Katholieke
Universiteit
Leuven

Departement
Computerwetenschappen

REMOTE MEASUREMENT, MONITORING AND CONTROL SYSTEM

The complete architecture

Software Architecture (H09B5a) - Part 2b

Toon Nolten (r0258654)
Nele Rober (r0262954)

Academic year 2013-2014

Contents

List of Figures

1 Introduction

The architecture satisfies all non-functional requirements while kept as simple as possible. This results in an architecture that allows for an easy division of the implementation of the subsystems across fairly independent business units. It also allows the investers to minise their risk by implementing the system in a modular fashion, thereby testing the waters to see if people are interested in the services *ReMeS* offers.

2 Overview

2.1 Architectural decisions

***Av2*: Missing Measurements** *ReMeS* should be able to detect missing measurements. This responsability is assigned to a MeasurementArrivalMonitor. All RMM-*ReMeS* communication passes through the RMMFacade and is acknowledged. Availability of communication is very important, therefore a CommunicationAvailabilityMonitor is added to the RMMCommunication.

***Av3*: Third Party Billing Service Failure** To improve reliability of communication between *ReMeS* and third parties a ThirdPartyCommunication is added. This component is used to make sure that invoices either arrive at the Third Party Billing Service or an operator is notified.

***P2*: Anomaly Detection** To enable performant anomaly detection, the AnomalyScheduler identifies when to go to overloaded modus and will then schedule incoming measurements appropriately. Multiple instances of MeasurementProcessors and AlarmValidator allow to process any number trames within the required deadline.

***P3*: Requests to the Measurement Database** When in overloaded modus, the system should go back to normal modus as soon as possible while giving priority to premium service level customers. A scheduler is necessary to accomplish this.

***M2*: Fine-grained Metering for Enterprises** The CustomerDB knows who should pay for the measurements and who should be paid. This allows an easy modification of the Billing component when secondary RMMs are added.

***M3*: Decentralized Electricity Generation** The CustomerDB knows who should pay for the measurements and who should be paid. This allows an easy modification of the Billing component when RMMs that measure production of electricity are added.

2.2 Discussion

Responsibilities Every service of *ReMeS* is provided by a different component. This makes it possible to implement and maintain each service independently, allowing responsibility for subsystems to be delegated to individual business entities with a minimum of interdependence.

Another benefit of this clear separation lies in the fact that *ReMeS* can be bootstrapped in a modular fashion. Provided the core functionalities are available (i.e. receiving measurements, storing measurements and making them available to our customers), *ReMeS* can start functioning (as a pilot project: generating valuable feedback early) and add other functionality incrementally. This decreases the initial investment and associated risk, making the project more attractive from a business standpoint.

Lastly it allows for graceful degradation of the system: a failure of the billing subservice does not influence unrelated functionality. Even when the communication with RMMs fails, users can still consult an overview of measurements up to the failure.

Coupling The databases are highly coupled with other components of *ReMeS*. However the customer database is often only needed in case a subsystem is overloaded: only then its scheduler needs the SLAs to infer the priority. In the current architecture the SLAs have to be cached, to avoid shifting the load to the customer database.

In retrospect we might have better distributed the data over more databases, e.g. separate information regarding devices from information regarding customers, providing a seperate database for SLAs, etc.

Security Having a single component responsible for most of the communication with the outside world reduces the attack surface of *ReMeS*. This makes it easier to focus security related development.

Currently a user is identified by a secret token he receives when logging in. However, because of lack of knowledge, we are not sure this is a proper way of securing the communication. We did not focus on improving this aspect of the architecture because there were no security related non-functional requirements.

3 Context

External components can communicate with *ReMeS* through three components: RMMCommunication, ThirdPartyCommunication and NotificationHandler. All communication between *ReMeS* and RMMs goes through the RMMCommunication, e.g. reporting measurements and alarms, configuring and actuating modules.

All communication with third parties is handled by the ThirdPartyCommunication and the NotificationHandler. The first provides 'online' communication, the second 'offline' communication, e.g. regular mail, SMS, etc. The ThirdPartyCommunication communicates with all kinds of interfaces, for example an online web client for customers, the UIS.

Figure 1 illustrates these three components and the interfaces offered to third parties. The following interfaces are considerd:

- *UtilityInformationSystem:* The interface provided by the utility company.
- *ThirdPartyBillingService:* The interface provided by an external service that handles the sending of invoices and the following up of payments.
- *UserInterface:* A web interface that a user (customer and technician) can use to communicate with *ReMeS*.
- *CallCenter:* A web interface that a CallCenter operator can use to communicate with *ReMeS*.
- *PostalServce:* The interface provided by an external service that sends regular mail for *ReMeS*.
- *SMSService:* The interface provided by an external service that sends text messages (sms) for *ReMeS*.
- *EmailService:* The interface provided by an external service that e-mail for *ReMeS*.
- *RemoteModule:* The interface provided by an RMM. All communication goes via trames.
- *RemoteActuator:* The interface provided by a remote actuator. All communication goes via trames.

4 Architectural overview

ReMeS provides several services: collection and storage of measurements, anomaly detection, billing, etc. Each of these services is provided by a different component (respectively RMMCommunication, MeasurementDBManager, AnomalyDetector, Billling, etc.), figure 2.

4.1 Main architectural decisions

4.1.1 *Av2*: Missing Measurements

Design figure 3

1. The MeasurementArrivalMonitor can detect missing meaurements by comparing the elapsed time with the expected arrival time of a trame. The expected time is calculated from the arrival time of the last trame of that RMM and the scheduled trame frequency.
2. All communication between RMMs and *ReMeS* passes through and is acknowledged by the RMMFacade.
3. The CommunicationAvailabilityMonitor regulary pings the Communication components and notifies an operator when one of them does not respond.
4. When the communication channel fails, the MeasurementArrivalMonitor keeps track of how long the failure remains.

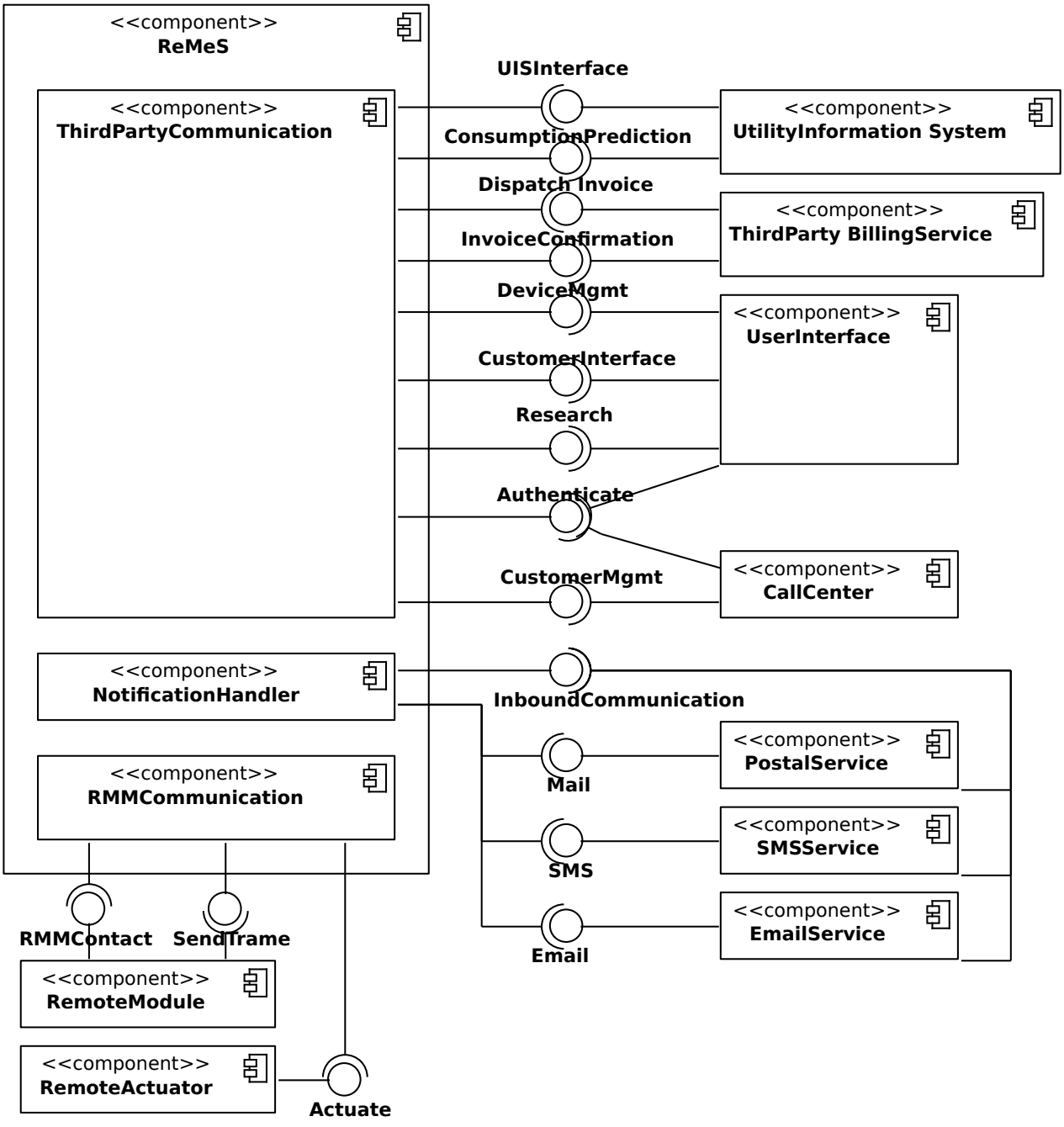

Figure 1: Context diagram for *ReMeS*.

