAppCalls`, `GWAppProvidesAPI`, `MemoryMgmt`, `ScheduleEvent`
Deployed on: GatewayNode
Visible on diagrams: figs. A.1, B.1, C.1, C.2, D.14 and D.42

E.1.35 GWApplicationHeartbeatMonitor

Responsibility: This component monitors the application instances executing on a `Gateway` as well as the components running them by keeping of their heartbeats (or lack thereof).
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AppFailure`
Required interfaces: None
Deployed on: GatewayNode
Visible on diagrams: figs. B.1 and C.2

E.1.36 GWApplicationScheduler

Responsibility: The `GWApplicationScheduler` is responsible for scheduling all information/commands, such as sensor data, to the various application instances on a `Gateway`.
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AppInstanceSuspension`, `ScheduleApplication`
Required interfaces: `AppFailure`, `CallThroughContainer`
Deployed on: GatewayNode
Visible on diagrams: figs. B.1, C.2, D.11, D.14, D.22, D.23 and D.42

E.1.37 GWApplicationStorageMonitor

Responsibility: The `ApplicationStorageMonitor` stores the processID(from the operating system or file system perspective) and runs a daemon process which monitors the memory taken by each process. It is the responsibility of the application container to set the max limit for any process. If any process tries to use more than max allowed memory then the application will be suspended on gateway and it's counter will be notified about this event. If the gateway has no connection then this command will be saved and online service will be notified later.
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `MemoryMgmt`
Required interfaces: `AppInstanceSuspension`, `GWAppCalls`
Deployed on: GatewayNode
Visible on diagrams: figs. B.1, C.1 and C.2

E.1.38 GWCommandRouter

Responsibility: The `GWCommandRouter` is responsible for routing commands from the Online Service to the correct component on the `Gateway` and vice versa. Incoming actuation commands are forwarded to the correct actuators and application commands, including commands to (de)activate or suspend instances, are forwarded to the `GWApplicationScheduler`.
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None
Provided interfaces: `AckMessage`, `CRReactToFailure`, `GWAppCalls`, `OSToGWCommand`
Required interfaces: `Actuate`, `AppInstanceChange`, `AppInstanceMapChange`, `AppInstanceSuspension`, `Config`, `FailureDetection`, `GWTempDataMgmgt`, `GWToOSCommand`, `ScheduleApplication`
Deployed on: GatewayNode
Visible on diagrams: figs. A.1, B.1, C.1, C.2, D.1, D.2, D.5, D.14, D.17, D.18, D.22, D.23, D.24, D.25, D.26 and D.28

E.1.39 GWEventDataDB

Responsibility: The `GWEventDataDB` stores different kinds of events for application parts on the gateway. Events can be conditional events (e.g. temperature exceeding 30C) or timed events set by the application.Note that this should be a small database.
Super-components: `Gateway` ▷ `SIoTIP system`
Sub-components: None

Provided interfaces: -○ `EventMgmt`
Required interfaces: None
Deployed on: GatewayNode
Visible on diagrams: figs. B.1 and C.2

E.1.40 GWEventManager

Responsibility: The `GWEventManager` is responsible for handling different kind of events for application parts on the gateway. Events can be conditional events (e.g. temperature exceeding 30C) or timed events set by the application. For events that happen when an application has not received data for a set amount of time it keeps a timer that is refreshed every time data comes in.
Super-components: ▣ `Gateway` ▷ ▣ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `ScheduleEvent`
Required interfaces: -< `EventMgmt`, -< `ScheduleApplication`
Deployed on: GatewayNode
Visible on diagrams: figs. B.1, C.2 and D.42

E.1.41 GWFailureHandler

Responsibility: The `GWFailureHandler` is responsible for reacting to and resolving various kinds of failures on the `Gateway` or the communication channel between the `Gateway` and the Online Service. This includes the failure of communication components running on the `Gateway` (monitored by using heartbeats) and the failure of the communication channel (reported by the three communication components when they do not receive an acknowledgement on a message from the Online Service).
Super-components: ▣ `Gateway` ▷ ▣ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `FailureDetection`
Required interfaces: -< `AMReactToFailure`, -< `CRReactToFailure`, -< `DHReactToFailure`, -< `RebootGW`
Deployed on: GatewayNode
Visible on diagrams: figs. A.1, B.1, C.2 and D.5

E.1.42 GWTempDataDB

Responsibility: The `GWTempDataDB` is responsible for storing (a limited amount of) sensor and actuation data when there is a communication failure with the Online System.
Super-components: ▣ `Gateway` ▷ ▣ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `GWTempDataMgmgt`
Required interfaces: None
Deployed on: GatewayNode
Visible on diagrams: figs. B.1 and C.2

E.1.43 InfrastructureOwnerClient

Responsibility: The `InfrastructureOwnerClient` is external to the `SIoTIP system` and represents the client of the infrastructure owner that communicates with the `SIoTIP system`.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: -< `ConsultNotifications`, -< `ManageDevices`, -< `UserAuthentication`
Deployed on: InfrastructureOwnerNode
Visible on diagrams: figs. A.1 and C.1

E.1.44 InfrastructureOwnerDashboard

Responsibility: The `InfrastructureOwnerDashboard` provides the main interface to any Infrastructure Owners, it is used to configure devices as well as consulting notifications.
Super-components: ▣ `SIoTIP system`

Sub-components: None
Provided interfaces: ⊸ `ConsultNotifications`, ⊸ `ManageDevices`, ⊸ `UserAuthentication`
Required interfaces: ≺ `TopologyMgmt`, ≺ `UserAuthentication`, ≺ `VerifySession`
Deployed on: DashboardsNode
Visible on diagrams: figs. A.1, A.2, C.1 and C.2

E.1.45 InvoiceManager

Responsibility: The `InvoiceManager` is responsible for sending invoices for used products to a third party invoicing service (e.g. Zoomit or a credit card company). At the end of each month the `InvoiceManager` will check which subscriptions that month have run and for how long, it will use the `OtherDataDB` to get information on both the subscriptions and the payment options. Invoices will also be sent whenever an unsubscription happens.
Super-components: 🗐 `SIoTIP system`
Sub-components: None
Provided interfaces: None
Required interfaces: ≺ `OtherDataMgmt`, ≺ `SendInvoice`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.1, A.2, C.1 and C.2

E.1.46 ManufacturerSpecificDataAdapter

Responsibility: This component is responsible for translating a manufacturer specific API into an API used by the `DataHandler`. Initially the only `ManufacturerSpecificDataAdapter` is specific to MicroPnP
Super-components: 🗐 `Gateway` ▷ 🗐 `SIoTIP system`
Sub-components: None
Provided interfaces: ⊸ `PnPSensorData`, ⊸ `RequestData`
Required interfaces: ≺ `RequestPnPData`, ≺ `SensorData`
Deployed on: GatewayNode
Visible on diagrams: figs. B.1, C.2 and D.36

E.1.47 MobileApp

Responsibility: A mobile application acting as front-end for an application.
Super-components: None
Sub-components: None
Provided interfaces: ⊸ `SendMobileAppMsg`
Required interfaces: ≺ `RcvMobileAppMsg`
Deployed on: MobileDevice
Visible on diagrams: figs. A.1, A.2, C.1, D.13 and D.37

E.1.48 MobileAppCommunication

Responsibility: This component is responsible for communication with mobile applications acting as front-ends. It will propagate any commands issued (if deemed valid) by the external actor to the `ApplicationCommandRouter` as well as send data back to any connected mobile applications.
Super-components: 🗐 `SIoTIP system`
Sub-components: None
Provided interfaces: ⊸ `DeliverMsgMobileApp`, ⊸ `RcvMobileAppMsg`
Required interfaces: ≺ `IncomingCommand`, ≺ `ScheduleApplication`, ≺ `SendMobileAppMsg`
Deployed on: MobileAppCommunicationNode
Visible on diagrams: figs. A.1, A.2, C.1, C.2, D.13 and D.37

E.1.49 Mote

Responsibility: The `Mote` is external to the `SIoTIP system` and represents the hardware device in which the various pluggable devices the system uses are plugged in.
Super-components: None

Sub-components: None
Provided interfaces: `Actuate`, `Config`, `RequestPnPData`
Required interfaces: `DeviceManagement`, `Heartbeat`, `PnPSensorData`
Deployed on: MoteNode
Visible on diagrams: figs. A.1, C.1, D.2, D.7, D.10 and D.18

E.1.50 NotificationHandler

Responsibility: The `NotificationHandler` is responsible for sending notifications to the appropriate parties, e.g. system administrators, customer organisations and application developers. The `OtherDataDB` is used to determine in what way the notifications will be sent. When only the trigger is supplied, the `NotificationHandler` will look at end-roles and other relationships to determine the receiver. If applicable, it will act as an interface for confirming deliveries of messages.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `AppNotifications`, `ConfirmDelivery`, `NotifyRecipient`
Required interfaces: `ConsultNotifications`, `DeliverNotification`, `OtherDataMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.1, A.2, C.1, C.2, D.4, D.6, D.7, D.10, D.20, D.22 and D.32

E.1.51 OSAppStateDB

Responsibility: The `OSAppStateDB` is responsible for storing historic states from all applications. It should remove states once they are older than two weeks It employs a scheduling policy for database requests.
Super-components: `SIoTIP system`
Sub-components: `AppStateDataDB`, `AppStateDataLoadBalancer`, `AppStateDataScheduler`
Provided interfaces: `StateMgmt`
Required interfaces: None
Deployed on: OnlineServiceNode, AppStateDBNode
Visible on diagrams: figs. A.2 and B.5

E.1.52 OSKernel

Responsibility: Represents the kernell of the system.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ProcessInfo`
Required interfaces: None
Deployed on: ApplicationExecutionNode
Visible on diagrams: figs. A.2, C.2 and D.1

E.1.53 OSSensorDataDB

Responsibility: The `OSSensorDataDB` is responsible for storing sensor data sent by the Gateways. It employs a scheduling policy for database requests.
Super-components: `SIoTIP system`
Sub-components: `SensorDataLoadBalancer`, `SensorDataScheduler`, `SensorDataDB`
Provided interfaces: `SensorDataMgmt`
Required interfaces: `DBRequestResults`
Deployed on: OnlineServiceNode, SensorDBNode
Visible on diagrams: figs. A.2, B.3 and D.43

E.1.54 OSWithGWCommunication

Responsibility: The `OSWithGWCommunication` is a supercomponent responsible for all communication with the Gateways (incoming and outgoing).
Super-components: `SIoTIP system`

Sub-components: ▭ `DataReceiver`, ▭ `DeviceStatusMonitor`, ▭ `CommunicationHeartbeatMonitor`, ▭ `ApplicationCommandPropagator`
Provided interfaces: -○ `DeviceAvailabilityCheck`, -○ `DeviceAvailabilityMessage`, -○ `GWToOSCommand`, -○ `RcvSensorData`, -○ `SendCommandToGW`
Required interfaces: -≺ `AckMessage`, -≺ `ApplicationActivation`, -≺ `DeviceInitCacheMgmt`, -≺ `DHUpdate`, -≺ `IncomingCommand`, -≺ `NotifyRecipient`, -≺ `OSToGWCommand`, -≺ `RouteDeviceData`, -≺ `SensorDataMgmt`, -≺ `TopologyMgmt`
Deployed on: OnlineServiceNode, CommunicationFailureNode
Visible on diagrams: figs. A.2 and B.2

E.1.55 OtherDataDB

Responsibility: The `OtherDataDB` is responsible for storing any data in the `SIoTIP system` aside from topology information, sensor and session data. For all users this includes account information such as name, credentials and contact information. For each subscription it includes a status, timeframe and which application. It also includes application instance info per application such as the configuration (end-user roles, topology assignment). The sent notifications and their status is also stored here. Any information about applications in the system is also stored (description, payment info, status, developer, default configuration, requirements) in the `OtherDataDB`. Lastly, hardware information such as the pairing between device models and types, conversion libraries and a listing of the various supported categories/models is also stored here.
Super-components: ▭ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `OtherDataMgmt`
Required interfaces: None
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2, D.4, D.7, D.12, D.20, D.21, D.22, D.23, D.24, D.25, D.26, D.32, D.35 and D.41

E.1.56 SensorDataDB

Responsibility: The `SensorDataDB` is responsible for actually storing the sensor data.
Super-components: ▭ `SIoTIP system` ▷ ▭ `OSSensorDataDB`
Sub-components: None
Provided interfaces: -○ `SensorDataMgmt`
Required interfaces: None
Deployed on: SensorDBNode
Visible on diagrams: figs. B.3, C.2 and D.11

E.1.57 SensorDataLoadBalancer

Responsibility: The `SensorDataLoadBalancer` is responsible for sending database queries (reads/writes) to the correct server on which part of the sensor data is located.
Super-components: ▭ `SIoTIP system` ▷ ▭ `OSSensorDataDB`
Sub-components: None
Provided interfaces: -○ `SensorDataMgmt`
Required interfaces: -≺ `DBRequestResults`, -≺ `SensorDataMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. B.3, C.2 and D.11

E.1.58 SensorDataScheduler

Responsibility: The `SensorDataScheduler` is responsible for handling all read/write requests to the `SensorDataDB`. It will monitor the completion times and execute them after scheduling them in an order based off the current modus of the database (normal or overloaded). In normal mode all requests are processed in a first-in-first-out order, in overload mode write requests are priotised over specific lookup queries which are in turn prioritised over broad lookup queries. Furthermore, while in overload mode, requests from critical application instances have priority over non-critical application instances. Write requests should be handled within 500msec, read requests from critical application instances within in 750msec and read requests from non-critical application instances within 1000msec.

Super-components: SIoTIP system ▷ OSSensorDataDB
Sub-components: None
Provided interfaces: SensorDataMgmt
Required interfaces: SensorDataMgmt
Deployed on: SensorDBNode
Visible on diagrams: figs. B.3, C.2 and D.11

E.1.59 SessionDB

Responsibility: The SessionDB is responsible for storing any data on open sessions of users.
Super-components: SIoTIP system
Sub-components: None
Provided interfaces: SessionMgmt
Required interfaces: None
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2 and C.2

E.1.60 SIoTIP system

Responsibility: The SIoTIP system component is the highest level supercomponent representing the system as a whole.
Super-components: None
Sub-components: ApplicationDeveloperDashboard, InfrastructureOwnerDashboard, SysAdminDashboard, Gateway, ApplicationHandler, CustomerOrganisationDashboard, InvoiceManager, MobileAppCommunication, NotificationHandler, ApplicationCommandRouter, ApplicationHeartbeatMon ApplicationInstanceInfoDB, ApplicationStarter, AppTester, AuthenticationHandler, DBRequestHandler, DeviceDataRouter, DeviceInitCache, OSSensorDataDB, OSWithGWCommunication, OtherDataDB, SessionDB, TopologyDB, TopologyLoadBalancer, EventManager, EventDataDB, ApplicationStorageMonitor, ViolationManager, ViolationDataDB, OSAppStateDB, OSKernel
Provided interfaces: ApplicationInfo, AppUsesAPI, ConfirmDelivery, ConsultNotifications, DeviceManagement, GWAppUsesAPI, Heartbeat, ManageDevices, MemoryMgmt, PnPSensorData, RcvMobileAppMsg, Registration, Subscription, UploadApplication, UserAuthentication
Required interfaces: Actuate, AppProvidesAPI, Config, DeliverNotification, GWAppProvidesAPI, MemoryMgmt, RebootGW, RequestPnPData, SendInvoice, SendMobileAppMsg
Deployed on: OnlineServiceNode, ApplicationExecutionNode, GatewayNode, TopologyNode, SensorDBNode, ApplicationFailureNode, CommunicationFailureNode, MobileAppCommunicationNode, AppTestingNode, DashboardsNode, AppStateDBNode
Visible on diagrams: fig. A.1

E.1.61 SysAdminClient

Responsibility: The SysAdminClient is external to the SIoTIP system and represents the client of a SIoTIP system administrator that communicates with the SIoTIP system.
Super-components: None
Sub-components: None
Provided interfaces: None
Required interfaces: ConsultNotifications, UserAuthentication
Deployed on: SysAdminNode
Visible on diagrams: figs. A.1 and C.1

E.1.62 SysAdminDashboard

Responsibility: The `SysAdminDashboard` provides the main interface to all `SIoTIP system` administrators, it is currently only used to consult notifications.
Super-components: ▣ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `ConsultNotifications`, -○ `UserAuthentication`
Required interfaces: -< `OtherDataMgmt`, -< `UserAuthentication`, -< `VerifySession`
Deployed on: DashboardsNode
Visible on diagrams: figs. A.1, A.2, C.1, C.2 and D.32

E.1.63 ThirdPartyInvoicingService

Responsibility: The `ThirdPartyInvoicingService` is external to the `SIoTIP system` and represents a service that allows SIoTIP to send invoices, for example Zoomit or a credit card company.
Super-components: None
Sub-components: None
Provided interfaces: -○ `SendInvoice`
Required interfaces: None
Deployed on: ThirdPartyInvoicingNode
Visible on diagrams: figs. A.1 and C.1

E.1.64 ThirdPartyNotificationDeliverySystem

Responsibility: The `ThirdPartyNotificationDeliverySystem` is external to the `SIoTIP system` and represents a system that allows SIoTIP to send notifications via email and SMS.
Super-components: None
Sub-components: None
Provided interfaces: -○ `DeliverNotification`
Required interfaces: -< `ConfirmDelivery`
Deployed on: ThirdPartyNotificationNode
Visible on diagrams: figs. A.1, C.1 and D.20

E.1.65 TopologyDB

Responsibility: Responsible for storing all data related to topologies, access rights and device configurations.
Super-components: ▣ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `TopologyMgmt`
Required interfaces: None
Deployed on: TopologyNode
Visible on diagrams: figs. A.2, C.2, D.6, D.10, D.21 and D.36

E.1.66 TopologyLoadBalancer

Responsibility: The `TopologyLoadBalancer` is responsible for sending queries to the `TopologyDB` to the correct server, i.e. the server on which the relevant data is stored.
Super-components: ▣ `SIoTIP system`
Sub-components: None
Provided interfaces: -○ `TopologyMgmt`
Required interfaces: -< `DBRequestResults`, -< `DeviceInitCacheMgmt`, -< `NotifyRecipient`, -< `TopologyMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2, D.6, D.10, D.21 and D.36

E.1.67 ViolationDataDB

Responsibility: This database is used to store violations for each applicationInstance. There is only one record for each application Instance.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ViolationDataMgmt`
Required interfaces: None
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2 and D.33

E.1.68 ViolationManager

Responsibility: The `ViolationManager` is responsible for storing past violations of application instances. If an application instance has committed more than 20 violations in the past 24h then the `ApplicationStarter` will be asked to suspend the application instance on the online service and gateways. It is also responsible for removing violations older than 24h.
Super-components: `SIoTIP system`
Sub-components: None
Provided interfaces: `ViolationProcessor`
Required interfaces: `InternalAppActivation`, `ViolationDataMgmt`
Deployed on: OnlineServiceNode
Visible on diagrams: figs. A.2, C.2, D.2, D.33, D.37, D.38 and D.43

E.2 Interfaces

E.2.1 AckMessage

Provided by: `AvailabilityMonitor`, `DataHandler`, `DeviceStatusMonitor`, `GWCommandRouter`, `Gateway`
Required by: `ApplicationCommandPropagator`, `DataHandler`, `DataReceiver`, `DeviceStatusMonitor`, `OSWithGWCommunication`
Operations:

- void ackMessage(int ackNr)
  - Effect: The caller will send an acknowledgement over to a communication component of the `Gateway`. This message carries an acknowledgement number which was also in the message which has to be acknowledged. This is to make sure the receiver knows which of its messages was acknowledged such that it can be thrown away.
  - Sequence Diagrams: figs. D.7, D.10, D.11 and D.28

Diagrams: figs. A.2, B.1 and B.2

E.2.2 Actuate

Provided by: `Mote`
Required by: `GWCommandRouter`, `Gateway`, `SIoTIP system`
Operations:

- boolean sendActuationCommand(string commandName)
  - Effect: Send an actuation command to the actuator. The actuator acknowledges execution of the command. Sending an incorrect actuation command (e.g. 'take picture' to a 'buzzer') has no effect.
  - Sequence Diagrams: figs. D.2 and D.18
- boolean sendAsyncActuationCommand(string commandName, int requestID)
  - Effect: Send an actuation command to the actuator (Callback). The actuator acknowledges receipt of the command. Sending an incorrect actuation command (e.g. 'take picture' to a 'buzzer') has no effect.
  - Sequence Diagrams: None

Diagrams: figs. A.1, A.2 and B.1

E.2.3 AMReactToFailure

Provided by: `AvailabilityMonitor`
Required by: `GWFailureHandler`
Operations:

- void reactToCommunicationFailure(bool failed)
  - Effect: The `AvailabilityMonitor` will start/stop sending messages to the Online System depending on the value of the failed parameter (false/true respectively) as this message signifies the state of the communication channel at that point.
  - Sequence Diagrams: fig. D.5
- void restart()
  - Effect: Causes the `AvailabilityMonitor` component to restart.
  - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.4 AppCalls

Provided by: `ApplicationCommandRouter`
Required by: `ApplicationContainer`, `ApplicationHandler`
Operations:

- void applicationCommand(**InterAppCommand** cmd)
  - Effect: The `ApplicationCommandRouter` routes the **InterAppCommand** to the application instance part.
  - Sequence Diagrams: figs. D.19 and D.38
- void applicationCommand(**MoteCommand** cmd)
  - Effect: The `ApplicationCommandRouter` routes the **MoteCommand** to the correct actuator or sensor.
  - Sequence Diagrams: figs. D.2, D.18, D.37 and D.39
- void applicationCommand(**DatabaseCommand** cmd)
  - Effect: The `ApplicationCommandRouter` routes a **DatabaseCommand** to the `DBRequestHandler` where it is handled
  - Sequence Diagrams: fig. D.43

Diagrams: figs. A.2 and B.4

E.2.5 AppContainerMgmt

Provided by: `ApplicationContainer`
Required by: `ApplicationManager`, `ApplicationScheduler`
Operations:

- boolean doRollback(**State** state, **Code** code)
  - Effect: Does a rollback of the application instance to the specified state. Returns true if it succeeds
  - Sequence Diagrams: fig. D.35
- void killContainer(**ApplicationInstanceID** appInID)
  - Effect: Forces a container (and its stand-in) to shut down, killing the application executing inside. This method is not responsible for making sure the application is killed in a controlled or safe manner, the caller of this method is responsible for making sure of that.
  - Sequence Diagrams: fig. D.26
- void startNewContainer(**ApplicationInstanceID** appInID, **Code** code, **ApplicationRequirements** appReq, **ConversionCode** conversionCode, **TopologyRequirementsInstantiation** topReqIn)
  - Effect: Starts a new container running the specified code. The provided application requirements will be used for validity checking of certain commands.
  - Sequence Diagrams: figs. D.1 and D.41

Diagrams: fig. B.4

E.2.6 AppFailure

Provided by: `ApplicationHeartbeatMonitor`, `GWApplicationHeartbeatMonitor`

Required by: `ApplicationCommandRouter`, `ApplicationContainer`, `ApplicationHandler`, `ApplicationManager`, `ApplicationScheduler`, `GWApplicationContainer`, `GWApplicationScheduler`
Operations:
- void applicationProcedureCrashed(**ApplicationInstanceID** appInID, **CrashInfo** crashInfo)
  - Effect: Informs the relevant scheduler that a procedure of a certain application instance threw an exception/error, implying a crash. The scheduler will inform the application itself of this by scheduling it to receive the info.
  - Sequence Diagrams: fig. D.3
- void containerRollbackFailed(**ApplicationInstanceID** AppInID)
  - Effect: Notifies the `ApplicationHeartbeatMonitor` that a container has crashed and the rollback attempt was unsuccessful. The `ApplicationStarter` should be called to suspend the `Application`.
  - Sequence Diagrams: fig. D.41
- void heartbeat()
  - Effect: This will cause the a reset of the time for the calling component. Used by the heartbeat monitors to check whether the requiring component is still available.
  - Sequence Diagrams: None

Diagrams: figs. A.2, B.1 and B.4

E.2.7 AppInstanceChange

Provided by: `ApplicationHandler`, `ApplicationManager`, `GWApplicationContainer`
Required by: `ApplicationStarter`, `GWCommandRouter`
Operations:
- boolean doRollback(**Code** code, **State** state, **ApplicationInstanceID** AppInID)
  - Effect: Perform a rollback of the given application instance with the given state and code. Return true if the rollback was successful
  - Sequence Diagrams: fig. D.35
- void finalizeApplicationUpdate(**ApplicationInstanceID** appInID)
  - Effect: Finalizes an application update. The caller must ensure that the environment is in a state where the update can be finalized. More specifically, this method assumes that the old version of the application is ready to shut down and its containers are already dormant. This method kills the old version's container, adds the entries that are necessary for the new version to the `ApplicationInstanceInfoDB`, activates the new version's container and calls its initialize procedure. The Online Service version of this method is also responsible for sending the message to the gateways to call the gateway version of this method.
  - Sequence Diagrams: None
- void finalizeApplicationUpdate(**ApplicationInstanceID** appInID, **State** state)
  - Effect: Finalizes an application update. The caller must ensure that the environment is in a state where the update can be finalized. More specifically, this method assumes that the old version of the application is ready to shut down and its containers are already dormant. This method kills the old version's container, adds the entries that are necessary for the new version to the `ApplicationInstanceInfoDB`, activates the new version's container and calls its initialize procedure. The Online Service version of this method is also responsible for sending the message to the gateways to call the gateway version of this method.
  - Sequence Diagrams: fig. D.25
- void killContainer(**ApplicationInstanceID** appInID)
  - Effect: The called container is deleted along with its contents.
  - Sequence Diagrams: figs. D.23 and D.26
- void startApplicationInstance(**ApplicationInstanceID** appInID, **Code** code, **ApplicationRequirements** appReq, **ConversionCode** conversionCode, boolean update, **TopologyRequirementsInstantiation** topReqIn)
  - Effect: Start a new container for an application or reactivate an existing one (decided by whether there exists a container with the given appInID). The conversionCode is passed along to be used by the container. The update parameter signifies whether routing data should be activated already, if an update of the app is going on, it should not. (First the old container has to be killed).
  - Sequence Diagrams: figs. D.1 and D.24
- void startShutDownProcedure(**ApplicationInstanceID** oldAppInID)

  - Effect: Initializes the shut down procedure for an application in the context of an application update. The application is warned of its impending shutdown, but it can still act and postpone its own shutdown to prevent unsafe or inconsistent situations. Eventually, calling this method should result in the application being shut down everywhere, both on the Online Service and on the Gateways.
  - Sequence Diagrams: fig. D.24
- void suspendApplicationInstance(**ApplicationInstanceID** appInID)
  - Effect: The `ApplicationManager` is asked to locate and suspend the application container corresponding to the given id.
  - Sequence Diagrams: fig. D.22

Diagrams: figs. A.2, B.1 and B.4

E.2.8 AppInstanceMapChange

Provided by: `ApplicationInstanceInfoDB`, `DataHandler`
Required by: `ApplicationCommandRouter`, `ApplicationHandler`, `ApplicationManager`, `ApplicationStarter`, `GWCommandRouter`
Operations:

- void addMapEntry(**ApplicationInstanceID** appInID, **DeviceResourceLocator** pRL)
  - Effect: Adds the specified pluggable device to the list of devices the specified application instance is interested in (and wishes to recieve data from).
  - Sequence Diagrams: figs. D.1 and D.26
- void removeMapEntry(**ApplicationInstanceID** appInID, **DeviceResourceLocator** pRL)
  - Effect: Removes the specified pluggable device to the list of devices the specified application instance is interested in (and wishes to recieve data from). Nothing happens if this device was not in the list.
  - Sequence Diagrams: fig. D.26

Diagrams: figs. A.2, B.1 and B.4

E.2.9 AppInstanceSuspension

Provided by: `ApplicationScheduler`, `GWApplicationScheduler`
Required by: `ApplicationManager`, `GWApplicationStorageMonitor`, `GWCommandRouter`
Operations:

- void activateApplicationInstance(**ApplicationInstanceID** appInID)
  - Effect: Activates the specified application instance, causing the initialize procedure to be called on the appropriate container.
  - Sequence Diagrams: None
- void activateApplicationInstance(**ApplicationInstanceID** appInID, **State** state)
  - Effect: Activates the specified application instance, causing the initialize procedure to be called on the appropriate container.
  - Sequence Diagrams: fig. D.25
- void deactivateApplicationInstance(**ApplicationInstanceID** appInID)
  - Effect: Initiate the shutdown of an application by sending Commands to gateways and prepareShutdown() calls to application instance's containers.
  - Sequence Diagrams: fig. D.23
- void periodicShutdownForInstance(**ApplicationInstanceID** appInID)
  - Effect: Used during the update of an application, the scheduler will start calling the shutdown procedure on an application until it is ready to be killed and make way for the updated version of the application.
  - Sequence Diagrams: fig. D.24
- void suspendApplicationInstance(**ApplicationInstanceID** appInID)
  - Effect: Marks the specified application instance as suspended, meaning no more scheduling will be done for the instance. The suspend call is done to the application via the appropriate container running the instance.
  - Sequence Diagrams: fig. D.22

Diagrams: figs. B.1 and B.4

E.2.10 ApplicationActivation

Provided by: `ApplicationStarter`
Required by: `ApplicationContainer`, `ApplicationDeveloperDashboard`, `ApplicationHandler`, `ApplicationManager`, `CustomerOrganisationDashboard`, `DeviceStatusMonitor`, `OSWithGWCommunication`
Operations:

- void UpdateApplicationInstances(**Code** code, string description, **ApplicationMetaData** metadata, list<**RoleDescription**> roles, list<**Version**> versions)
  - Effect: Starts the update procedure for all active instances of the application with a version that's in the 'versions' list.
  - Sequence Diagrams: fig. D.4
- void deactivate(**ApplicationInstanceID** appInID)
  - Effect: Starts the deactivation process of a certain subscription (appInstance). The `ApplicationStarter` will set the process of killing the old applications in motion.
  - Sequence Diagrams: fig. D.23
- void devicesStatusChange(list<**DeviceResourceLocator**> pRLs, boolean active)
  - Effect: Informs the `ApplicationStarter` of a device that underwent a change that may cause application instances to activate/deactivate. Checks will be started to see if any application instances need to be suspended/reactivated (checks are limited to the instances which have one of the specified devices in their configuration). A list is used as some events (such as mote failure or a topology change) may cause multiple pluggable devices to be changed at once.
  - Sequence Diagrams: figs. D.6 and D.7
- void endUserRoleChange(**UserID** cudtOrgID, **UserID** uID, string role, **ApplicationID** appID)
  - Effect: The specified end user for a customer organisation will have a new role assigned to them for a certain app.
  - Sequence Diagrams: None
- void gwDown(**GatewayID** gwID)
  - Effect: The `ApplicationStarter` is responsible for coordinating the update process of applications. In that context, it needs to know about any gateways becoming unavailable. An update procedure for an application needs to be aborted if a gateway goes down that's running a component of the application which needs to be updated, otherwise we might end up in an inconsistent situation where most of the system is running the new version but some gateways are still running the old version. This method is responsible for checking whether any updates must be aborted when a gateway goes down, and initializing that process if it needs to be done.
  - Sequence Diagrams: None
- void startApplication(**ApplicationInstanceID** appInID, **Code** code, **ApplicationRequirements** appReq, **TopologyRequirementsInstantiation** topReqIn)
  - Effect: Will start the process of starting the various parts of an application on the Online Service and the gateways (if applicable).
  - Sequence Diagrams: fig. D.21
- void tryRollback(**ApplicationInstanceID** id)
  - Effect: The application instance has crashed and should be rolled back to the most recent state
  - Sequence Diagrams: figs. D.3 and D.41

Diagrams: figs. A.2, B.2 and B.4

E.2.11 ApplicationInfo

Provided by: `ApplicationDeveloperDashboard`, `SIoTIP system`
Required by: `ApplicationDeveloperClient`
Operations:

- **HTML** getInfoOnApprovedApp(**SessionID** sID, **ApplicationID** appID)
  - Effect: Fetch detailed statistics on the specified application and will return these to the client.
  - Sequence Diagrams: None
- **HTML** getOverviewOfUploadedApplications(**SessionID** sID)
  - Effect: Provides the application provider with an overview of all their uploaded applications on their dashboard.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.12 ApplicationInstanceLookup

Provided by: `ApplicationInstanceInfoDB`
Required by: `DeviceDataRouter`
Operations:

- list<**ApplicationInstanceID**> getInstancesInterestedInDevice(**PluggableDeviceID** devID)
  - Effect: Return the list of **ApplicationInstanceID**'s that correspond to application instances that are currently interested in getting updates from a device.
  - Sequence Diagrams: fig. D.16

Diagrams: fig. A.2

E.2.13 AppNotifications

Provided by: `NotificationHandler`
Required by: `ApplicationStorageMonitor`
Operations:

- void sendNotification(**ContactInfo** Info, **UserID** ID)
  - Effect: This function enables the applications to send the notifications to System admin.
  - Sequence Diagrams: fig. D.32

Diagrams: fig. A.2

E.2.14 AppProvidesAPI

Provided by: `Application`
Required by: `ApplicationContainer`, `ApplicationHandler`, `SIoTIP system`
Operations:

- void handleCrash(**CrashInfo** error) throws *ApplicationUncaughtException*
  - Effect: When an application procedure crashes, its effects are not executed and the application should still be in a workable state. The application is notified of the fact that one of its procedures crashed by calling this procedure with information about the crash. This allows the application to respond to the event.
  - Sequence Diagrams: fig. D.3
- void initialize(**Code** code)
  - Effect: The application is started with the given code.
  - Sequence Diagrams: fig. D.1
- void initialize(**Code** code, **State** state)
  - Effect: The application is started with the given code.
  - Sequence Diagrams: None
- void pluggableDeviceFailed(**DeviceResourceLocator** pRL) throws *ApplicationUncaughtException*
  - Effect: Applications should be informed when a device they were using has crashed, this is done via a function call on the application.
  - Sequence Diagrams: fig. D.27
- boolean prepareShutdown() throws *ApplicationUncaughtException*
  - Effect: When this procedure is called, the system requests the application to prepare itself for an impending shutdown. The application can return a list of commands it wants to see executed before it shuts down, and a boolean indicating whether it is ready to shut down.
  - Sequence Diagrams: fig. D.9
- void rcvDataFromDB(**PluggableDeviceID** ID, **SystemSensorData** data)
  - Effect: The requested data from the DB will be recieved here.
  - Sequence Diagrams: fig. D.43
- void reactToCommand(**InterAppCommand** cmd) throws *ApplicationUncaughtException*
  - Effect: Pass the provided command to the application.
  - Sequence Diagrams: fig. D.15
- void reactToData(**PluggableDeviceID** pID, **SystemSensorData** data) throws *ApplicationUncaughtException*
  - Effect: Pass the provided sensor data to the application so that it can process it.
  - Sequence Diagrams: fig. D.31
- boolean restore(**State** state, **Code** code)
  - Effect: When an application had crashed this method is called to restore the application to the most recent saved state

  - Sequence Diagrams: fig. D.35
- void wakeUp(**Event** event) throws *ApplicationUncaughtException*
  - Effect: Wake up the application after a given event has occurred..
  - Sequence Diagrams: figs. D.30 and D.31

Diagrams: figs. A.1, A.2 and B.4

E.2.15 AppUsesAPI

Provided by: `ApplicationContainer`, `ApplicationHandler`, `SIoTIP system`
Required by: `Application`
Operations:

- **TopologyRequirementsInstantiation** getInstantiatedRequirements()
  - Effect: Returns the instantiatedTopologyRequirements.
  - Sequence Diagrams: None
- void saveState(**State** state)
  - Effect: The `Application` saves it's current state. The application can restore it's state for up to two weeks.
  - Sequence Diagrams: fig. D.34
- void scheduleEvent(**Event** event)
  - Effect: The `Application` schedules a future event. The event can either be a conditional or timed event.
  - Sequence Diagrams: fig. D.29
- void sendCommand(**InterAppCommand** cmd)
  - Effect: The `Application` sends an **InterAppCommand** to a different part of the application instance. This can be a gatewayApplication or a `MobileApp`.
  - Sequence Diagrams: figs. D.19, D.38 and D.40
- void sendCommand(**MoteCommand** cmd)
  - Effect: The `Application` sends a **MoteCommand** to the system. This can be either an **ActuationCommand** or a **ConfigurationCommand** to an actuator or sensor.
  - Sequence Diagrams: figs. D.2 and D.39
- void sendCommand(**DatabaseCommand** cmd)
  - Effect: The `Application` sends a DataBaseCommand to the `DBRequestHandler`. The command should be an SQL-query.
  - Sequence Diagrams: fig. D.43

Diagrams: figs. A.1 and B.4

E.2.16 CallThroughContainer

Provided by: `ApplicationContainer`, `GWApplicationContainer`
Required by: `ApplicationScheduler`, `GWApplicationScheduler`
Operations:

- void handleCrash(**CrashInfo** error) throws *ApplicationUncaughtException*
  - Effect: When an application procedure crashes, its effects are not executed and the application should still be in a workable state. The application is notified of the fact that one of its procedures crashed by calling this procedure with information about the crash. This allows the application to respond to the event.
  - Sequence Diagrams: fig. D.3
- void initialize(**Code** code)
  - Effect: The application starts to run with the code given to its container.
  - Sequence Diagrams: fig. D.1
- void initialize(**Code** code, **State** state)
  - Effect: The application starts to run with the code given to its container.
  - Sequence Diagrams: fig. D.25
- void pluggableDeviceFailed(**DeviceResourceLocator** pRL) throws *ApplicationUncaughtException*
  - Effect: See AppProvidesAPI::pluggableDeviceFailed
  - Sequence Diagrams: fig. D.27
- boolean prepareShutdown() throws *ApplicationUncaughtException*

    - Effect: When this procedure is called, the system requests the application to prepare itself for an impending shutdown. The application now able to call functions by iteself so it does not need to return the list of command.
    - Sequence Diagrams: fig. D.9
- void reactToCommand(**InterAppCommand** cmd)
    - Effect: An incoming **InterAppCommand** will be passed to application instance running on the online service.
    - Sequence Diagrams: figs. D.14 and D.15
- void reactToCommand(**MoteCommand** cmd)
    - Effect: An incoming **MoteCommand** will be passed to application instance running on the online service.
    - Sequence Diagrams: fig. D.37
- void reactToCommand(**DatabaseResultCommand** DBR)
    - Effect: An incoming DataBaseResultCommand will be passed to application instance running on the online service.
    - Sequence Diagrams: fig. D.43
- void reactToData(**PluggableDeviceID** pID, **SystemSensorData** data) throws *ApplicationUncaughtException*
    - Effect: Pass the provided sensor data to the application so that it can process it.
    - Sequence Diagrams: figs. D.31 and D.42
- void wakeUp(**Event** event) throws *ApplicationUncaughtException*
    - Effect: Pass an incoming event, originated from `EventManager`, to the container to wake up the application. The application will handle the event.
    - Sequence Diagrams: figs. D.30, D.31 and D.42

Diagrams: figs. B.1 and B.4

E.2.17 ChangeSyncTimer

Provided by: `DataReceiver`
Required by: `ApplicationCommandPropagator`, `DeviceStatusMonitor`
Operations:

- void checkGWSyncInterval()
    - Effect: Informs of a possible change of the expected synchronization interval for a gateway. This change may be needed due to an `Application` command changing the update frequency of a device, installation of a new device with a shorter synchronization period, or a failing of the previous device with the shortest synchronization period.
    - Sequence Diagrams: None

Diagrams: fig. B.2

E.2.18 CommHeartbeat

Provided by: `CommunicationHeartbeatMonitor`
Required by: `ApplicationCommandPropagator`, `DataReceiver`, `DeviceStatusMonitor`
Operations:

- void heartbeat()
    - Effect: This will cause the `CommunicationHeartbeatMonitor` to reset the time for the calling component. Used by the `CommunicationHeartbeatMonitor` to check whether the requiring component is still available.
    - Sequence Diagrams: None

Diagrams: fig. B.2

E.2.19 Config

Provided by: `Mote`
Required by: `GWCommandRouter`, `Gateway`, `SIoTIP system`
Operations:

- Map<string, string> getConfig()
    - Effect: Returns the current configuration of a PluggableDevice as a map.
    - Sequence Diagrams: None

- void setConfig(Map<string, string> config)
  - Effect: Changes to configuration of the PluggableDevice. Sending values to an incorrect PluggableDevice (e.g., ‘resolution’,‘640x480’ to the buzzer) has no effect.
  - Sequence Diagrams: None

Diagrams: figs. A.1, A.2 and B.1

E.2.20 ConfirmDelivery

Provided by: `NotificationHandler`, `SIoTIP system`
Required by: `ThirdPartyNotificationDeliverySystem`
Operations:

- void acknowledgeSMSDelivery(int messageId)
  - Effect: Causes the `NotificationHandler` to mark the SMS message with the matching messageId as "sent" in the `OtherDataDB`.
  - Sequence Diagrams: fig. D.20

Diagrams: figs. A.1 and A.2

E.2.21 ConsultNotifications

Provided by: `ApplicationDeveloperDashboard`, `CustomerOrganisationDashboard`, `InfrastructureOwnerDashb` `SIoTIP system`, `SysAdminDashboard`
Required by: `ApplicationDeveloperClient`, `CustomerOrganisationClient`, `InfrastructureOwnerClient`, `NotificationHandler`, `SysAdminClient`
Operations:

- void AppViolationNotification(**ContactInfo** info, **UserID** ID)
  - Effect: The notifications from the application instance part can be consulted in this function.
  - Sequence Diagrams: fig. D.32
- **HTML** displayNotificationsForUser(**SessionID** sID)
  - Effect: The dashboard will fetch the notifications for the user with the **UserID** contained in the **SessionID** from the `OtherDataDB`. It will ask the front-end to display these.
  - Sequence Diagrams: None
- **HTML** giveNotificationDetails(**SessionID** sID, **NotificationID** nID)
  - Effect: The dashboard will fetch the details of the identified notification from the `OtherDataDB` and will ask the front-end to display these.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.22 CRReactToFailure

Provided by: `GWCommandRouter`
Required by: `GWFailureHandler`
Operations:

- void reactToCommunicationFailure(bool failed)
  - Effect: The component will start/stop sending messages to the Online System depending on the value of the failed parameter (false/true respectively) as this message signifies the state of the communication channel at that point.
  - Sequence Diagrams: fig. D.5
- void restart()
  - Effect: Causes the `GWCommandRouter` component to restart.
  - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.23 DBRequestResults

Provided by: `DBRequestHandler`
Required by: `OSSensorDataDB`, `SensorDataLoadBalancer`, `TopologyLoadBalancer`
Operations:

- void queryResults(**DatabaseResultCommand** cmd)

  - Effect: The `DBRequestHandler` will schedule the results of a certain query to be sent to the application instance that requested them.
  - Sequence Diagrams: fig. D.43

Diagrams: figs. A.2 and B.3

E.2.24 DeliverMsgMobileApp

Provided by: `MobileAppCommunication`
Required by: `ApplicationCommandRouter`
Operations:

- void sendToMobileApp(string hostname, int port, string message)
  - Effect: A message is sent to the specified hostname/port combination.
  - Sequence Diagrams: fig. D.13

Diagrams: fig. A.2

E.2.25 DeliverNotification

Provided by: `ThirdPartyNotificationDeliverySystem`
Required by: `NotificationHandler`, `SIoTIP system`
Operations:

- void sendEmail(string emailaddress, string message)
  - Effect: Will cause the third-party messaging system to send an email with the specified message to the specified email address.
  - Sequence Diagrams: fig. D.20
- void sendSMS(int phonenumber, string message, int messageid)
  - Effect: Will cause the third-party messaging system to send an sms with the specified message to the specified number.
  - Sequence Diagrams: fig. D.20

Diagrams: figs. A.1 and A.2

E.2.26 DeviceAvailabilityCheck

Provided by: `DeviceStatusMonitor`, `OSWithGWCommunication`
Required by: `ApplicationCommandRouter`, `ApplicationStarter`
Operations:

- boolean isAnyOfTheseDevicesAvailable(list<**DeviceResourceLocator**> devices)
  - Effect: Returns true if any of the specified devices is marked as available in the `DeviceStatusMonitor` and false otherwise.
  - Sequence Diagrams: fig. D.7
- boolean isDeviceAvailable(**DeviceResourceLocator** pRL)
  - Effect: Returns true if the specified device is marked as available in the `DeviceStatusMonitor` and false otherwise.
  - Sequence Diagrams: fig. D.18

Diagrams: figs. A.2 and B.2

E.2.27 DeviceAvailabilityMessage

Provided by: `DeviceStatusMonitor`, `OSWithGWCommunication`
Required by: `AvailabilityMonitor`, `Gateway`
Operations:

- void moteAvailable(**MoteInfo** moteInfo, int ackNr)
  - Effect: Message will be propagated to the `TopologyDB` together with the ID of the gateway this message came from. The returned list of pluggable devices will then be propagated to the `ApplicationStarter` and a notification will be sent to the owner of the infrastructure this mote belongs.
  - Sequence Diagrams: None
- void moteUnavailable(**MoteInfo** moteInfo, int ackNr)

- Effect: Message will be propagated to the `TopologyDB` together with the ID of the gateway this message came from. The returned list of pluggable devices will then be propagated to the `ApplicationStarter` and a notification will be sent to the infrastructure owner this mote belongs to.
  - Sequence Diagrams: None
- void newMote(**MoteInfo** moteInfo, int ackNr)
  - Effect: The message (along with information about the gateway/topology this message came from) will be propagated to the `TopologyDB`.
  - Sequence Diagrams: None
- void newPluggableDeviceInMote(int moteID, **PluggableDeviceID** pID, **PluggableDeviceType** pType, int ackNr, **GatewayID** gwID)
  - Effect: Message will be propagated to the `TopologyDB` together with the ID of the gateway this message came from.
  - Sequence Diagrams: fig. D.10
- void pluggableDeviceAvailable(**PluggableDeviceID** pID, **MoteID** mID, int ackNr)
  - Effect: Informs the `DeviceStatusMonitor` that a pluggable device on a certain mote has become available. This will be marked as such and then propagated to other components (such as the `ApplicationStarter`) which might be interested in this information.
  - Sequence Diagrams: None
- void pluggableDeviceUnavailable(**PluggableDeviceID** pID, **MoteID** mID, int ackNr)
  - Effect: Informs the `DeviceStatusMonitor` that a pluggable device on a certain mote has become unavailable. This will be marked as such and then propagated to other components (such as the `ApplicationStarter`) which might be interested in this information.
  - Sequence Diagrams: fig. D.7

Diagrams: figs. A.2, B.1 and B.2

E.2.28 DeviceInitCacheMgmt

Provided by: `DeviceInitCache`
Required by: `DataReceiver`, `OSWithGWCommunication`, `TopologyLoadBalancer`
Operations:

- boolean isDeviceInitialized(**DeviceResourceLocator** pRL)
  - Effect: The `DeviceInitCache` returns whether the given device is marked "initialized".
  - Sequence Diagrams: fig. D.11
- void setDeviceInitialized(**DeviceResourceLocator** pRL)
  - Effect: The `DeviceInitCache` marks the given device as "initialized"
  - Sequence Diagrams: None

Diagrams: figs. A.2 and B.2

E.2.29 DeviceManagement

Provided by: `AvailabilityMonitor`, `Gateway`, `SIoTIP system`
Required by: `Mote`
Operations:

- void pluggableDevicePluggedIn(**MoteInfo** mInfo, **PluggableDeviceID** pID)
  - Effect: Notify the gateway that a new PluggableDevice is connected.
  - Sequence Diagrams: fig. D.10
- void pluggableDeviceRemoved(**PluggableDeviceID** pID)
  - Effect: Notify the gateway that a PluggableDevice is removed.
  - Sequence Diagrams: None

Diagrams: figs. A.1, A.2 and B.1

E.2.30 DHReactToFailure

Provided by: `DataHandler`
Required by: `GWFailureHandler`
Operations:

- void reactToCommunicationFailure(bool failed)

  - Effect: The `DataHandler` will either start storing all messages for the gateway in the `GWTempDataDB` or stop doing so and clean it out depending on the value of the boolean.
  - Sequence Diagrams: fig. D.5
- void restart()
  - Effect: Causes the `DataHandler` component to restart.
  - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.31 DHUpdate

Provided by: `DataHandler`, `Gateway`
Required by: `DeviceStatusMonitor`, `OSWithGWCommunication`
Operations:

- void initializeAdapter(**Code** adapter)
  - Effect: Initialize a manufacturer specific adapter with the given code
  - Sequence Diagrams: fig. D.36

Diagrams: figs. B.1 and B.2

E.2.32 EventMgmt

Provided by: `EventDataDB`, `GWEventDataDB`
Required by: `EventManager`, `GWEventManager`
Operations:

- list<**Event**> getEventsForApplication(**ApplicationInstanceID** appID)
  - Effect: Returns the list of all events associated with a specific application ID.
  - Sequence Diagrams: None
- void saveEvent(**ApplicationInstanceID** appID, **Event** event)
  - Effect: The event will be saved with the given application ID.
  - Sequence Diagrams: fig. D.29

Diagrams: figs. A.2 and B.1

E.2.33 Failure

Provided by: `ApplicationHandler`, `ApplicationManager`
Required by: `ApplicationHeartbeatMonitor`
Operations:

- void containerFailed(**ApplicationInstanceID** appInID)
  - Effect: Initiate the start of a new `ApplicationContainer` for the application instance associated with the crashed container.
  - Sequence Diagrams: fig. D.41
- void serverFailed(list<**ApplicationInstanceID**> appInIDs)
  - Effect: Restart the crashed server and all active application instances.
  - Sequence Diagrams: None

Diagrams: figs. A.2 and B.4

E.2.34 FailureDetection

Provided by: `GWFailureHandler`
Required by: `AvailabilityMonitor`, `DataHandler`, `GWCommandRouter`
Operations:

- void internalHeartbeat()
  - Effect: The `GWFailureHandler` will use the heartbeats it recieves to check whether the communication components are still available.
  - Sequence Diagrams: None
- void noAckRecieved()
  - Effect: The `GWFailureHandler` will assume there is something wrong with either a communication component on the Online Service or the communication channel and will order the other Communication components to store their messages in the `GWTempDataDB`.

    - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.35 GWAppCalls

Provided by: `GWCommandRouter`
Required by: `GWApplicationContainer`, `GWApplicationStorageMonitor`
Operations:

- void applicationCommand(**MoteCommand** cmd)
    - Effect: The GWApplicationCommandRouter routes the **InterAppCommand** to the application instance part.
    - Sequence Diagrams: fig. D.14
- void applicationCommand(**InterAppCommand** cmd)
    - Effect: The GWApplicationCommandRouter routes the **MoteCommand** to the correct actuator or sensor.
    - Sequence Diagrams: None
- void applicationCommand(**DatabaseCommand** cmd)
    - Effect: The GWApplicationCommandRouter routes a **DatabaseCommand** to the `DBRequestHandler` where it is handled
    - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.36 GWAppChange

Provided by: `ApplicationCommandRouter`
Required by: `ApplicationStarter`
Operations:

- void sendGWAppChangeCommand(**UpdateCommand** cmd)
    - Effect: Sends a command to a gateway informing the application execution system that an application needs to be suspended/updated/started/needs to start shutdown of old container, along with the data required to do so (e.g. **ApplicationInstanceID**, **ApplicationRequirements**, **ConversionCode** and **Code** for starting an application). The update parameter is also passed here (see AppInstanceChanges' startApplicationContainer function).
    - Sequence Diagrams: figs. D.1, D.22, D.23, D.24, D.25 and D.26

Diagrams: fig. A.2

E.2.37 GWAppProvidesAPI

Provided by: `GWApplication`
Required by: `GWApplicationContainer`, `Gateway`, `SIoTIP system`
Operations:

- void handleCrash(**CrashInfo** error) throws *ApplicationUncaughtException*
    - Effect: When an application procedure crashes, its effects are not executed and the application should still be in a workable state. The application is notified of the fact that one of its procedures crashed by calling this procedure with information about the crash. This allows the application to respond to the event.
    - Sequence Diagrams: None
- void initialize(**Code** code)
    - Effect: The application is started with the given code.
    - Sequence Diagrams: None
- void pluggableDeviceFailed(**DeviceResourceLocator** pRL) throws *ApplicationUncaughtException*
    - Effect: Applications should be informed when a device they were using has crashed, this is done via a function call on the application.
    - Sequence Diagrams: None
- void prepareShutdown() throws *ApplicationUncaughtException*
    - Effect: When this procedure is called, the system requests the application to prepare itself for an impending shutdown. The application can return a list of commands it wants to see executed before it shuts down, and a boolean indicating whether it is ready to shut down.

  - Sequence Diagrams: None
- void reactToCommand(**InterAppCommand** cmd) throws *ApplicationUncaughtException*
  - Effect: Pass the provided command to the application.
  - Sequence Diagrams: fig. D.14
- void reactToData(**PluggableDeviceID** pID, **SystemSensorData** data) throws *ApplicationUncaughtException*
  - Effect: Pass the provided sensor data to the application so that it can process it.
  - Sequence Diagrams: fig. D.42
- void wakeUp(**Event** event) throws *ApplicationUncaughtException*
  - Effect: Wake up the application after a given event has occurred..
  - Sequence Diagrams: fig. D.42

Diagrams: figs. A.1, A.2 and B.1

E.2.38 GWAppUsesAPI

Provided by: `GWApplicationContainer`, `Gateway`, `SIoTIP system`
Required by: `GWApplication`
Operations:
- void saveState(**State** state)
  - Effect: The `GWApplication` saves it's current state. The application can restore it's state for up to two weeks.
  - Sequence Diagrams: None
- void scheduleEvent(**Event** event)
  - Effect: The `GWApplication` schedules a future event. The event can either be a conditional or timed event.
  - Sequence Diagrams: None
- void sendCommand(**InterAppCommand** cmd)
  - Effect: The `GWApplication` sends an **InterAppCommand** to a different part of the application instance. This can be an `Application` or a `MobileApp`.
  - Sequence Diagrams: None
- void sendCommand(**MoteCommand** cmd)
  - Effect: The `GWApplication` sends a **MoteCommand** to the system. This can be either an **ActuationCommand** or a **ConfigurationCommand** to an actuator or sensor.
  - Sequence Diagrams: None
- void sendCommand(**DatabaseCommand** cmd)
  - Effect: The `GWApplication` sends a DataBaseCommand to the `DBRequestHandler`. The command should be an SQL-query.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and B.1

E.2.39 GWAvailability

Provided by: `DeviceStatusMonitor`
Required by: `DataReceiver`
Operations:
- void gatewayFailed(**GatewayID** gwID)
  - Effect: Register the fact that a `Gateway` has failed in the `DeviceStatusMonitor`. The `DeviceStatusMonitor` will inform all interested parties of this event.
  - Sequence Diagrams: fig. D.6
- void registerNewGateway(**GatewayID** gatewayID, **IPAddress** ipAddress)
  - Effect: The `DeviceStatusMonitor` will register the new gateway and its IP address internally.
  - Sequence Diagrams: None

Diagrams: fig. B.2

E.2.40 GWTempDataMgmgt

Provided by: `GWTempDataDB`
Required by: `DataHandler`, `GWCommandRouter`
Operations:

- void addCommand(**DatabaseCommand** cmd)
  - Effect: Temporarily add a message/command to the gateway's storage. (This should only occur in case of a communication channel failure or failure of a communication component on the OS).
  - Sequence Diagrams: None
- void addData(**DeviceData** data, **PluggableDeviceID** pID)
  - Effect: Temporarily add a sensor reading to the gateway's storage. (This should only occur in case of a communication channel failure or failure of a communication component on the OS).
  - Sequence Diagrams: None
- list<**DatabaseCommand**> getCommand()
  - Effect: Return a list of all messages/commands stored in the `GWTempDataDB` and delete them from the `GWTempDataDB`.
  - Sequence Diagrams: None
- list<Tuple<**PluggableDeviceID**, **DeviceData**>> getData()
  - Effect: Return a list of all sensor data stored in the `GWTempDataDB` and delete them from the `GWTempDataDB`.
  - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.41 GWToOSCommand