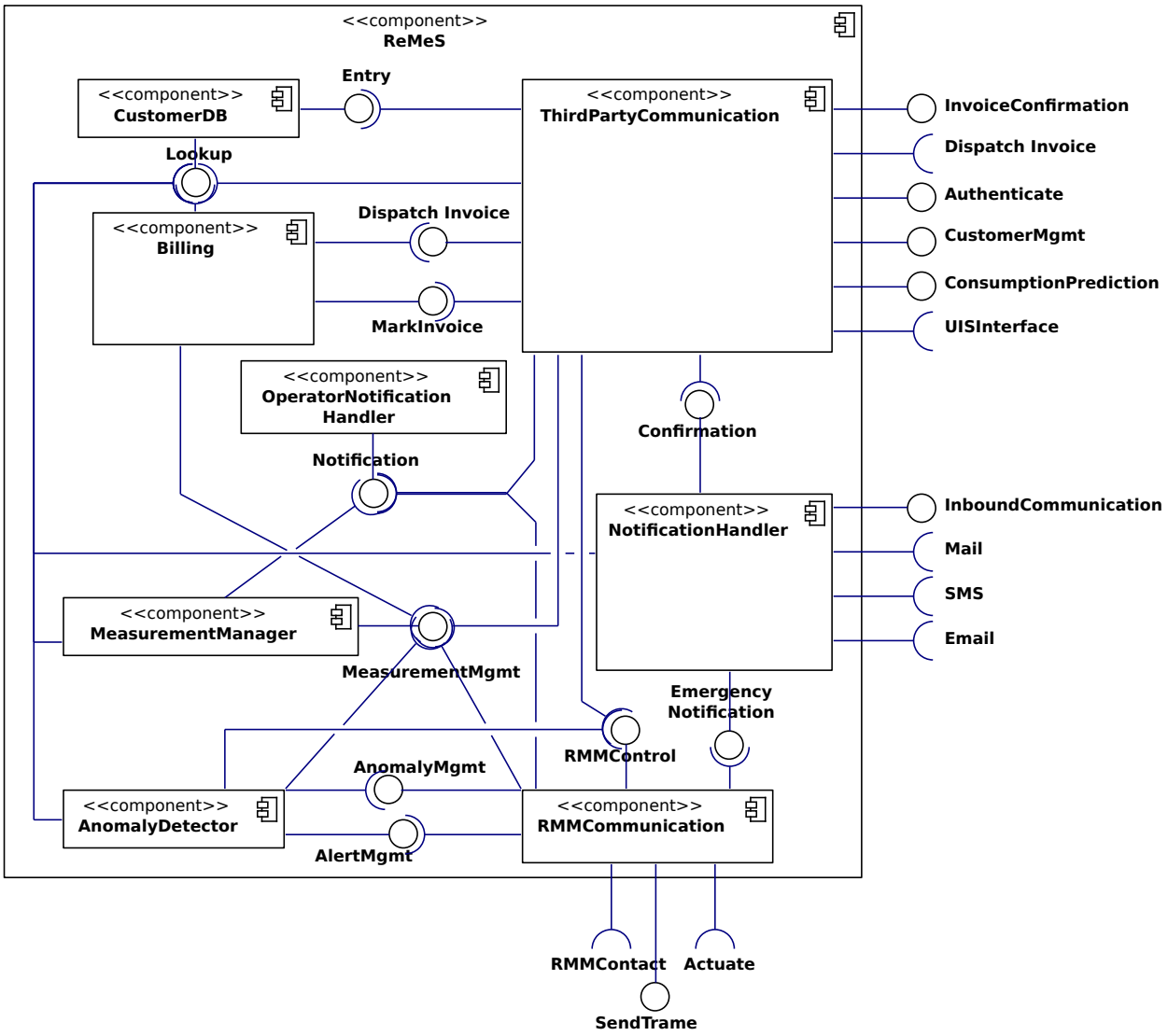

Figure 2: Main components

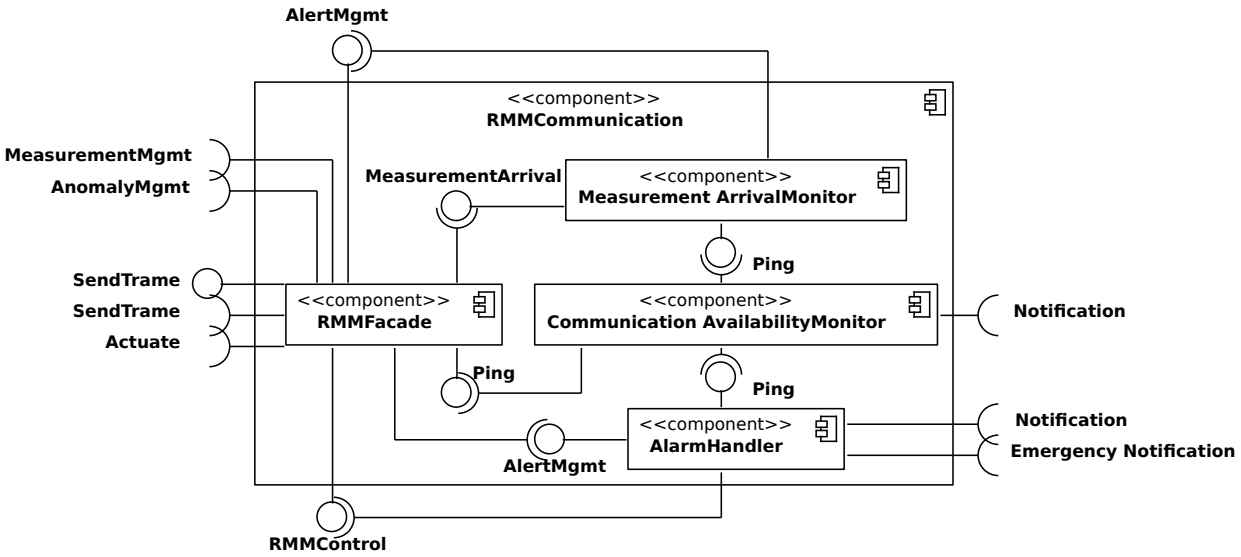

Figure 3: Decomposition of the `RMMCommunication`

Tactics and patterns used:

- Ping-echo
- Facade

Rationale Alarms are irrelevant for the detection of missing measurements so a RMMFacade forwards only incoming measurements to the MeasurementArrivalMonitor. To minimise the load on the RMMFacade, the checking for missing measurements is done by a seperate component: the MeasurementArrivalMonitor. Detecting failures should happen on a different component than the component to check. The CommunicationAvailabilityMonitor is created for this purpose. Other Communication components can be monitored with it later on.

Alternative for the CommunicationAvailabilityMonitor The components of the Communication subsystem could be checked for availability by other components of the Communication subsystem, for example, the Alarm Handler could check the MeasurementArrivalMonitor. However, all components would have to know how to notify the operators and all components would have to run on different machines so this would be less flexible than the CommunicationAvailabilityMonitor.

Alternative for the RMMFacade The RMMs could send measurements and alarms to different receivers, but this complicates communication to and from RMMs.

4.1.2 *Av3*: Third Party Billing Service Failure

Design figure 4

1. Every request to the Third Party Billing Service is acknowledged to the ThirdPartyCommunication component.
2. The ThirdPartyCommunication component caches all billing requests untill they are acknowledged.
3. When the ThirdPartyCommunication component does not reveice an acknowledgement in time, the request is resent in a proper fashion, e.g. exponential backoff.
4. The ThirdPartyCommunication keeps track of the number of times a request is sent but not acknowledged and notifies an operator when this happens five times.

Tactics and patterns used:

- Facade

Rationale Availability of third party components may not be limited to the Third Party Billing Service: other third parties may also require some kind of action when they become unavailable. Therefore the ThirdPartyCommunication is assigned this task.

Alternative for availability in the ThirdPartyCommunication The Billing component may handle the resending of unacknowledged requests itself, but then it would not be possible to reuse this functionality for other components that communicate with third parties.

4.1.3 *P2*: Anomaly Detection

Design figures 5 and 6

1. In normal modus, the AnomalyScheduler forwards the incoming measurements in first-in, first-out order. Alarms are handled before measurements and scheduled based on priority.
2. In overloaded modus, the AnomalyScheduler forwards the incoming measurements by prioritising based on the SLA. Alarms are handled before measurements, but only when this does not endanger the individual deadline of measurements.
3. Based on the load multiple instances of components other than the AnomalyScheduler can be added.