Provided by: `ApplicationCommandPropagator`, `OSWithGWCommunication`
Required by: `GWCommandRouter`, `Gateway`
Operations:
- void routeCommandInGW(**InterAppCommand** cmd, int ackNr)
  - Effect: The `ApplicationCommandPropagator` will propagate the command to the `ApplicationCommandRouter` and will then send an acknowledgement back to the sender.
  - Sequence Diagrams: fig. D.28

Diagrams: figs. A.2, B.1 and B.2

E.2.42 HandleDBRequest

Provided by: `DBRequestHandler`
Required by: `ApplicationCommandRouter`
Operations:
- void handleDBRequest(**DatabaseCommand** cmd)
  - Effect: Process the database request command.
  - Sequence Diagrams: None

Diagrams: fig. A.2

E.2.43 Heartbeat

Provided by: `AvailabilityMonitor`, `Gateway`, `SIoTIP system`
Required by: `Mote`
Operations:
- void heartbeat(**MoteInfo** moteinfo, List<Tuple<**PluggableDeviceID**, **PluggableDeviceType**>> pds)
  - Effect: Periodic heartbeat from the mote to the gateway, including a list of the PluggableDevices and their DeviceTypes (i.e. those currently plugged into the mote)
  - Sequence Diagrams: fig. D.7

Diagrams: figs. A.1, A.2 and B.1

E.2.44 IncomingCommand

Provided by: `ApplicationCommandRouter`
Required by: `ApplicationCommandPropagator`, `MobileAppCommunication`, `OSWithGWCommunication`
Operations:
- void routeIncomingCommand(**InterAppCommand** cmd) throws *DestinationNotAvailableException*

  - Effect: The `ApplicationCommandRouter` determines which application instance or other component a command is intended for (from the **Command** information), and makes sure the scheduler schedules the application's reactToCommand procedure or the other component receives the correct call.
  - Sequence Diagrams: fig. D.28
- void routeIncomingCommand(**MoteCommand** mc)
  - Effect: The `ApplicationCommandRouter` determines which application instance or other component a command is intended for (from the **Command** information), and makes sure the scheduler schedules the application's reactToCommand procedure or the other component receives the correct call.
  - Sequence Diagrams: None
- void routeIncomingCommand(**DatabaseResultCommand** DBR)
  - Effect: The `ApplicationCommandRouter` determines which application instance or other component a command is intended for (from the **Command** information), and makes sure the scheduler schedules the application's reactToCommand procedure or the other component receives the correct call.
  - Sequence Diagrams: None

Diagrams: figs. A.2 and B.2

E.2.45 InternalAppActivation

Provided by: `ApplicationStarter`
Required by: `ApplicationHeartbeatMonitor`, `ApplicationStorageMonitor`, `ViolationManager`
Operations:

- void suspendApplication(**ApplicationInstanceID** appInID)
  - Effect: Informs the `ApplicationStarter` that an application has crashed on a critical procedure and that suspension has to be initiated.
  - Sequence Diagrams: figs. D.3, D.32, D.33 and D.41

Diagrams: fig. A.2

E.2.46 IPLookup

Provided by: `DeviceStatusMonitor`
Required by: `ApplicationCommandPropagator`
Operations:

- **IPAddress** getGatewayIPAddress(**GatewayID** gwID)
  - Effect: The `DeviceStatusMonitor` returns the IP address associated with the specified gateway.
  - Sequence Diagrams: figs. D.2, D.17 and D.22

Diagrams: fig. B.2

E.2.47 ManageDevices

Provided by: `InfrastructureOwnerDashboard`, `SIoTIP system`
Required by: `InfrastructureOwnerClient`
Operations:

- **HTML** changeTopologyTo(**SessionID** sID, **Topology** newTopology)
  - Effect: Will ask the front end to display the new topology.
  - Sequence Diagrams: None
- **HTML** doneChangingTopology()
  - Effect: Will propagate the last topology submitted by this user for storage in the `TopologyDB`. If there are still unplaced motes and/or pluggable devices, the front end will be asked to inform the user of this.
  - Sequence Diagrams: None
- **HTML** doneConsultingDeviceOverview(**SessionID** sID)
  - Effect: The `InfrastructureOwnerDashboard` will ask the front end to display the **Topology** overview again.
  - Sequence Diagrams: None
- **HTML** getDetailedInfoAboutDevice(**PluggableDeviceID** pID)

  - Effect: The `InfrastructureOwnerDashboard` will get information about the status of the selected gateway/mote/pluggable device from the `OtherDataDB`.
  - Sequence Diagrams: None
- **HTML** grantAccess(**SessionID** sID, **UserID** custOrgID)
  - Effect: Forward the access right (customer organisation gains access to a specific device) to the `TopologyDB`.
  - Sequence Diagrams: None
- **HTML** wantToConfigureAccessRights(**SessionID** sID)
  - Effect: The `InfrastructureOwnerDashboard` will fetch the list of devices associated with the user (according to the **UserID**) and will ask the front end to present this list.
  - Sequence Diagrams: None
- **HTML** wantToConfigureAccessRightsOfDevice(**SessionID** sID, **PluggableDeviceID** pID)
  - Effect: The `InfrastructureOwnerDashboard` will fetch the list of customer organisations associated with the specified device, as well as their current access rights (based on a request to the `TopologyDB` about which of those currently have access). It will then ask the front-end to display this information.
  - Sequence Diagrams: None
- **HTML** wantToConfigureTopology(**SessionID** sID)
  - Effect: The `InfrastructureOwnerDashboard` will fetch the right (corresponding to the current user) topology from the `TopologyDB` and will ask the front end to display this information.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.48 MemoryMgmt

Provided by: `ApplicationStorageMonitor`, `GWApplicationStorageMonitor`, `SIoTIP system`
Required by: `ApplicationContainer`, `ApplicationHandler`, `ApplicationManager`, `GWApplicationContainer`, `SIoTIP system`
Operations:
- void saveProcess(**ApplicationInstanceID** appID, **MaxMemoryAllowed** memory, **ProcessID** process)
  - Effect: This function will save the given values as an event in the `ApplicationStorageMonitor`. The event will trigger if the application uses more memory then allowed.
  - Sequence Diagrams: figs. D.1 and D.41

Diagrams: figs. A.2, B.1 and B.4

E.2.49 NotifyRecipient

Provided by: `NotificationHandler`
Required by: `ApplicationDeveloperDashboard`, `ApplicationHeartbeatMonitor`, `ApplicationStarter`, `DeviceStatusMonitor`, `OSWithGWCommunication`, `TopologyLoadBalancer`
Operations:
- void notify(**NotificationTrigger** trigger)
  - Effect: Places a notification as ‘unread’ in the inbox of the recipient, and may send out an SMS or e-mail alert depending on the recipient’s preferences. The notification configurations will be looked up based off the information in the **NotificationTrigger**.
  - Sequence Diagrams: figs. D.4, D.6, D.7, D.10 and D.22

Diagrams: figs. A.2 and B.2

E.2.50 OSToGWCommand

Provided by: `GWCommandRouter`, `Gateway`
Required by: `ApplicationCommandPropagator`, `OSWithGWCommunication`
Operations:
- void sendCommand(**MoteCommand** cmd)
  - Effect: Sending a command (actuation) to a gateway. The command will be routed to either the scheduler or a mote. If it’s a reconfiguration command, a new command directed to the sender of this command is created with the info that the reconfiguration worked.
  - Sequence Diagrams: figs. D.2, D.17 and D.18

- void sendCommand(**UpdateCommand** cmd)
  - Effect: Sending a command (SoftwareUpdateCommand) to a gateway. The command will be routed to either the scheduler .
  - Sequence Diagrams: fig. D.22
- void sendCommand(**InterAppCommand** cmd)
  - Effect: Sending a command (**InterAppCommand**) to a gateway. The command will be routed to the application instance running on the gateway .
  - Sequence Diagrams: fig. D.14

Diagrams: figs. A.2, B.1 and B.2

E.2.51 OtherDataMgmt

Provided by: `OtherDataDB`
Required by: `ApplicationCommandRouter`, `ApplicationDeveloperDashboard`, `ApplicationHandler`, `ApplicationManager`, `ApplicationStarter`, `ApplicationStorageMonitor`, `AuthenticationHandler`, `CustomerOrganisationDashboard`, `InvoiceManager`, `NotificationHandler`, `SysAdminDashboard`
Operations:

- **ApplicationID** addApplication(**UserID** appDevID, **Code** code, string description, **Version** version, **ApplicationMetaData** metadata)
  - Effect: Add the new application to the store by putting it in the database and marking it as having been tested.
  - Sequence Diagrams: None
- void addContactInfo(**UserID** uID, **ContactInfo** contactInfo)
  - Effect: Associate the contactInfo to the user identified by userID.
  - Sequence Diagrams: fig. D.12
- **UserID** addCustomerOrganisation(string userName, string password, **ContactInfo** contactInfo, **PaymentInfo** paymentInfo)
  - Effect: Add the information of the new customer organisation to the database, generates a new **UserID** and associate the new organisation with that id. The generated **UserID** is returned.
  - Sequence Diagrams: None
- void addDeviceLogForDevice(**Timestamp** ts, **DeviceResourceLocator** pRL, string description)
  - Effect: Add a log entry for a device describing an availability event (e.g. failed) for a specific pluggable device.
  - Sequence Diagrams: None
- void addDeviceLogForGateway(**Timestamp** ts, **GatewayID** gwID, string description)
  - Effect: Add a log entry for a device describing an availability event (e.g. failed) for a specific gateway.
  - Sequence Diagrams: None
- void addDeviceLogForMote(**Timestamp** ts, **MoteID** moteID, string description)
  - Effect: Adds a log entry for a device describing an availability event (e.g. failed) for a specific mote.
  - Sequence Diagrams: None
- boolean addEndUser(**UserID** custOrgID, string userName, **ContactInfo** contactInfo) throws *UserAlreadyExistsException*
  - Effect: Store the provided information as a new end user for the identified customer organisation. Return true if successful, false otherwise.
  - Sequence Diagrams: None
- **NotificationID** addNotification(**UserID** uID, string msg, **Timestamp** ts, **NotificationTrigger** trigger, **NotificationStatus** status, **CommunicationChannel** channel) throws *NoSuchUserException*
  - Effect: Adds a notification to the database with the given info for the specified user with uID.
  - Sequence Diagrams: fig. D.20
- void addSubscription(**ApplicationID** appID, **UserID** custOrgID, **Version** appVersion, list<Tuple<**UserID**, string>> userRoles, **TopologyRequirementsInstantiation** config, **Timestamp** timestamp)
  - Effect: Store the Customer Organisations' subscription and its associated data.
  - Sequence Diagrams: fig. D.21
- boolean checkEmailAlreadyInUseByOrg(string email)
  - Effect: Return true if the email address is already registered for a Customer Organisation in the database, and false otherwise.

- Sequence Diagrams: None
- boolean checkSubscription(**UserID** custOrgID, **ApplicationID** appID)
  - Effect: Return true if there is already a subscription of a Customer Organisation for an application (no matter which version)
  - Sequence Diagrams: fig. D.21
- **Version** checkSubscriptionVersion(**UserID** custOrgID, **ApplicationID** appID) throws *NoSuchSubscription*
  - Effect: Return the version of the current subscription for a Customer Organisation for an application. If there is no such subscription, an error is thrown.
  - Sequence Diagrams: None
- boolean checkUserNameExists(string userName)
  - Effect: Return true if there already exists a Customer Organisation using the given string as username, false otherwise.
  - Sequence Diagrams: None
- **ApplicationStatistics** detailedStatisticsForApplication(**ApplicationID** appID)
  - Effect: Return statistics related to a certain application, e.g. number of subscriptions.
  - Sequence Diagrams: None
- list<Tuple<**Timestamp**, string>> getAllLogsForDevice(**DeviceResourceLocator** pRL)
  - Effect: Return all availability log entries for the specified pluggable device.
  - Sequence Diagrams: None
- list<Tuple<**Timestamp**, string>> getAllLogsForGateway(**GatewayID** gwID)
  - Effect: Return all availability log entries for the specified `Gateway`.
  - Sequence Diagrams: None
- list<Tuple<**Timestamp**, string>> getAllLogsForMote(**MoteID** mID)
  - Effect: Return all availability log entries for the specified mote.
  - Sequence Diagrams: None
- Tuple<**UserID**, **Code**, string, **Version**, **ApplicationMetaData**> getApplicationInfo(**ApplicationID** appID) throws *NoSuchApplication*
  - Effect: Return info on the specified application.
  - Sequence Diagrams: figs. D.35 and D.41
- **ApplicationRequirements** getApplicationRequirements(**ApplicationID** appID, **Version** version)
  - Effect: Return the application requirements for a certain version of an application.
  - Sequence Diagrams: figs. D.21 and D.41
- list<**ApplicationMetaData**, int, string> getAppsForAppDeveloper(**UserID** uID) throws *NoSuchUserException*
  - Effect: Return the information on all of the applications of a certain application developer, along with the amount of subscribers and a possible note for each of those applications.
  - Sequence Diagrams: fig. D.4
- list<**ApplicationMetaData**> getAppsNotSubscribedOrNewVersionAvailable(**UserID** custOrgID)
  - Effect: Return the **ApplicationMetaData** of all application for which the given user does not have an active subscription or for which the user has a subscription to an older version.
  - Sequence Diagrams: fig. D.21
- list<**ApplicationMetaData**> getAppsSubscribed(**UserID** custOrgID) throws *NoSuchUserException*
  - Effect: Return the **ApplicationMetaData** of all applications for which the given user has an active subscription.
  - Sequence Diagrams: None
- Tuple<**ContactInfo**, **UserID**> getContactInfoForAppInstancesOwner(**ApplicationInstanceID** appInID) throws *NoSuchApplicationInstance*
  - Effect: Return the **ContactInfo** and **UserID** associated with the Customer Organisation which owns the given application instance.
  - Sequence Diagrams: figs. D.20 and D.32
- Tuple<**ContactInfo**, **UserID**> getContactInfoForInfrastructuresOwner(**InfrastructureID** infID) throws *NoSuchInfrastructure*
  - Effect: Return the **ContactInfo** and **UserID** associated with the owner of the given infrastructure.
  - Sequence Diagrams: fig. D.20
- **ContactInfo** getContactInfoForUser(**UserID** uID) throws *NoSuchUserException*
  - Effect: Return the **ContactInfo** associated with the given user.
  - Sequence Diagrams: fig. D.20
- Tuple<**Timestamp**, **TopologyRequirementsInstantiation**, list<Tuple<**UserID**, string>>> getCurrentSubscriptionInfo(**ApplicationInstanceID** appInID) throws *NoSuchApplicationInstance*