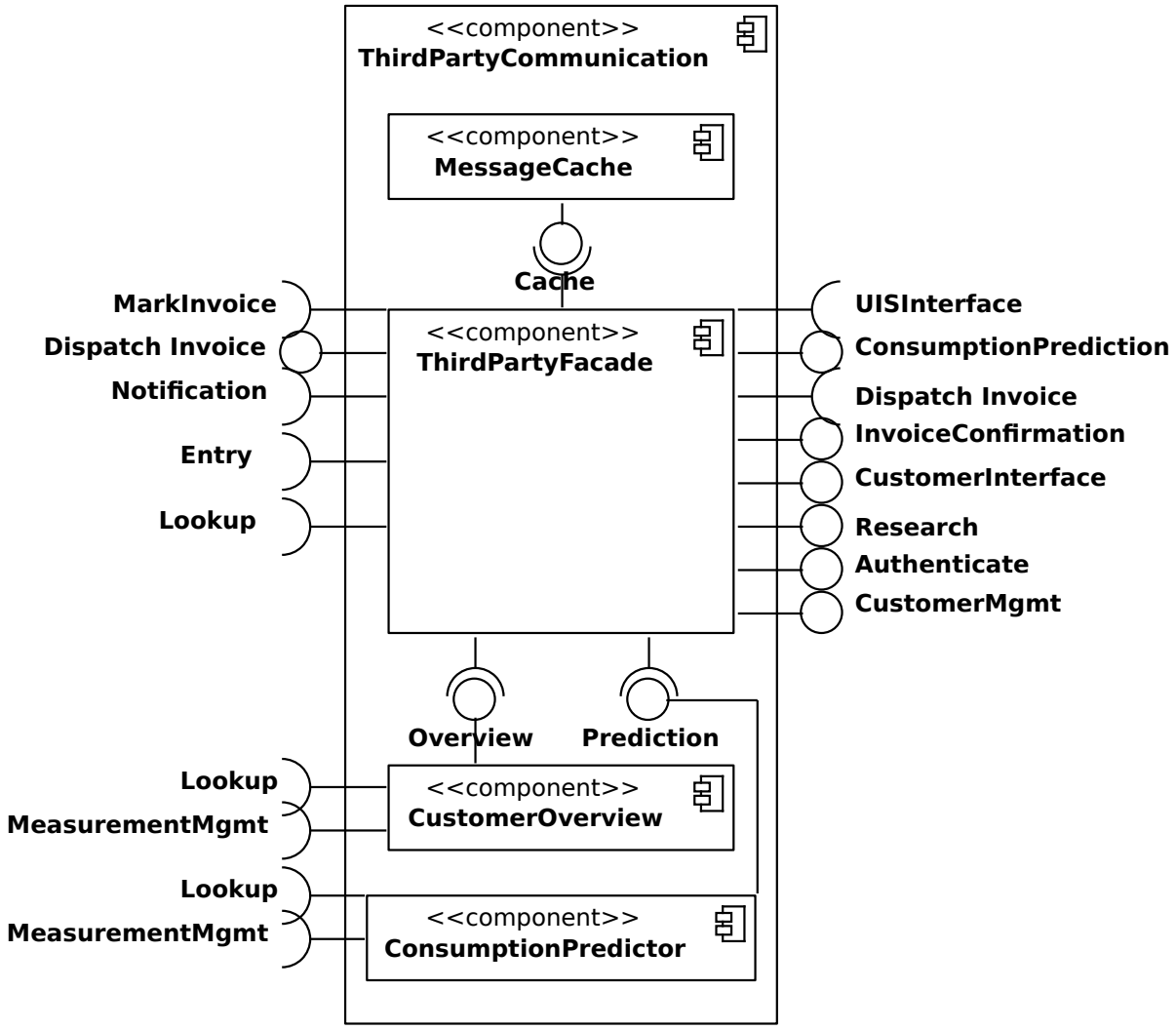

Figure 4: Decomposition of the `ThirdPartyCommunication`

4. The AnomalyScheduler can detect when it is necessary to go into overloaded modus.
5. Alarms that come from an RMM are checked for validity by the AlarmValidator.
6. High priority alarms from an RMM are sent to the AlarmHandler as well as the AnomalyDetector. When the alarm does not appear to be valid, the AlarmHandler is notified to cancel/undo the handling of the alarm.
7. The AlarmValidator can keep track of wich RMMs sends invalid alarms to *ReMeS* and can notify the operators when one of the RMMs ferquently sends false alarms.
8. The AlarmValidator can ask an RMM for more measurements via the RMMFacade.

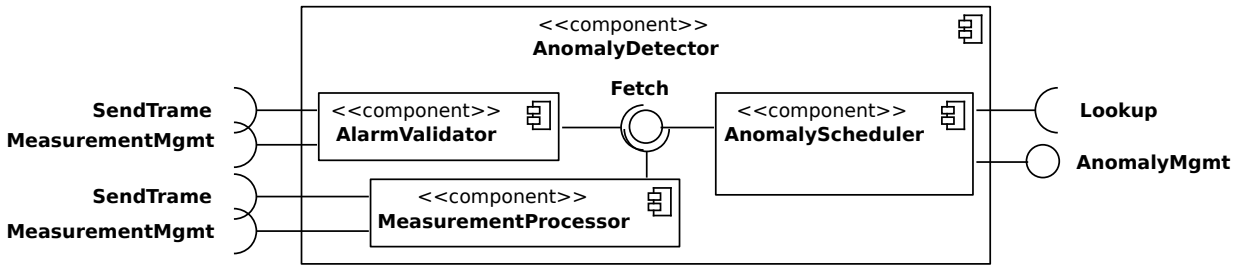

Figure 5: Decomposition of the `AnomalyDetector`

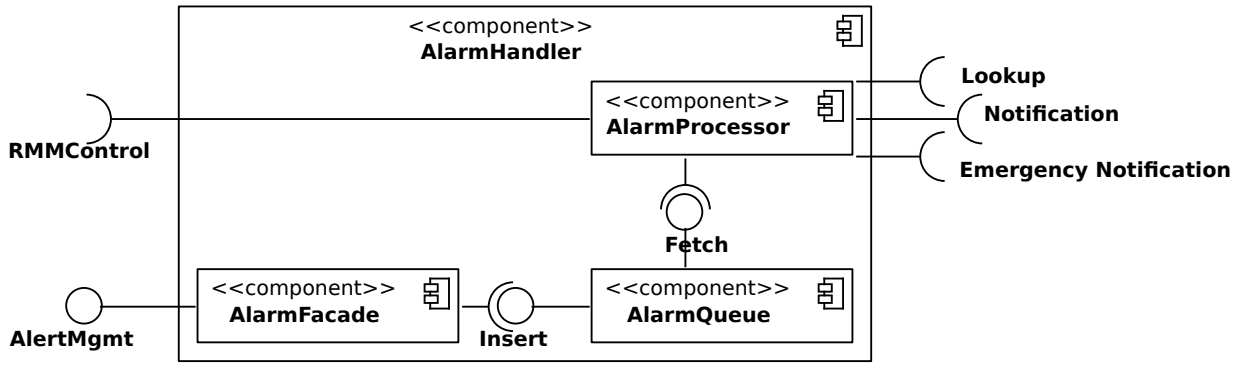

Figure 6: Decomposition of the `AlarmHandler`

Tactics and patterns used:

- Scheduler

Rationale In overloaded modus, it is required to schedule the processing by SLA, so an AnomalyScheduler is in order. Alarms that are generated by RMMs may not be reliable because RMMs do not have a lot of resources to ensure validity of alarms. When an RMM malfunctions and keeps sending invalid alarms, the AlarmValidator can notice this, notify operators via RMMCommunication and shut off the device if necessary. Some anomalies may not be recognised by an RMM, because more knowledge is required. The MeasurementProcessor can detect these anomalies.

Alternative for the AlarmValidator Assuming that RMMs send only valid alarms, an AlarmValidator is not necessary and will slow the alarm handling down. However, if an RMM malfunctions and floods *ReMeS* with alarm notifications, this may block everything else. The AlarmValidator avoids this.

4.1.4 *P3*: Requests to the measurement database

Design figure 7

1. The MeasurementManager receives all incoming requests and sends all outgoing replies to the appropriate recipient.
2. In normal modus, the RequestScheduler schedules the incoming requests in FIFO order.
3. If the system fails to comply to the specified deadlines, the MeasurementManager will notice this and activates the RequestScheduler's overloaded modus.
4. To make sure that requests can be scheduled approriately, the MeasurementManager adds the SLA to the request before handing it over to the RequestScheduler.
5. In overloaded modus, the RequestScheduler schedules requests in the order that returns the system to normal modus the fastest, giving precedence to premium service level customers and history queries that are used for anomaly detection.
6. In overloaded modus, the ReplicaManager may return a stale cached anwser to history queries.