- Effect: Get the subscription info from a currently active subscription with the given id.
- Sequence Diagrams: None

- list<Tuple<**UserID**, string, **ContactInfo**>> getCustOrgEndUsers(**UserID** CustOrgID) throws *NoSuchUserException*
  - Effect: Return a list of tuples containing information on the various end users of a Customer Organisation.
  - Sequence Diagrams: fig. D.21
- list<**UserID**> getCustomerOrganisationsAssociatedWithInfrastructureOwner() throws *NoSuchUserException*
  - Effect: Return all Customer Organisations who are associated with the specified Infrastructure Owner, i.e. the organisations have an application running in the infrastructure of the owner.
  - Sequence Diagrams: None
- void getDeviceSpecifications(**PluggableDeviceType** type) throws *NoSuchPluggableDeviceType*
  - Effect: Return the Specifications for the specified device type. This includes information on the category of the device, the limitations (e.g. max sampling frequency).
  - Sequence Diagrams: None
- list<**GatewayID**> getGatewaysOnWhichApplicationInstanceRuns(**ApplicationInstanceID** appInID)
  - Effect: Returns all the gateways on which, according to the subscriptions **TopologyRequirementsInstantiation**, an application instance of the subscription runs.
  - Sequence Diagrams: figs. D.22 and D.23
- list<**ApplicationInstanceID**> getInstancesOfAppInVersions(**ApplicationID** appID, list<**Version**> versions)
  - Effect: Return the application identifiers of all application instances of a certain application for one of several versions.
  - Sequence Diagrams: fig. D.24
- list<**NotificationID**, string> getNotificationsForUser(**UserID** uID) throws *NoSuchUserException*
  - Effect: Returns the IDs and descriptions of all the Notifications that belong to the user with id uID.
  - Sequence Diagrams: None
- list<Tuple<**TopologyRequirementsInstantiation**, **ApplicationInstanceID**>> getTopologyRequirementsInstantiationForApplicationsWhichUseDevice(**DeviceResourceLocator** pRL)
  - Effect: Return the **TopologyRequirementsInstantiation** and **ApplicationInstanceID** for all the application instances whose current configuration includes the device with id pID.
  - Sequence Diagrams: fig. D.7
- Tuple<string, **Timestamp**, **NotificationTrigger**, **NotificationStatus**, **CommunicationChannel**, **NotificationStatus**> readNotification(**NotificationID** nID)
  - Effect: Return information on the specified notification and sets the status as read.
  - Sequence Diagrams: None
- list<**ApplicationID**> removeEndUser(**UserID** custOrgID, **UserID** uID) throws *NoSuchUserException*
  - Effect: Marks the specified end user profile as inactive and will remove the end user with **UserID** uID from any application instance configurations. The ID of any application for which this user was assigned a mandatory role will be returned.
  - Sequence Diagrams: None
- void setAppInstanceStatus(string status)
  - Effect: Set the current status of an application instance, e.g. suspended or running.
  - Sequence Diagrams: None
- void setApplicationNote(**ApplicationID** appID, string note)
  - Effect: A short description of the current state of the application is attached to that application. Usually this will be no more thatn "running" but a reason of failure (runtime or testing) may also be attached here.
  - Sequence Diagrams: None
- void setNotificationStatus(**NotificationID** notificationID, **NotificationStatus** status)
  - Effect: Set the status of the notification (sent, delivered, stored).
  - Sequence Diagrams: fig. D.20
- void subscriptionActivationChange(**ApplicationInstanceID** appInID, **Timestamp** ts, boolean active)
  - Effect: Adds an (in)activation timestamp to the subscription and marks it as (in)active depending on the boolean value.
  - Sequence Diagrams: None
- void suspendSubscription(**ApplicationInstanceID** appInID, **Timestamp** ts)
  - Effect: **Timestamp** the end of the subscription and mark it as "suspended".

  - Sequence Diagrams: fig. D.22
- boolean unsubscribe(**UserID** custOrgID, **ApplicationID** appID, **Version** version)
  - Effect: Removes the subscription entry of the Customer Organisation with custOrgID for the application with **ApplicationID**. If there was no such subscription, false is returned, true is there was.
  - Sequence Diagrams: fig. D.23
- void updateSubscription(**ApplicationInstanceID** appInID, **Version** to)
  - Effect: The old subscription (application instance) of the given application will be timestamped as "ended". A new subscription will be started (along with a timestamp) of that application (for the new version).
  - Sequence Diagrams: fig. D.26

Diagrams: figs. A.2 and B.4

E.2.52 PnPSensorData

Provided by: `Gateway`, `ManufacturerSpecificDataAdapter`, `SIoTIP system`
Required by: `Mote`
Operations:
- void rcvActuationCallback(int requestID)
  - Effect: Confirm execution of an MicroPnP actuation command.
  - Sequence Diagrams: None
- void rcvData(**PluggableDeviceID** pID, **DeviceData** sensorReading)
  - Effect: Provide MicroPnP sensor data to the gateway (Periodic).
  - Sequence Diagrams: None
- void rcvData(**PluggableDeviceID** pID, **DeviceData** sensorReading, int requestID)
  - Effect: Provide requested MicroPnP sensor data to the gateway (Callback).
  - Sequence Diagrams: None

Diagrams: figs. A.1, A.2 and B.1

E.2.53 ProcessInfo

Provided by: `OSKernel`
Required by: `ApplicationContainer`, `ApplicationHandler`
Operations:
- void getMemoryUserBY(**ProcessID** ID)
  - Effect: Returns the memory used by the application with corresponding processID.
  - Sequence Diagrams: None
- void getProcessID(**ApplicationInstanceID** appID)
  - Effect: Returns the processID of a specific applicationInstance.
  - Sequence Diagrams: fig. D.1

Diagrams: figs. A.2 and B.4

E.2.54 RcvMobileAppMsg

Provided by: `MobileAppCommunication`, `SIoTIP system`
Required by: `MobileApp`
Operations:
- void rcvMobileAppMsg(**MoteCommand** cmd) throws *CommandNotAllowedException*, *DestinationNotAvailableException*
  - Effect: A command is received from a mobile app, the component makes sure the command destination is an `Application` container and then it will ask the `ApplicationCommandRouter` to send it to the correct container.
  - Sequence Diagrams: fig. D.37
- void rcvMobileAppMsg(**DatabaseCommand** cmd)
  - Effect: A command is received from a mobile app, the component makes sure the command destination is an `Application` container and then it will ask the `ApplicationCommandRouter` to send it to the correct container.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.55 RcvSensorData

Provided by: `DataReceiver`, `OSWithGWCommunication`
Required by: `DataHandler`, `Gateway`
Operations:

- void handleData(**PluggableDeviceID** pID, **SystemSensorData** data, int ackNr)
  - Effect: Sensor data is sent from the gateway to the online service.
  - Sequence Diagrams: fig. D.11

Diagrams: figs. A.2, B.1 and B.2

E.2.56 RebootGW

Provided by: `GatewayOS`
Required by: `GWFailureHandler`, `Gateway`, `SIoTIP system`
Operations:

- void reboot()
  - Effect: The operating system of the gateway will reboot the gateway.
  - Sequence Diagrams: None

Diagrams: figs. A.1, A.2 and B.1

E.2.57 Registration

Provided by: `CustomerOrganisationDashboard`, `SIoTIP system`
Required by: `CustomerOrganisationClient`
Operations:

- **HTML** ChangeCustOrgInfo(**SessionID** sessionID, string userName, string password, **PaymentInfo** paymentInfo, **ContactInfo** contactInfo, string companyName, **Address** companyAddress)
  - Effect: The `CustomerOrganisationDashboard` will propagate this data to the `OtherDataDB` and thus change customer organisation info in the system.
  - Sequence Diagrams: None
- **HTML** completeCustOrgRegistration(string userName, string password, **PaymentInfo** paymentInfo, **ContactInfo** contactInfo, string companyName, **Address** companyAddress)
  - Effect: The `CustomerOrganisationDashboard` will do validity checks on the contactInfo, address and paymentInfo. If these are valid, the user will be registered in the system. If not, the `CustomerOrganisationDashboard` will request that the invalid options be corrected.
  - Sequence Diagrams: None
- **HTML** confirmCustOrgRegistration(string link)
  - Effect: This method is called when the representative of an unregistered customer organization follows the confirmation link they received via e-mail. It triggers the `CustomerOrganisationDashboard` to send out the rest of the registration form.
  - Sequence Diagrams: None
- **HTML** enterCompanyEmailAddress(string email)
  - Effect: The `CustomerOrganisationDashboard` checks the validity of the email address. If it is valid, an activation email will be sent. If it is not valid, the user is notified.
  - Sequence Diagrams: None
- **HTML** registerEndUser(**SessionID** sessionID, string endUserName, **ContactInfo** contactInfo)
  - Effect: The `CustomerOrganisationDashboard` will validate the end-user details (end-user already exists or badly formatted contact info). If the information is correct, it is stored in the `OtherDataDB` and the user is informed of this. If not the form needs to be resubmitted.
  - Sequence Diagrams: fig. D.12
- **HTML** requestCustOrgRegistrationForm()
  - Effect: The `CustomerOrganisationDashboard` will make the front end display the first steps of the registration process.
  - Sequence Diagrams: None
- **HTML** requestEndUserRegistrationForm(**SessionID** sessionID)

    - Effect: The `CustomerOrganisationDashboard` will make the front end display the first steps of the end-user registration process.
    - Sequence Diagrams: fig. D.12
- **HTML** unregisterCustOrg(**SessionID** sessionID)
    - Effect: The `CustomerOrganisationDashboard` will ask the `OtherDataDB` to delete any subscriptions this user has, as well as the profile itself. Invoice creation will be triggered before the profile is fully deleted. ApplicationContainers are asked to shut the corresponding application instances down.
    - Sequence Diagrams: None
- **HTML** unregisterEndUser(**SessionID** sessionID, string endUserName, **ContactInfo** contactInfo)
    - Effect: The entry for the specified user for the customer organisation with the **UserID** in the **SessionID** will be removed from the database and from any application instance configurations. Checks will be done to see if those configurations are still valid and whether application instances will have to be suspended.
    - Sequence Diagrams: None
- **HTML** wantToUnregisterEndUser(**SessionID** sessionID)
    - Effect: The customer organisation with the **UserID** in the **SessionID** wishes to receive a list of end users such that they can select which ones they wish to unregister.
    - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.58 RequestData

Provided by: `ManufacturerSpecificDataAdapter`
Required by: `DataHandler`
Operations:

- **SystemSensorData** getData()
    - Effect: Synchronously retrieve the current value of a sensor
    - Sequence Diagrams: None
- void getData(int requestID)
    - Effect: Asynchronously retrieve the current value of the sensor (Callback).
    - Sequence Diagrams: None
- void initialize(**Code** adapter)
    - Effect: Initialize a manufacturer specific adapter with the given code
    - Sequence Diagrams: fig. D.36

Diagrams: fig. B.1

E.2.59 RequestPnPData

Provided by: `Mote`
Required by: `Gateway`, `ManufacturerSpecificDataAdapter`, `SIoTIP system`
Operations:

- **DeviceData** getData()
    - Effect: Synchronously retrieve the current value of the MicroPnP sensor.
    - Sequence Diagrams: None
- boolean getData(int requestID)
    - Effect: Asynchronously retrieve the current value of the MicroPnP sensor (Callback).
    - Sequence Diagrams: None

Diagrams: figs. A.1, A.2 and B.1

E.2.60 RouteDeviceData

Provided by: `DeviceDataRouter`
Required by: `DataReceiver`, `OSWithGWCommunication`
Operations:

- void routeData(**PluggableDeviceID** devID, **SystemSensorData** data)
    - Effect: The `DeviceDataRouter` will use the `ApplicationInstanceInfoDB` to decide which application instances are interested in the given sensor data. It will then call the

`ApplicationManager` once per instance, asking them to schedule the receiving of the data by that instance.
    – Sequence Diagrams: fig. D.11

Diagrams: figs. A.2 and B.2

E.2.61 ScheduleApplication

Provided by: `ApplicationHandler`, `ApplicationManager`, `ApplicationScheduler`, `GWApplicationScheduler`
Required by: `ApplicationCommandRouter`, `ApplicationManager`, `ApplicationStarter`, `DBRequestHandler`, `DataHandler`, `DeviceDataRouter`, `EventManager`, `GWCommandRouter`, `GWEventManager`, `MobileAppCommunication`
Operations:

- void scheduleHandleCrash(**CrashInfo** crashInfo, **ApplicationInstanceID** appInID)
    - Effect: Schedule to notify application instance of an ocurred crash.
    - Sequence Diagrams: None
- void scheduleInitialize(**Code** code)
    - Effect: Schedule to initialize a new application instance.
    - Sequence Diagrams: fig. D.1
- void schedulePluggableDeviceFailed(**DeviceResourceLocator** pRL, **ApplicationInstanceID** InID)
    - Effect: Schedule to notify an application instance that a pluggable device has failed.
    - Sequence Diagrams: fig. D.27
- void scheduleReactToCommand(**InterAppCommand** cmd, **ApplicationInstanceID** appInID)
    - Effect: Schedule to pass the provided command to the application.
    - Sequence Diagrams: figs. D.14 and D.15
- void scheduleReactToCommand(**MoteCommand** cmd, **ApplicationInstanceID** appID)
    - Effect: Schedule to pass the provided command to the application.
    - Sequence Diagrams: fig. D.37
- void scheduleReactToCommand(**DatabaseResultCommand** cmd, **ApplicationInstanceID** appID)
    - Effect: Pass the data provided in the command to the application whom ID is mentioned in parameters.
    - Sequence Diagrams: fig. D.43
- void scheduleReactToData(**PluggableDeviceID** pID, **SystemSensorData** data, **ApplicationInstanceID** appInID)
    - Effect: Schedule to pass the provided sensor data to the application so that it can process it.
    - Sequence Diagrams: figs. D.11 and D.16
- void scheduleWakeUp(**ApplicationInstanceID** appID, **Event** event)
    - Effect: The application instance will be awakened and asked to handle the event.
    - Sequence Diagrams: figs. D.30, D.31 and D.42