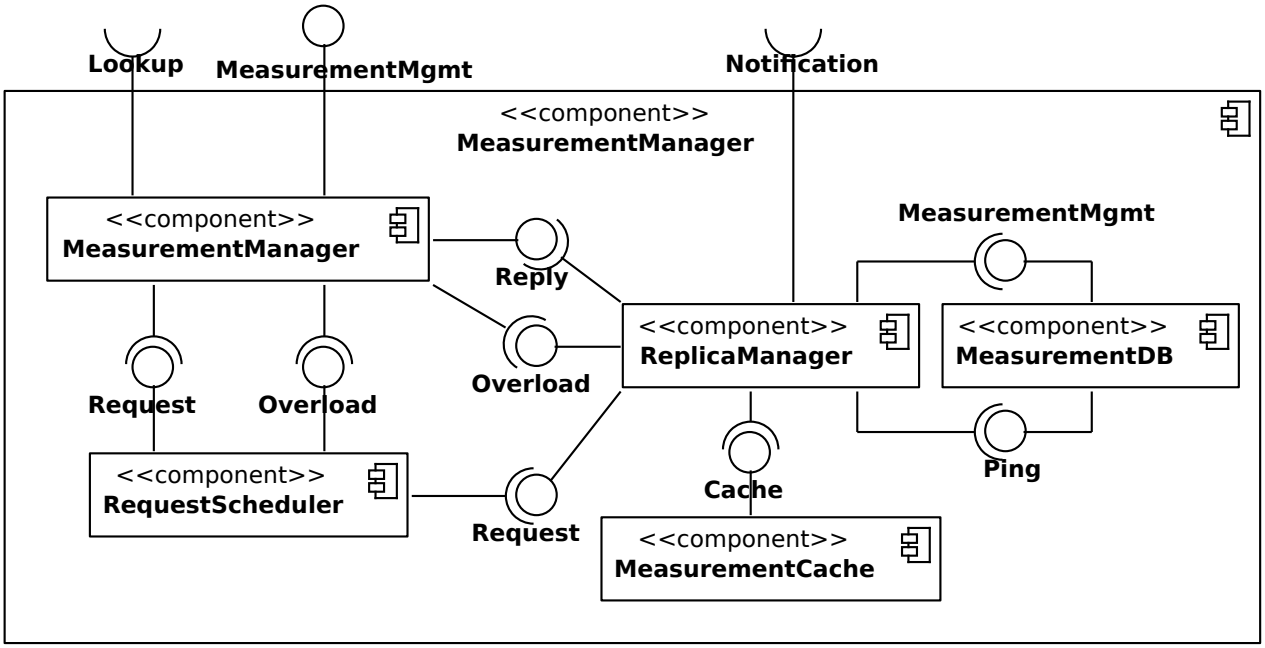

Figure 7: Decomposition of the `MeasurementDBManager`

Tactics and patterns used:

- Scheduler

Rationale The RequestScheduler is necessary to determine the order of requests in overloaded modus. To avoid that the RequestScheduler is bothered with outgoing replies, the MeasurementManager is added.

Alternative for the MeasurementManager The MeasurementManager is not strictly necessary: the RequestScheduler could send the replies to the appropriate recipients and measure the times to process requests to decide when to go into overloaded modus. However, the RequestScheduler should not be bothered with things other than scheduling.

4.1.5 *M2*: Fine-grained metering for enterprises

Design

1. The CustomerDB keeps track of who pays for and who receives payment for a utility measured by an RMM. If these are undefined the RMM is considered a secondary RMM.
2. The Billing component will only take into account consumption measured by primary RMMs.

Rationale The sole difference between primary and secondary RMMs is that only consumption measured by primary RMMs is billed. So only the Billing component has to know there is a difference.

4.1.6 *M3*: Decentralized Electricity Generation

Design

1. A customer who produces electricity should have two RMMs (this may be one device with two IDs): one for produced electricity and one for consumed electricity.
2. The CustomerDB knows who pays for and who receives payment for the utility measured by each RMM.
3. The Billing component uses this information to retrieve every RMM ID for which a customer should be billed. The Billing component has access to the MeasurementDBManager to retrieve the relevant measurements.

Rationale To be able to distinguish between consumed and produced electricty two RMM IDs are used. The Billing component should know who should be billed and who that person should pay, therefore two fields are stored in the CustomerDB. Keeping two fields increases flexibility: in the future it might be possible that person A pays person B, without any Utility Company in between or *ReMeS* might offer RMMs that measure things that don't need to be billed (e.g. temperature, chemicals...), a customer then 'pays himself'.

Alternatives for the two fields in the CustomerDB A trame could contain a field that indicates whether it measures consumption or production. However, this addition in functionality would require a change in the MeasurementDB schema and is less flexible than the current solution.

It is possible to store a Billing Type in the CustomerDB instead of two fields: positive billing (consumed), negative billing (produced) and no billing. However, this solution does not make it possible to let person A provide a utility to person B without the intervention of a Utility Company (e.g. a chemical plant that supplies another plant with its products).

5 Deployment view

Communication with RMM has to be simple (for the module) and allow for alternative channels (like SMS), therefore we chose UDP. Communication with most third parties relies on tcp for reliability, the communication with the users and possibly the callcenter would be based on an html interface, figure 8.

Figure 9, shows our deployment. There are three measurement database nodes, as long as each node has an availability of 90% or more, the total availability will be 99.9%. The CustomerOverview and Consumption predictions could be run on a seperate node but since we do not expect a high load from third party communication this is not yet necessary. Most components run on a node together with the other components they are closely related to, except for the availability monitor which needs to be on a separate node and the alarmhandler which is a critical subsystem relied on by multiple parts of *ReMeS*.

6 Scenarios

6.1 User profile creation

Customers can register in two ways: via the CallCenter and through the online interface. In the first case (fig. 10) a CallCenter Operator initiates the registration and asks the customer for the required information. When all required information is acquired the CallCenter Operator goes over the information with the customer to verify and then confirms the registration.

In the second case (fig. 11) the customer iniates the registration online and enters the required information which is stored by *ReMeS* as a tentative registration. A CallCenter Operator then goes over the information and approves the registration. Then the customer is sent a unique registration identifier via regular mail, which he can use to confirm the registration to *ReMeS*.

After confirmation the customer's profile is permanently stored in the CustomerDB and ready to be associated with devices. A technician is also scheduled to install one or more new devices.

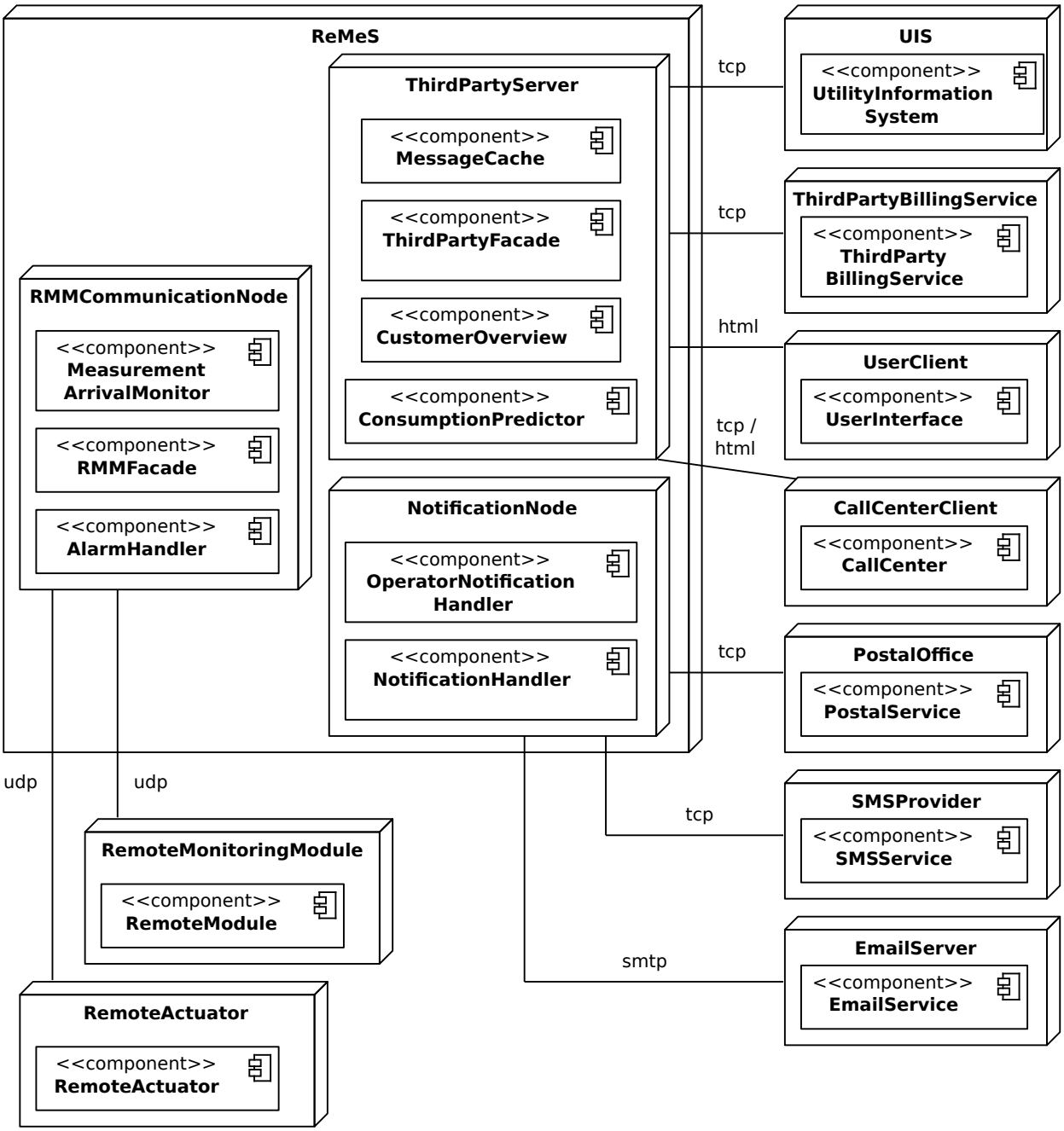

Figure 8: Context diagram for the deployment view.

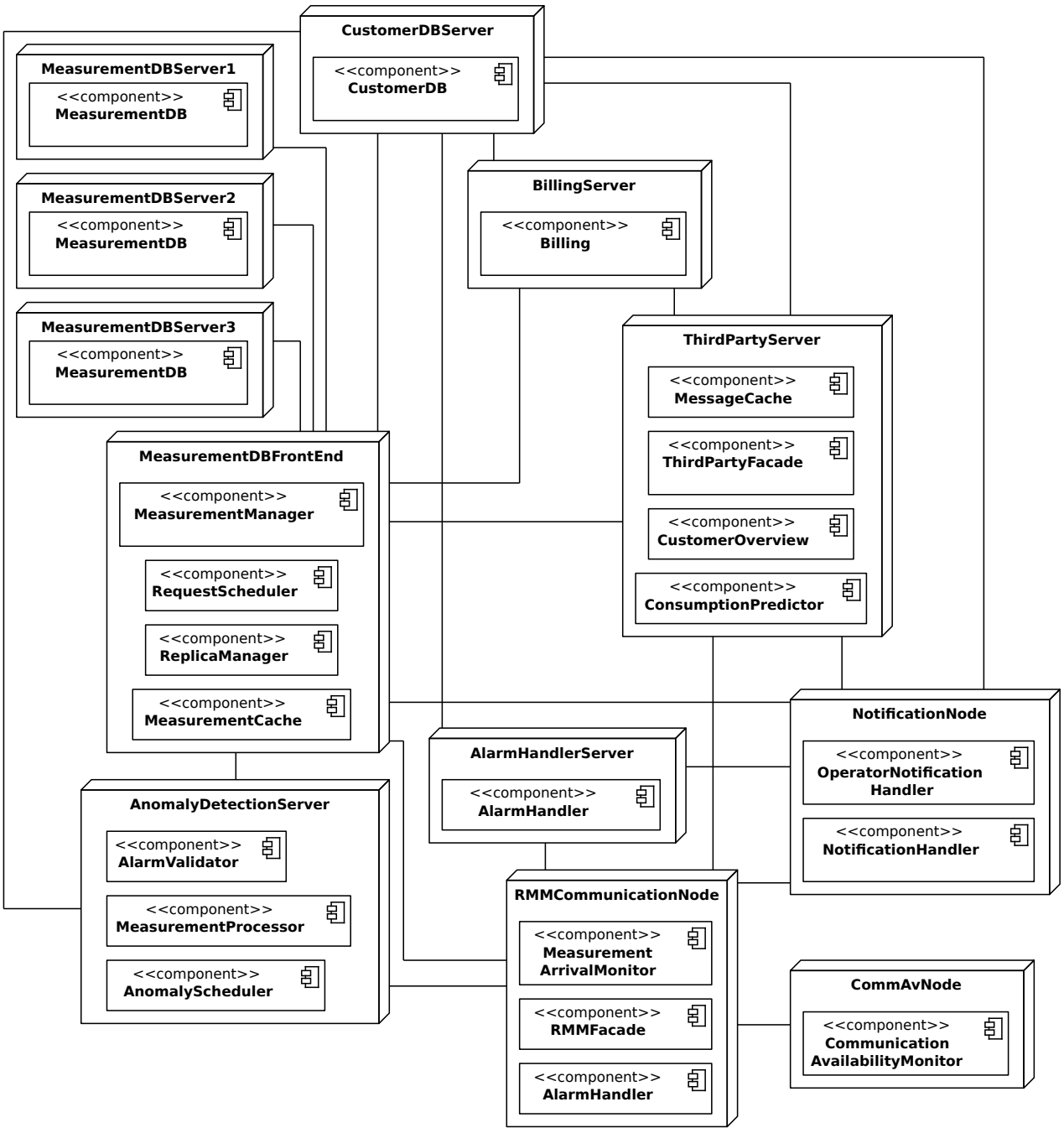

Figure 9: Diagram for the deployment view.

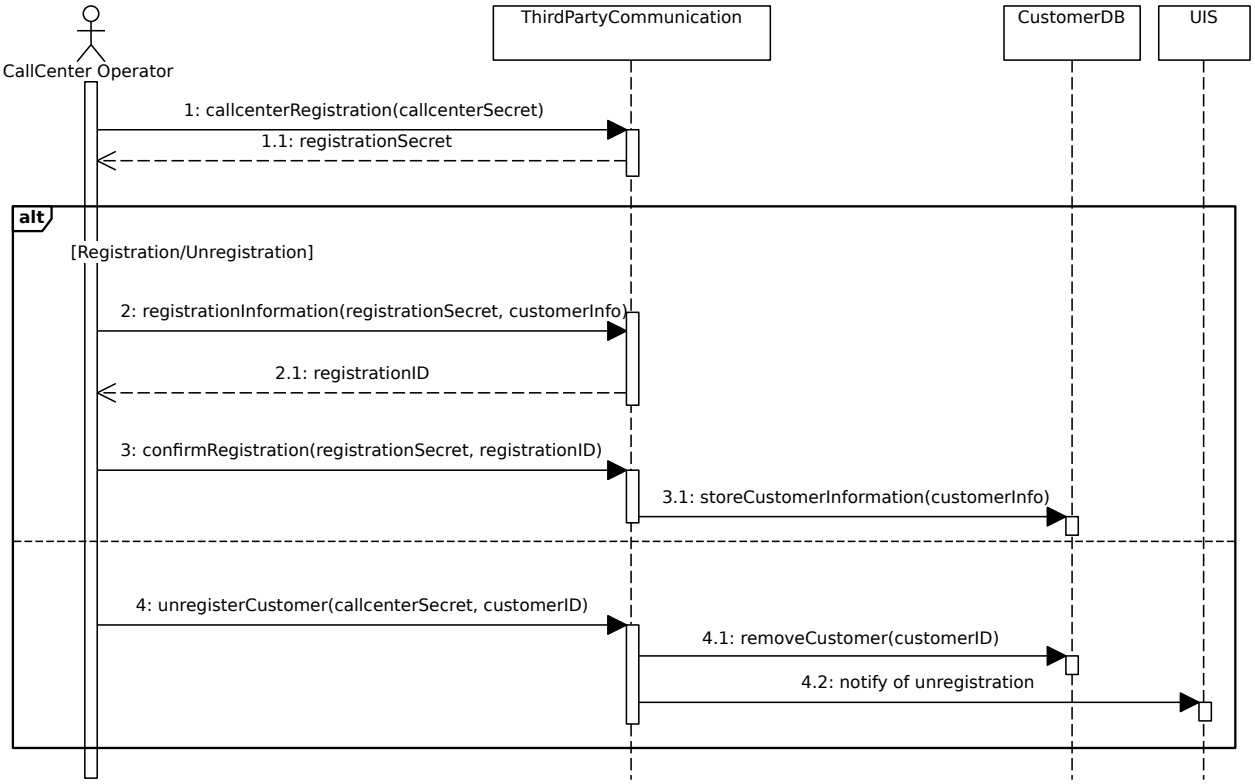

Figure 10: Customer registration through the CallCenter.

6.2 User profile association with a remote monitoring module

After installing a RMM the technician notifies *ReMeS*, *ReMeS* then updates the customer's information and notifies the UIS (fig. 12).

6.3 Transmission frequency reconfiguration

The customer indicates via the online interface that he wants to change the transmission frequency of one of his devices through the ThirdPartyCommunication. This change to the configuration is checked and then communicated to the RMM through the RMMCommunication.

6.4 Troubleshooting

When remote monitoring modules malfunction the *ReMeS* operators are notified, they try to resolve the problem and conctact the customer for onsite troubleshooting if necessary. If the customer indicates he cannot resolve the problem a technician is scheduled to repair/replace the device.

6.5 Alarm notification recipient configuration

The customer accesses his overview by logging in to the online interface. Here he can add, remove and edit alarm recipients for each of his devices. Each recipient is asked to confirm their willingness to receive alarm notifications. The ThirdPartyCommunication then updates the DeviceInformation stored in the CustomerDB.

6.6 Remote control

After logging in to the online interface a customer has the options to control a device directly or to alter the device's configuration which may cause the device to actuate the corresponding actuator.