Diagrams: figs. A.2, B.1 and B.4

E.2.62 ScheduleEvent

Provided by: `EventManager`, `GWEventManager`
Required by: `ApplicationContainer`, `ApplicationHandler`, `GWApplicationContainer`
Operations:

- void reactToData(**ApplicationInstanceID** appID, **SystemSensorData** data)
    - Effect: The data will be passed to the `EventManager` for processing.
    - Sequence Diagrams: figs. D.31 and D.42
- void scheduleEvent(**ApplicationInstanceID** appID, **Event** event)
    - Effect: The event will be scheduled on the `EventManager`.
    - Sequence Diagrams: fig. D.29

Diagrams: figs. A.2, B.1 and B.4

E.2.63 SendCommandToGW

Provided by: `ApplicationCommandPropagator`, `OSWithGWCommunication`
Required by: `ApplicationCommandRouter`
Operations:

- void sendCommandToGW(**MoteCommand** cmd)
  - Effect: Route the command to the correct gateway according to the **DeviceResourceLocator** in the **Command** (contains the **GatewayID**).
  - Sequence Diagrams: figs. D.2, D.17, D.18 and D.23
- void sendCommandToGW(**UpdateCommand** cmd)
  - Effect: Route the command to the correct gateway according to the **DeviceResourceLocator** in the **Command** (contains the **GatewayID**).
  - Sequence Diagrams: fig. D.22
- void sendCommandToGW(**InterAppCommand** cmd)
  - Effect: Route the command to the correct gateway according to the **DeviceResourceLocator** in the **Command** (contains the **GatewayID**).
  - Sequence Diagrams: fig. D.14

Diagrams: figs. A.2 and B.2

E.2.64 SendInvoice

Provided by: `ThirdPartyInvoicingService`
Required by: `InvoiceManager`, `SIoTIP system`
Operations:

- void sendInvoice(string service, float amount, **Date** dueData, int id)
  - Effect: The external invoicing service is requested to handle the payment.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.65 SendMobileAppMsg

Provided by: `MobileApp`
Required by: `MobileAppCommunication`, `SIoTIP system`
Operations:

- void sendToMobileApp(string hostname, int port, string message)
  - Effect: A message is sent to the specified hostname/port combination.
  - Sequence Diagrams: fig. D.13

Diagrams: figs. A.1 and A.2

E.2.66 SensorData

Provided by: `DataHandler`
Required by: `ManufacturerSpecificDataAdapter`
Operations:

- void rcvActuationCallback(int requestID)
  - Effect: Confirm execution of an actuation command.
  - Sequence Diagrams: None
- void rcvData(**PluggableDeviceID** pID, **SystemSensorData** sensorReading)
  - Effect: Provide sensor data to the gateway (Periodic).
  - Sequence Diagrams: None
- void rcvData(**PluggableDeviceID** pID, **SystemSensorData** sensorReading, int requestID)
  - Effect: Provide requested sensor data to the gateway (Callback).
  - Sequence Diagrams: None

Diagrams: fig. B.1

E.2.67 SensorDataMgmt

Provided by: `OSSensorDataDB`, `SensorDataDB`, `SensorDataLoadBalancer`, `SensorDataScheduler`
Required by: `ApplicationCommandRouter`, `DBRequestHandler`, `DataReceiver`, `OSWithGWCommunication`, `SensorDataLoadBalancer`, `SensorDataScheduler`
Operations:

- void addData(**PluggableDeviceID** pID, **SystemSensorData** data)
  - Effect: Adds sensor data to the database.

  - Sequence Diagrams: fig. D.11
- void getDataFromTimeframe(**ApplicationInstanceID** applnID, list<**DeviceResourceLocator**> devices, **Timestamp** from, **Timestamp** to)
  - Effect: All sensor data for the given pluggable devices within the given timeframe will be sent as a **Command** to the application instance with the given id.
  - Sequence Diagrams: fig. D.43

Diagrams: figs. A.2, B.2 and B.3

E.2.68 SessionMgmt

Provided by: `SessionDB`
Required by: `AuthenticationHandler`
Operations:

- Map<**SessionAttributeKey**, **SessionAttributeValue**> isValidSession(**SessionID** sID) throws *NoSuchSessionException*
  - Effect: Returns the session attributes corresponding to the session with the supplied **SessionID** if it is a valid session.
  - Sequence Diagrams: None
- void deleteSession(**SessionID** sID) throws *NoSuchSessionException*
  - Effect: Deletes the session associated with the given **SessionID** from the database.
  - Sequence Diagrams: None
- **SessionID** newSession(**UserID** uID)
  - Effect: Generates a new **SessionID** for the given **UserID** and stores this as an active session.
  - Sequence Diagrams: None

Diagrams: fig. A.2

E.2.69 StateMgmt

Provided by: `AppStateDataDB`, `AppStateDataLoadBalancer`, `AppStateDataScheduler`, `OSAppStateDB`
Required by: `AppStateDataLoadBalancer`, `AppStateDataScheduler`, `ApplicationContainer`, `ApplicationHandler`, `ApplicationStarter`
Operations:

- **State** getMostRecentStateForApp(**ApplicationInstanceID** id)
  - Effect: Each state is saved along with a timestamp. getMostRecentStateForApp call returns the most recent state of the application instance.
  - Sequence Diagrams: figs. D.8 and D.35
- **State** getMostRecentStateForApp(**ApplicationInstanceID** id, **GatewayID** gatewayID)
  - Effect: Each state is saved along with a timestamp. getMostRecentStateForApp call returns the most recent state of the application instance.
  - Sequence Diagrams: None
- **State** getStateAtTime(**Timestamp** time)
  - Effect: The state of the application for a specific time interval will be retrieved from the `OSAppStateDB`.
  - Sequence Diagrams: None
- void saveState(**State** state, **ApplicationInstanceID** id)
  - Effect: Save the state of the application instance on the `OSAppStateDB`.
  - Sequence Diagrams: fig. D.34
- void saveState(**State** state, **ApplicationInstanceID** id, **GatewayID** gatewayID)
  - Effect: Save the state of the application instance on the `OSAppStateDB`.
  - Sequence Diagrams: None

Diagrams: figs. A.2, B.4 and B.5

E.2.70 Subscription

Provided by: `CustomerOrganisationDashboard`, `SIoTIP system`
Required by: `CustomerOrganisationClient`
Operations:

- **HTML** applicationCriticality(**ApplicationID** appID, bool isCritical, **Version** appVersion, **SessionID** sID)

  - Effect: The user to which the session belongs selects the criticality a certain application has to them.
  - Sequence Diagrams: fig. D.21
- **HTML** assignEndUsers(list<Tuple<**UserID**, string>> assignments, **SessionID** sID, **Version** version, **ApplicationID** appID)
  - Effect: The user to which the session belongs has assigned the end user roles, next, the user is asked the criticality of the application. After selecting it, all information regarding the subscription is stored in the `OtherDataDB` and the customer organisation is unsubscribed from previous versions of that application, if any. The `ApplicationStarter` is then asked to activate the application.
  - Sequence Diagrams: fig. D.21
- **HTML** browseApplications(**SessionID** sID)
  - Effect: The primary actor indicates they want to subscribe to an application. The customer organisition can be identifier based on the provides session information.
  - Sequence Diagrams: fig. D.21
- **HTML** selectApplication(**ApplicationID** appID, **SessionID** sID, **Version** version)
  - Effect: The user to which the session belongs selects an application to subscribe to, the `CustomerOrganisationDashboard` will get fetch the **Topology** corresponding to the user and will match it with the **ApplicationRequirements**. The configuration that still needs to be done is presented to the user by the front end.
  - Sequence Diagrams: fig. D.21
- **HTML** setTopologyConfiguration(**TopologyRequirementsInstantiation** appReqIn, **Version** version, **ApplicationID** appID)
  - Effect: The user to which the session belongs is finished with configuring the **Topology**. The `CustomerOrganisationDashboard` will look up the required end-user roles and the end-users associated with the customer organisation in the `OtherDataDB` and will present them together to the user via the front end such that they may assign the roles.
  - Sequence Diagrams: fig. D.21
- **HTML** unsubscribe(**SessionID** sID, **ApplicationID** appID, **Version** appVersion)
  - Effect: The user to which the session belongs wishes to cancel a subscription. The subscription matching the **UserID** and **ApplicationID** is removed from the database, the `InvoiceManager` is asked to create an invoice and the application is deactivated by the `ApplicationStarter`.
  - Sequence Diagrams: None
- **HTML** wantToUnsubscribe(**SessionID** sID)
  - Effect: The user to which the session belongs indicates they want to unsubscribe from an application. The `CustomerOrganisationDashboard` will fetch all applications the user is currently subscribed to from the `OtherDataDB` and will ask the front-end to present these.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.71 TestApplication

Provided by: `AppTester`
Required by: `ApplicationDeveloperDashboard`
Operations:

- Tuple<boolean, string> testCode(**Code** code, **ApplicationID** appID, **UserID** appDevID)
  - Effect: Run tests on the provided code and return true if they all pass and false if they don't, together with a string witch contains a small description of the failure (empty if successful).
  - Sequence Diagrams: fig. D.4

Diagrams: fig. A.2

E.2.72 TopologyMgmt

Provided by: `TopologyDB`, `TopologyLoadBalancer`
Required by: `ApplicationCommandRouter`, `ApplicationStarter`, `CustomerOrganisationDashboard`, `DBRequestHandler`, `DeviceStatusMonitor`, `InfrastructureOwnerDashboard`, `OSWithGWCommunication`, `TopologyLoadBalancer`
Operations:

- void addPluggableDeviceAdapter(**PluggableDeviceType** pType, **Code** adapter)
  - Effect: The code necessary for a new pluggable device is added to the `TopologyDB`. This code should be inserted by a SIoTIP developer.
  - Sequence Diagrams: None
- **Code** getAdapterForPluggableDevice(**PluggableDeviceType** pType)
  - Effect: returns the code necessary to create an adapter component for the pluggable device in the `Gateway`.
  - Sequence Diagrams: fig. D.36
- list<**UserID**> getCostumerOrganisationsWithAccessToDevice(**PluggableDeviceID** pID)
  - Effect: Returns all UserIDs where an access right entry exists for the Pluggable Device identified by pID.
  - Sequence Diagrams: None
- Map<String, String> getDeviceConfiguration(**PluggableDeviceID** pID)
  - Effect: Fetch the current configuration of the specified pluggable device.
  - Sequence Diagrams: None
- list<Tuple<**PluggableDeviceID**, **PluggableDeviceType**, **GatewayID**>> getDevicesForInfrastructureOwner(**UserID** infOwnerID)
  - Effect: Will return the IDs of all devices which belong to a topology of which the user with the specified **UserID** is the owner. **PluggableDeviceType** and **GatewayID** are also returned to give context about the nature of the device.
  - Sequence Diagrams: None
- list<**DeviceResourceLocator**> getPluggableDevicesForGW(**GatewayID** gwID)
  - Effect: Return the DeviceResourceLocators of the pluggable devices associated with the given gateway.
  - Sequence Diagrams: fig. D.6
- **Topology** getTopologyForUser(**UserID** infOwnerID)
  - Effect: Retrieve the topology of the infrastructure owned by the infrastructure owner with the specified **UserID**.
  - Sequence Diagrams: fig. D.21
- void grantAccess(**UserID** uID, **PluggableDeviceID** pID)
  - Effect: A new access right (customer organisation gaining access to a specific device) is stored in the `TopologyDB`.
  - Sequence Diagrams: None
- boolean hasCorrectAdapter(**GatewayID** gwID, **PluggableDeviceType** pType)
  - Effect: Checks whether a gateway had the correct adapter to turn the API of a specific manufacturer into a general API used by the `DataHandler`.
  - Sequence Diagrams: fig. D.10
- **UserID** newMote(**MoteInfo** moteInfo, **GatewayID** gwID)
  - Effect: A new mote is added as unplaced to the topology to which the gateway with the specified ID belongs to. The **UserID** of the user which is the infrastructure owner of that topology is returned.
  - Sequence Diagrams: None
- void newPluggableDeviceInMote(int moteID, **PluggableDeviceID** pID, **PluggableDeviceType** pType)
  - Effect: The Pluggable Device will be associated with the specified mote and will be marked as uninitialized.
  - Sequence Diagrams: fig. D.10
- void registerAdapterForGateway(**GatewayID** gwID, **PluggableDeviceType** pType)
  - Effect: This operation should be called when an adapter for a specific **PluggableDeviceType** has been given to a `Gateway`. It registers that the gateway has the ability to adapt manufacturer specific API into a general API, used by the `DataHandler`.
  - Sequence Diagrams: None
- void removePluggableDevice(**DeviceResourceLocator** devRL)
  - Effect: Removes the entry for the pluggable device with the matching **DeviceResourceLocator** from the `TopologyDB`.
  - Sequence Diagrams: None
- void setDeviceConfiguration(**PluggableDeviceID** pID, Map<string, string> config)
  - Effect: Store the current configuration of the specified pluggable device.
  - Sequence Diagrams: None

- void setTopologyForUser(**Topology** newTopology)
  - Effect: Sets the topology for a certain user (infrastructure owner) to newTopology. If any devices (identified by **PluggableDeviceID**) who were previously not included yet are now included, they are marked as active and that change will be propagated to the `DeviceInitCache`.
  - Sequence Diagrams: None

Diagrams: figs. A.2 and B.2

E.2.73 UpdateMessage

Provided by: `ApplicationStarter`
Required by: `ApplicationCommandRouter`, `ApplicationHandler`, `ApplicationScheduler`
Operations:

- void gwContainerInitialize(bool failed, **ApplicationInstanceID** appInID, **GatewayID** gwID)
  - Effect: A call informing the `ApplicationStarter` that a (new) container on the gateway has (un)successfully been initialized and whether the old container has been killed.
  - Sequence Diagrams: fig. D.25
- void gwOldApplicationContainerShutDownCompleted(**ApplicationInstanceID** oldAppInID, **GatewayID** gwID)
  - Effect: A call informing the `ApplicationStarter` that the old `GWApplicationContainer` was successfully shut down and that the new one has been activated.
  - Sequence Diagrams: fig. D.25
- void gwUpdateContainerCreated(**ApplicationInstanceID** appInID, **GatewayID** gwID)
  - Effect: A call informing the `ApplicationStarter` that the parallel update `GWApplicationContainer` was successfully created on a certain gateway.
  - Sequence Diagrams: fig. D.24
- void osContainerInitialize(boolean failed, **ApplicationInstanceID** appInID)
  - Effect: A call informing the `ApplicationStarter` that the new container on the Online Service has (un)successfully been initialized and whether the old container has been killed.
  - Sequence Diagrams: fig. D.25
- void osOldApplicationContainerShutDownCompleted(**ApplicationInstanceID** appInID)
  - Effect: A call informing the `ApplicationStarter` that the old `ApplicationContainer` was successfully shut down on a certain gateway and that the new one has been activated.
  - Sequence Diagrams: fig. D.24

Diagrams: figs. A.2 and B.4

E.2.74 UploadApplication

Provided by: `ApplicationDeveloperDashboard`, `SIoTIP system`
Required by: `ApplicationDeveloperClient`
Operations:

- **HTML** automaticActivation(bool automatic, **SessionID** sID)
  - Effect: The `ApplicationDeveloperDashboard` will remember the fact that that developer wants the update to be either automatic or not. Depending on the answer either the previous versions of the application will be fetched and asked to be displayed by the front-end (in case of automatic), or the form for the application specifications will be displayed.
  - Sequence Diagrams: fig. D.4
- **HTML** upload(**Code** code, string description, **ApplicationMetaData** metadata, **SessionID** sID, **Application-Requirements** appReq)
  - Effect: The provided application specifications are first be stored in the `OtherDataDB`, then they are tested by the `AppTester` and if the tests pass the application is made available in the store. If they fail the user is notified. In case an automatic update was requested, the appropriate update data is passed to the `ApplicationStarter`.
  - Sequence Diagrams: fig. D.4
- **HTML** uploadExisting(bool exists, **SessionID** sID)
  - Effect: The `ApplicationDeveloperDashboard` will either ask the front end to show the form used for entering the application specifications (when the boolean is false) or make a list of the applications of that application developer and ask the front-end to show that list.
  - Sequence Diagrams: fig. D.4
- **HTML** uploadSelect(**ApplicationID** aID, **SessionID** sID)

- Effect: The `ApplicationDeveloperDashboard` will remember the fact that that the user wants to update the specified application and will ask the front end to display the question of whether or not the update should be automatic.
  - Sequence Diagrams: fig. D.4
- **HTML** versionsToUpdate(list<**Version**> versions, **SessionID** sID)
  - Effect: The `ApplicationDeveloperDashboard` will remember the fact that that user wants the specified versions to be automatically updated and will ask the front-end to display the application specification form.
  - Sequence Diagrams: fig. D.4
- **HTML** wantToUpload(**SessionID** sID)
  - Effect: The `ApplicationDeveloperDashboard` will start the application uploading procedure by making the front-end display the question of whether the developer wants to upload an existing application.
  - Sequence Diagrams: fig. D.4

Diagrams: figs. A.1 and A.2

E.2.75 UserAuthentication

Provided by: `ApplicationDeveloperDashboard`, `AuthenticationHandler`, `CustomerOrganisationDashboard`, `InfrastructureOwnerDashboard`, `SIoTIP system`, `SysAdminDashboard`
Required by: `ApplicationDeveloperClient`, `ApplicationDeveloperDashboard`, `CustomerOrganisationClient`, `CustomerOrganisationDashboard`, `InfrastructureOwnerClient`, `InfrastructureOwnerDashboard`, `SysAdminClient`, `SysAdminDashboard`
Operations:

- **SessionID** login(**Credentials** cred) throws *IncorrectCredentialsException*
  - Effect: Verify the credentials with the `OtherDataDB`. If valid, a new session is created (with a new ID) which is stored in the `SessionDB`. The users **UserID** will be added to this session as an attribute.
  - Sequence Diagrams: None
- bool logout(**SessionID** sID)
  - Effect: The session identified by the provided id sID is removed from the `SessionDB` if it exists. Otherwise nothing happens.
  - Sequence Diagrams: None

Diagrams: figs. A.1 and A.2

E.2.76 VerifySession

Provided by: `AuthenticationHandler`
Required by: `ApplicationDeveloperDashboard`, `CustomerOrganisationDashboard`, `InfrastructureOwnerDashb`
`SysAdminDashboard`
Operations:

- Map<**SessionAttributeKey**, **SessionAttributeValue**> verifySession(**SessionID** sID) throws *NoSuchSessionException*
  - Effect: Verify whether the provided session identifier sID corresponds to an existing session and, if so, return the session attributes.
  - Sequence Diagrams: None

Diagrams: fig. A.2

E.2.77 ViolationDataMgmt

Provided by: `ViolationDataDB`
Required by: `ViolationManager`
Operations:

- list<**ViolationInfo**> getViolationData(**ApplicationInstanceID** appID)
  - Effect: Returns the number of violations for a specific application instance.
  - Sequence Diagrams: fig. D.33
- void removeAppData(**ApplicationInstanceID** appID)
  - Effect: Removes the application instance record from the `ViolationDataDB`.

  - Sequence Diagrams: fig. D.33
- void saveViolationData(**ApplicationInstanceID** appID, **ViolationInfo** vInfo)
  - Effect: Number of violations will be saved in `ViolationDataDB`. For each application instance there will be only one record.
  - Sequence Diagrams: fig. D.33

Diagrams: fig. A.2

E.2.78 ViolationProcessor

Provided by: `ViolationManager`
Required by: `ApplicationContainer`, `ApplicationHandler`
Operations:

- void processViolation(**ApplicationInstanceID** appID, **ViolationInfo** vInfo)
  - Effect: A violation from an application instance is to be saved. If the the number of violations is higher than 20, the application will be suspended.
  - Sequence Diagrams: figs. D.2, D.33, D.37, D.38 and D.43

Diagrams: figs. A.2 and B.4

E.3 Exceptions

- *ApplicationUncaughtException* Thrown when an application throws an exception which it does not catch itself.
- *CommandNotAllowedException* Thrown when an external system tries to execute a command that is not allowed (e.g. Commands which are not actuation commands or messages to applications).
- *DestinationNotAvailableException* Thrown when the destination for which the command was meant is not available.
- *IncorrectCredentialsException* Thrown if the provided credentials are incorrect.
- *NoSuchApplication* Thrown if no application with the provided identifier exists.
- *NoSuchApplicationInstance* Thrown when no application instance with the given identifier exists.
- *NoSuchInfrastructure* Thrown when no infrastructure owner exists with the provided identifier.
- *NoSuchPluggableDeviceType* Thrown when the given pluggable device type does not exist in the system.
- *NoSuchSessionException* Thrown if no session exists with the provided identifier
- *NoSuchSubscription* Thrown when no subscription exists between the given UserID and ApplicationID.
- *NoSuchUserException* Thrown if no user with the provided identifier exists.
- *UserAlreadyExistsException* Thrown when a user already exists for the specified identifier and/or contact information.

E.4 Data types

- **ActuationCommand**:
  Subclass of: **MoteCommand**
  Attributes: string cmd
  A data structure representing an actuation command for an actuator. It inherits a DeviceResourceLocator from MoteCommand to find the actuator for which the command is meant
- **Address**:
  A data structure representing a physical address in the world. May contain info such as Country, city, postal code, street name and street number.
- **ApplicationConfigurationInstantiation**:
  Data structure representing the assigned pluggable devices (as DeviceResourceLocators) to an application instance.
- **ApplicationID**:
  Uniquely identifies an application. Note that there may be multiple versions of this application (different Version in the database).
- **ApplicationInstanceID**:
  Attributes: **UserID** custOrgID, **Version** version

Uniquely identifies an application instance.

- **ApplicationMetaData**:
  All metadata attached to an application such as an ID, a name, a version and a description.
- **ApplicationRequirements**:
  Attributes: **TopologyRequirements** topologyRequirements, list<**RoleDescription**> endUserRequirements, **MaxMemoryAllowed** memory
  A data structure representing the various requirements which need to be fulfilled before an application instance of that application is able to run. This includes topology requirements and end-user role requirements.
- **ApplicationStatistics**:
  A data structure representing detailed statistics of a certain application. Contains, for example, the number of customer organisations subscribed to the application.
- **Code**:
  Represents the code/libraries/other dependencies needed to run an instance of an application. This includes the location the code needs to run (Gateway/Online Service).
- **Command**:
  A data structure representing a command of any type in the system. This includes the type (sort of request to database, actuation command, message between parts of an application instance, communication from and to external mobile apps, reconfiguration command, application update messages). Also includes extra information depending on the type of command (e.g. what the actuation command is, timeframe for sensor database request, the ResourceLocator of the actuator, the name of the actuation command, etc.). It also includes the ApplicationInstanceID of the application instance that sent it. Most of the internal communication is done by sending Commands around.
- **CommunicationChannel**:
  An enum representing the available channels for communication with external parties, initially only email and SMS.
- **ConfigurationCommand**:
  Subclass of: **MoteCommand**
  Attributes: Map<string, string> config
  A data structure representing a reconfiguration command for an actuator or sensor. It inherits a DeviceResourceLocator from MoteCommand to find the actuator or sensor for which the command is meant
- **ContactInfo**:
  Describes the way a certain user should be contacted in case they need to be notified. Includes a CommunicationChannel and any additional information needed for the message to be sent (e.g. email address/phone number).
- **ConversionCode**:
  A library (or part of) used to convert sensor data from one unit to another, for example Celsius to Fahrenheit.
- **CrashInfo**:
  Contains information on how/why a function call of an application failed. An application can use this information to perhaps avoid such failures in the future.
- **Credentials**:
  The authentication credentials of a user of the SIoTIP system. The credentials include the identifier of the user and a proof of their identity, for example a username and password.
- **DatabaseCommand**:
  Attributes: string query
  A data structure representing a database request that is routed by the ApplicationCommandRouter to the DBRequestHandler. The query should be an SQL-query
- **DatabaseResultCommand**:
  A data structure representing the result of a database Query. The responses are encapsulated within a JSON message
- **Date**:
  Represents the Date type the external invoice components use.
- **DeviceCategory**:
  A datatype denoting the category of a device, for example a thermometer or a smoke detector.
- **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.

- **DeviceResourceLocator**:
  Can be used to locate a specific pluggable device since it includes info about what infrastructure/gateway/mote it belongs to.
- **Event**:
  Represents an event. It can be a time based event or data dependent event.
- **GatewayID**:
  Uniquely identifies a gateway.
- **HTML**:
  Hypertext Markup Language file, possibly stored as a string.
- **InfrastructureID**:
  Uniquely identifies an infrastructure.
- **InterAppCommand**:
  Attributes: **GatewayID** gatewayID, **ApplicationInstanceID** appInID
  InterAppCommand is used by different application instances to interact with each other.
- **IPAddress**:
  Represents an IP address of some device.
- **MaxMemoryAllowed**:
  Maximum allowed memory for a specific applicaton.
- **MoteCommand**:
  Attributes: **DeviceResourceLocator** pRL
  A data structure representing a command for a device on a mote. It can be either an ActuationCommand or a ConfigurationCommand.
- **MoteID**:
  Uniquely identifies a mote.
- **MoteInfo**:
  Attributes: int moteID, int manufactorerID, int iproductID, 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.
- **NotificationID**:
  Uniquely identifies a notification.
- **NotificationStatus**:
  An enum describing the various states a notification which needs to be sent to a user can be in (sent, delivered or read).
- **NotificationTrigger**:
  A data structure containing information about the reason a notification has to be sent. Depending on the reason, an ID of some sort may be attached to find the correct notification configuration. Examples of this are: Gateway failure (with UserID of Infrastructure Owner) New Mote is available (with UserID of Infrastructure Owner) An application instance crashed (with ApplicationInstanceID) Application tests failed/succeeded (with UserID of application developer)
- **PaymentInfo**:
  Contains Information on the method of payment (Credit Card or Zoomit), along with extra information needed to send invoices (such as credit card information).
- **PluggableDeviceID**:
  A unique identifier of the Pluggable Device.
- **PluggableDeviceType**:
  Denotes the type of the Pluggable Device.
- **ProcessID**:
  Represents the processID.
- **RoleDescription**:
  Attributes: string name, int required, int optional
  A data structure representing the need for a specific role for an application. The name of the role is a string, and the amount of required and optional user which need to fulfill this role for

the application to work are given as integers.

- **SessionAttributeKey**:
  The key of an attribute attached to a session.
- **SessionAttributeValue**:
  The value of an attribute attached to a session.
- **SessionID**:
  A piece of information uniquely identifying a user session in the SIoTIP system. Contains at least the user's UserID and the time the session was initiated (TimeStamp).
- **State**:
  Represents the state of the system.
- **SystemSensorData**:
  A data structure used to represent different kinds of device data in a generic manner. Contains the converted JSON map (which held sensor values or actuator state) as well as a TimeStamp representing the time the DataHandler received the data.
- **Timestamp**:
  The representation of a time (e.g. year/month/day/time) in the system
- **Topology**:
  The topology shows a representation of the gateways, and the motes and pluggable devices associated with each gateway, and specifies key relationships between these devices, such as physical location information, device model/category information, device status (e.g. uninitialized). There is one such topology per infrastructure.
- **TopologyRequirements**:
  Specifies the topological requirements for an application: mandatory and optional devices, device configurations, device relations and redundancy information. Device information entail a DeviceCategory, along with the physical requirements of the device (update speed, needed configuration settings). By using this format, multiple device models can be flagged as useable for a certain application.
- **TopologyRequirementsInstantiation**:
  The instantiation of the requirements regarding topology for an application. These are always linked to a description and include information such as the devices/gateways associated with the instance and information on the devices themselves such as model/category.
- **UpdateCommand**:
  A data structure representing an update command. It is used for either software updates or to change the state of the application.
- **UserID**:
  A piece of data uniquely identifying a user (Application Developer, SIoTIP system administrator, Customer Organisation or Infrastructure Owner) in the SIoTIP system.
- **Version**:
  Uniquely identifies a version of an application. Includes metadata such as the version number.
- **ViolationInfo**:
  Contains the violation information. This information contains type of violation