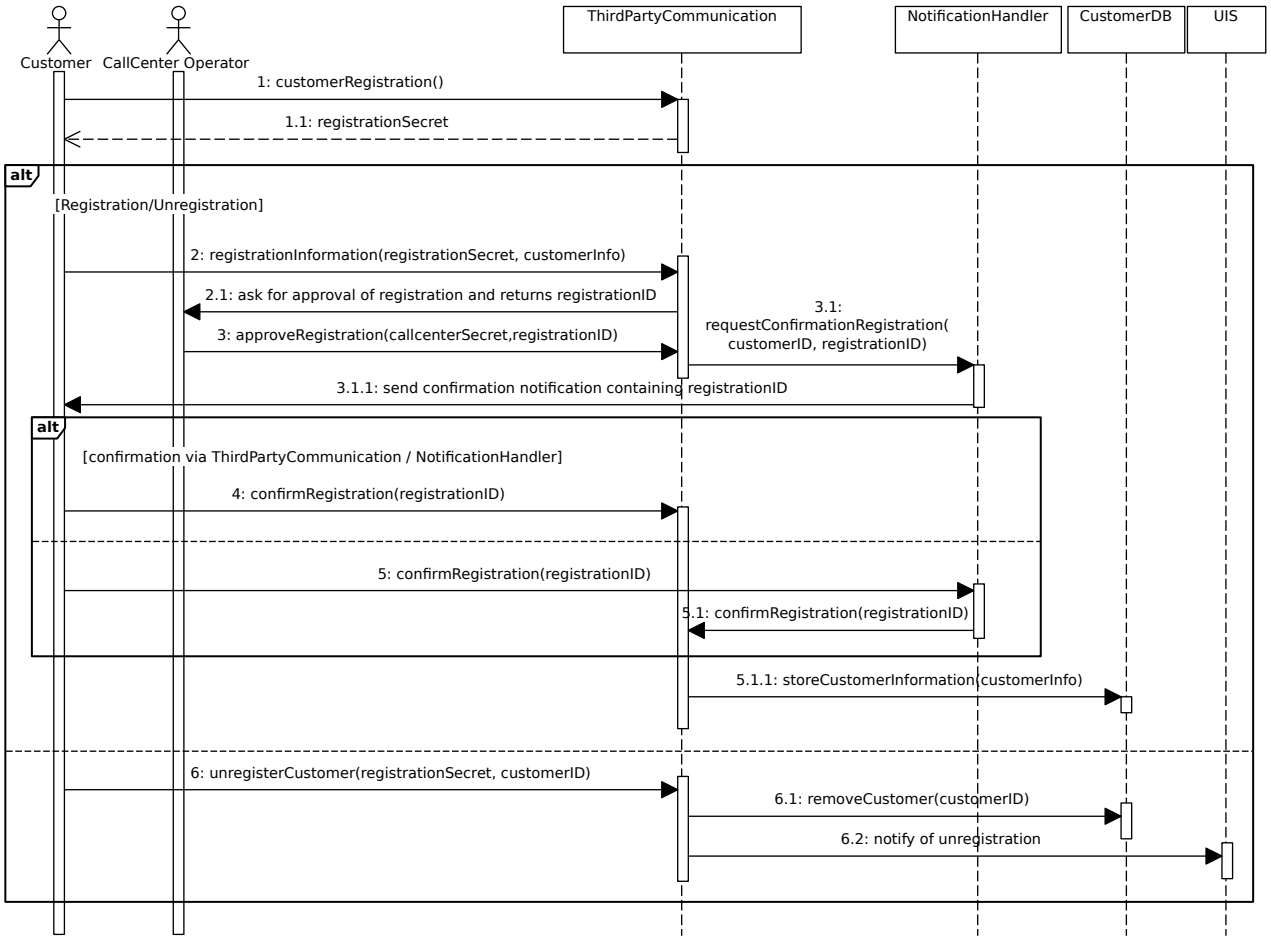

Figure 11: Customer registration through the online interface.

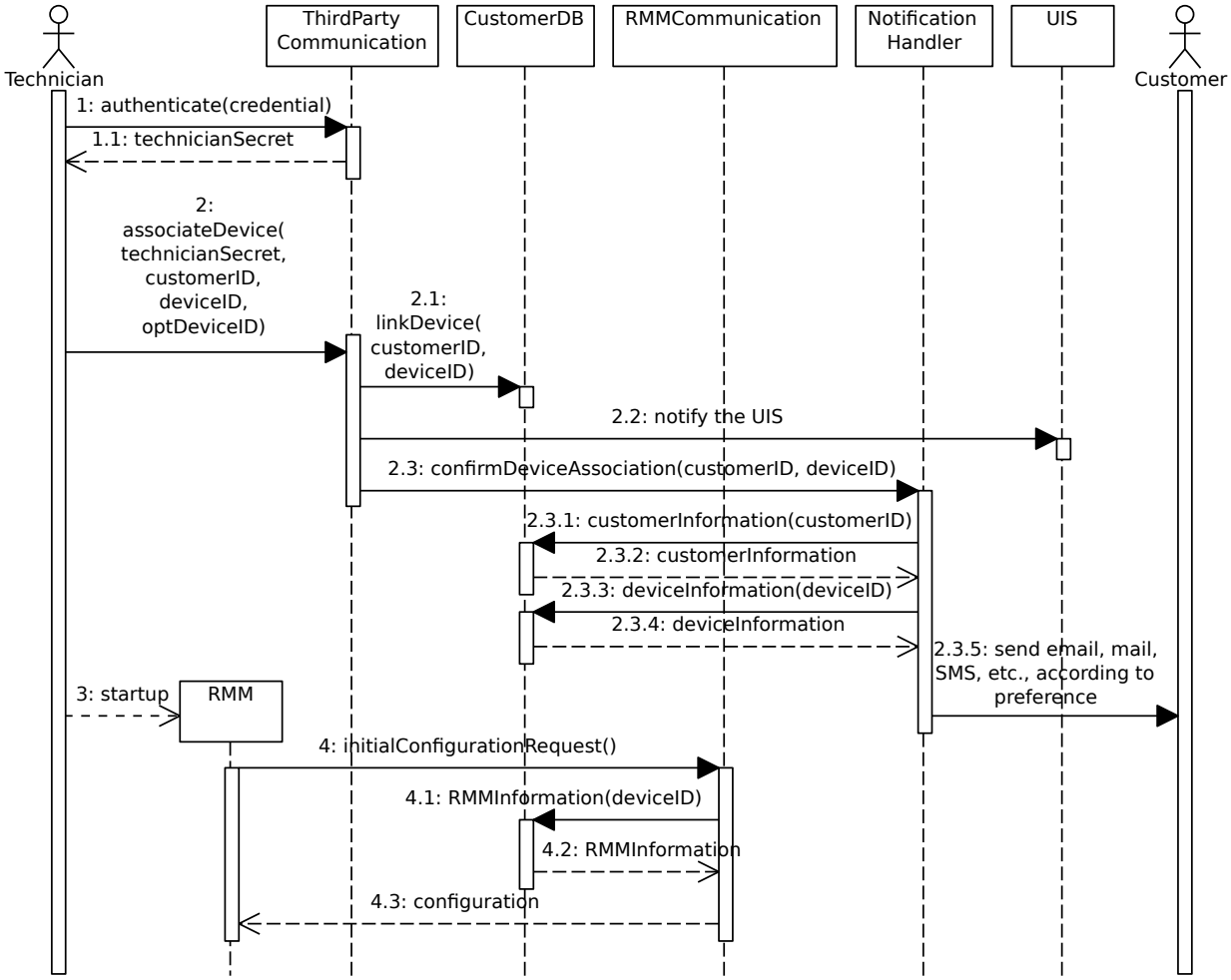

Figure 12: Device association with a customer.

6.7 Normal measurement data transmission

The RMM sends a measurement to *ReMeS* through the RMMFacade (fig. 13). The RMMFacade forwards this measurement to the MeasurementDBManager, the MeasurementArrivalMonitor and the AnomalyDetector where it is checked for anomalies (cf. Alarm data transmission: *ReMeS*).

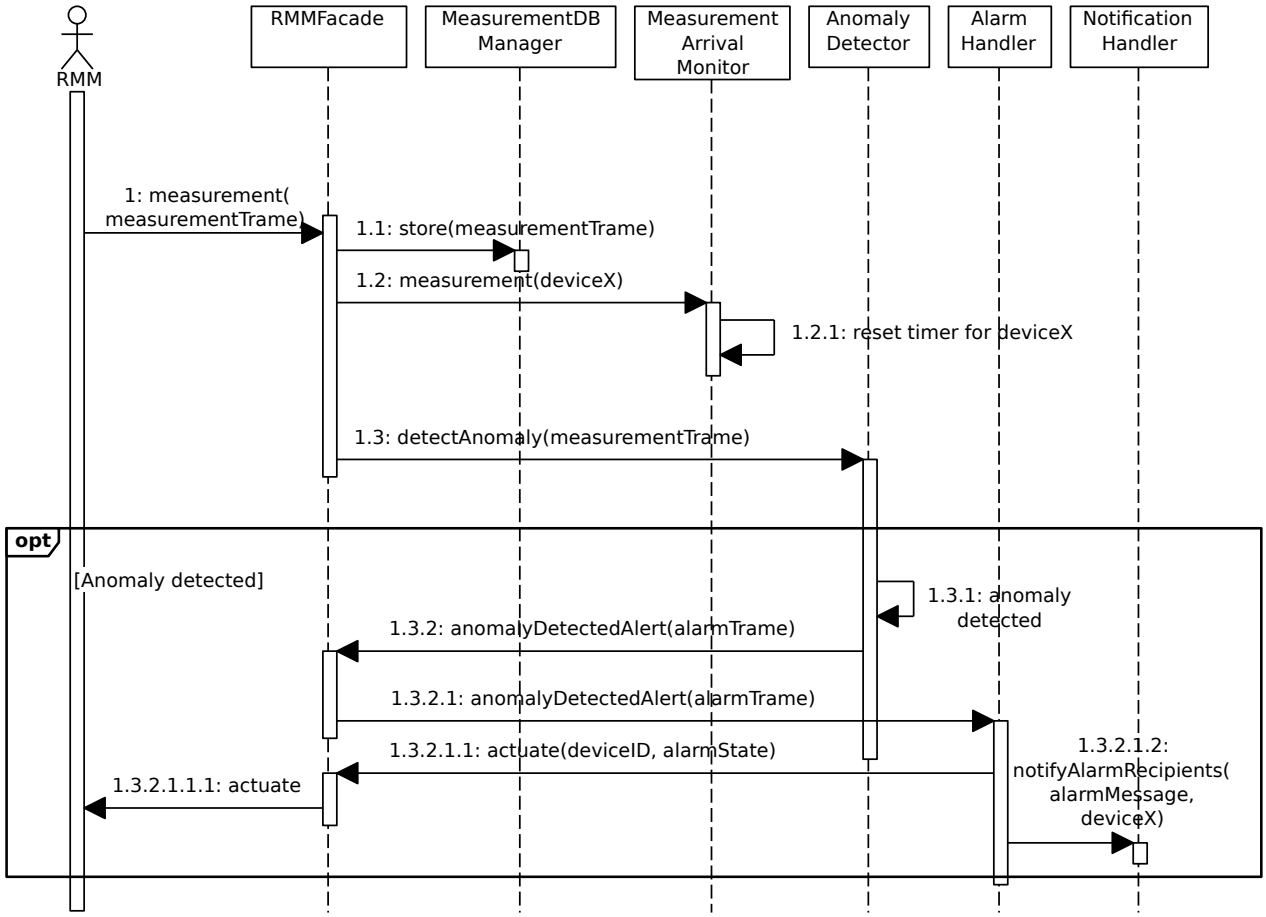

Figure 13: Incoming Measurement.

When a measurement is lost or an RMM does not send a measurement (fig. 14), this is noticed by the MeasurementArrivalMonitor which notifies the *ReMeS* operators and the alarm recipient of the problem.

6.8 Individual data analysis

All incoming measurements are forwarded to the AnomalyDetector (fig. 13) to be compared to historical consumption so as to detect anomalous measurements.

6.9 Utility production planning analysis

The ConsumptionPredictor regularly analyses the stored measurements to be able to forecast future demand. The Utility Information System can access predictions for his customer base through the ThirdPartyCommunication.

6.10 Information exchange towards the UIS

ReMeS utilizes the UISInterface to request current pricing information and to notify the utility company of new customers or customers who have cancelled their subscription.

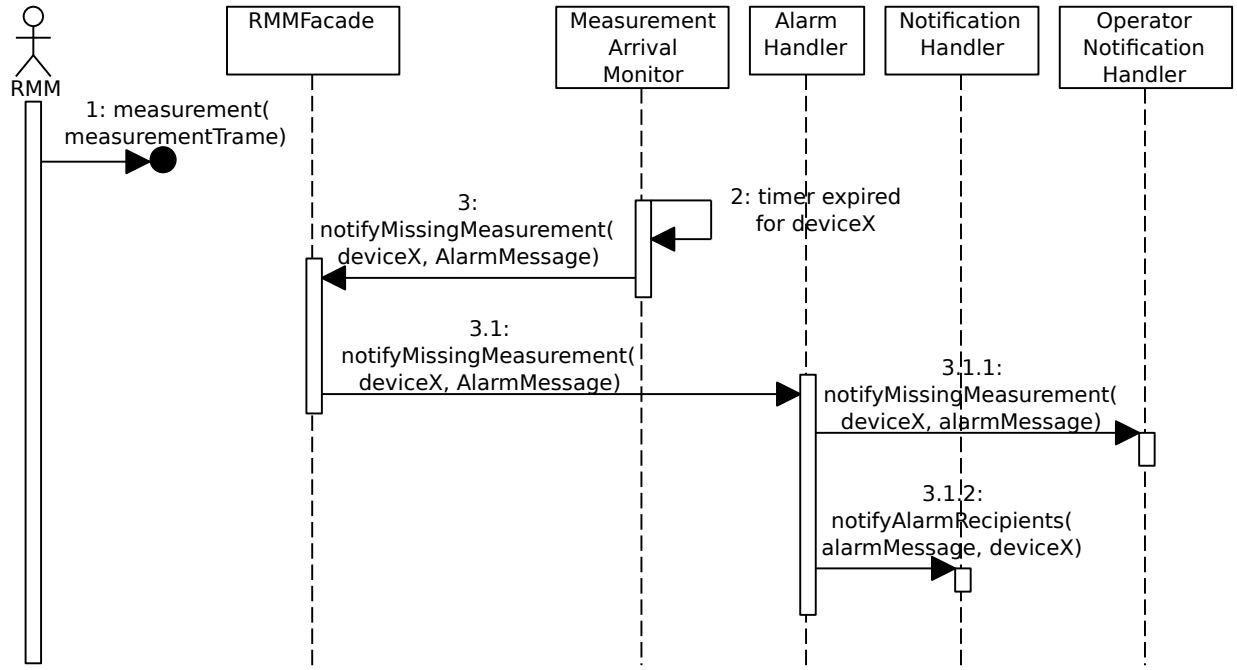

Figure 14: Missing measurement.

6.11 Alarm data transmission: remote monitoring module

When an RMM detects an anomaly it sends an AlarmTrame to the RMMFacade (fig. 15). How an alarm is handled depends on its priority. High priority alarms, i.e. gas alarms, are immediately sent to the AlarmHandler which resolves the alarm by actuating appropriate actuators and notifying alarm recipients and the Emergency Services. Furthermore the alarm is also sent to the AnomalyDetector which can request further measurements from the RMM to validate the alarm. If the alarm turns out not to be valid the AlarmHandler is requested, through the RMMFacade, to cancel the alarm by cancelling the handling of the alarm, sending a notfication that the alarm was a false alarm and restoring the state of all actuators that were influenced by the alarm.

Low priority alarms are first sent to the AnomalyDetector for validation. If the alarm is considered valid the AnomalyDetector will contact the AlarmHandler through the RMMFacade. The AlarmHandler actuates the appropriate actuators and notifies all alarm recipients.

6.12 Alarm data transmission: *ReMeS*

The AnomalyDetector detects an anomaly in the measurements (fig. 13) of an RMM that was not noticed by the RMM itself. It notifies the AlarmHandler through the RMMFacade. The AlarmHandler handles the alarm appropriately: in accordance with the customers profile.

6.13 Low battery alarm

The RMM sends a trame containing its PowerStatus. The RMMFacade passes this to the AlarmHandler which passes it to the NotificationHandler to be delivered to the customer. The customer can then replace the battery or contact the CallCenter to troubleshoot his device.

6.14 Remote control module de-activation

The customer indicates he wants to de-activate a device, the device is then reconfigured not to send any more measurements and is removed from the customer's account.

6.15 New bill creation

The Billing component decides an Invoice should be generated for a certain customer, it then contacts the ThirdPartyBillingService and the UIS through the ThirdPartyCommunication.

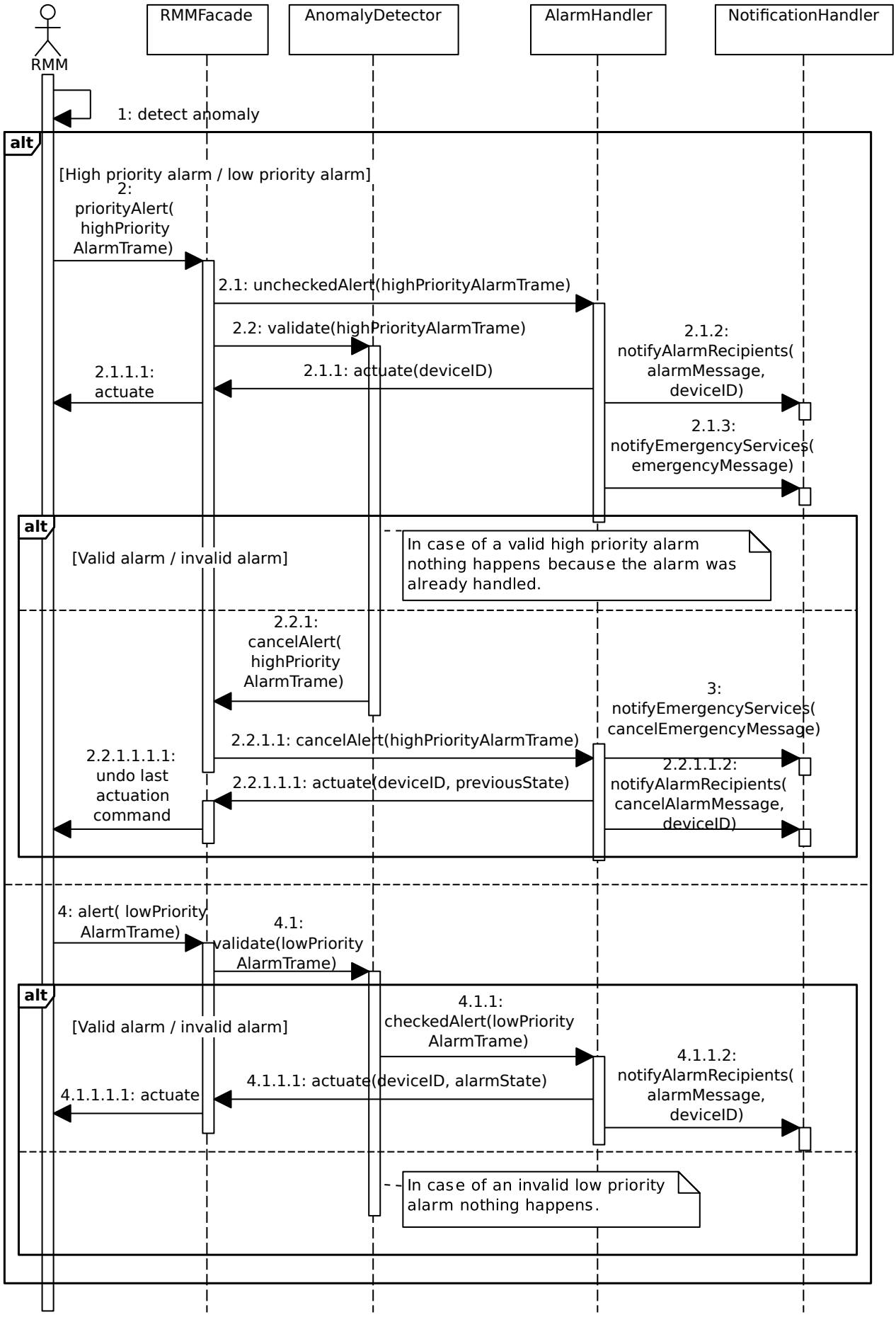

Figure 15: An alarm from an RMM reaches *ReMeS*.

6.16 Bill payment is received

The ThirdPartyBillingService contacs *ReMeS* via the ThirdPartyCommunication which tells the Billing component to mark the invoice paid.

A Element catalog

A.1 AnomalyDetector

Responsabilities The AnomalyDetector

- Detects anomalies in the measurements;
- Validates alarms from RMMs;
- Detects when an RMM keeps sending invalid alarms;

Super component *ReMeS*

Sub components AlarmValidator, AnomalyScheduler, MeasurementProcessor

Provided interfaces

- AnomalyMgmt

A.2 AnomalyScheduler

Responsabilities The AnomalyScheduler

- Schedules incoming anomaly detection and validation request;
- Determines when to go to overload modus;

Super component AnomalyDetector

Sub components None

Provided interfaces

- AnomalyMgmt
  - `void detectAnomaly(MeasurementTrame measurement)`
    - Effect: The measurement is checked upon anomalies.
    - Exceptions: None
  - `void validate(AlarmTrame alarm)`
    - Effect: The alarm is validated.
    - Exceptions: None
- Fetch
  - `MeasurementTrame fetchMeasurement()`
    - Effect: A measurement is returned.
    - Excsptions: None
  - `AlarmTrame fetchAlarm()`
    - Effect: An alarm is returned.
    - Excsptions: None

A.3 AlarmFacade

cf. assignment

A.4 AlarmHandler

Responsabilities The AlarmHandler

- Receives incoming alarms;
- Notifies the NotificiationHandler when nessecary;
- Activates the actuators when nessecary;

Super component RMMCommunication

Sub components None

Provided interfaces

- AlertMgmt
  - `void anomalyDetectedAlert(AlarmTrame alarm`
    - Effect: The alarm is handled by actuating appropriate actuators and notifying the NotificationHandler. This method is called for anomalies detected by the AnomalyDetector.
    - Exceptions: None
  - `void cancelAlert(AlarmTrame highPriorityAlarm)`
    - Effect: The alarm is canceled. This happens when the AlarmHandler was asked to handle a high priority alert that was later noticed to be invalid by the AnomalyDetector.
    - Exceptions: None
  - `void checkedAlert(AlarmTrame lowPriorityAlarm)`
    - Effect: The alarm is acknowledged and handled. This method is called after low priority alarms are validated by the AnomalyDetector.
    - Exceptions: None
  - `void lowBatteryAlert(DeviceID device, PowerStatus status) throws NoSuchDeviceException`
    - Effect: The customer is notified of the low power status of the RMM.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given device does not exist.
  - `void notifyDeviceMalfunction(DeviceID device, AlarmMessage alarm) throws NoSuchDeviceExceptio`
    - Effect: A *ReMeS* operator is notified of a malfunctioning device, e.g. many false alarms, sporadic communication, etc.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given device does not exist.
  - `void notifyMissingMeasurement(DeviceID device, AlarmMessage alarm) throws NoSuchDeviceExcepti`
    - Effect: A *ReMeS* operator is notified of a missing measurement.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given device does not exist.
  - `void uncheckedAlert(AlarmTrame highPriorityAlarm)`
    - Effect: The alarm is handled by actuating appropriate actuators and notifying the NotificationHandler. This method is called for urgent, unchecked alerts.
    - Exceptions: None
- Ping

A.5 AlarmProcessor

cf. assignment

A.6 AlarmQueue

cf. assignment

A.7 AlarmValidator

Responsabilities The AlarmValidator

- Checks the validity of alarms from RMMs;
- If a lot of invalid alarms from a certain RMM arrive, the operators are notified;

Super component AnomalyDetector

Sub components None

A.8 Billing

Responsabilities The Billing component

- Generates invoices, determining its schedule on the billing information of customers;
- Saves all sent invoices (in an internal database);

Super component *ReMeS*

Sub components None

Provided interface

- MarkInvoice
  - `void markInvoicePaid(InvoiceID invoice) throws NoSuchInvoiceException`
    - Effect: Marks the invoice as paid.
    - Exceptions:
      - NoSuchInvoiceException: Thrown when the given invoice does not exist.

A.9 CommunicationAvailabilityMonitor

Responsabilities The CommunicationAvailabilityMonitor

- Monitors the availability of the RMMCommunication subsystem components with pings;
- Notifies the operators when one of the internal components fails;

Super component RMMCommunication

Sub components None

A.10 ConsumptionPredictor

Responsabilities The ConsumptionPredictor

- Analyzes utility consumption patterns;
- Forecasts future consumption patterns;

Super component ThirdPartyCommunication

Sub components None

Provided interfaces

- Prediction
  - `ConsumptionPrediction requestConsumptionPrediction(TimePeriod timePeriod) throws NoPrediction`
    - Effect: Calculates the consumption prediction.
    - Exception:
      - NoPredictionPossibleException: Throws when the request cannot be fulfilled.

A.11 CustomerDB

Responsabilities The CustomerDB

- Stores all customer information;
- Stores which RMMs are associated to customers;
- Stores who pays for the measurements of each RMM and who receives payment;
- Stores the information relevant to billing;
- Stores the SLAs for each customer;

Super component *ReMeS*

Sub components None

Provided interfaces

- Entry
  - `void linkDevice(CustomerID customer, DeviceID device) throws NoSuchCustomerException, NoSuchDeviceException`
    - Effect: After a sanity check the given device and customer are linked.
    - Exception:
      - NoSuchCustomerException: Thrown when the given CustomerID is not associated with a customer.
      - NoSuchDeviceException: Thrown when the given RMM does not exist.
  - `void removeCustomer(CustomerID customer) throws NoSuchCustomerException`
    - Effect: The given customer is marked for removal.
    - Exception:
      - NoSuchCustomerException: Thrown when the given CustomerID is not associated with a customer.
  - `void removeDevice(DeviceID device) throws NoSuchDeviceException`
    - Effect: The given device is removed from the customer's account.
    - Exception:
      - NoSuchDeviceException: Thrown when the given RMM does not exist.
  - `void storeCustomerInformation(CustomerInformation customerInfo)`
    - Effect: The information is stored for the corresponding customer.
    - Exceptions: None
  - `void storeDeviceInformation(DeviceInformation deviceInfo)`
    - Effect: The information is stored for the corresponding RMM.

    * Exceptions: None

- Lookup
  - `Collection<DeviceID> associatedDevices(CustomerID customer) throws NoSuchCustomerException`
    * Effect: The DeviceIDs of devices associated to the given customer are returned.
    * Exceptions:
      · NoSuchCustomerException: Thrown when the given customer does not exist.
  - `Collection<DeviceID> billedDevices(CustomerID customer) throws NoSuchCustomerException`
    * Effect: The DeviceIDs for which the given customer should pay are returned.
    * Exceptions:
      · NoSuchCustomerException: Thrown when the given customer does not exist.
  - `CustomerInformation customerInformation(CustomerID customer) throws NoSuchCustomerException`
    * Effect: The information concerning the RMM is returned.
    * Exceptions:
      · NoSuchCustomerException: Thrown when the given customer does not exist.
  - `DeviceInformation deviceInformation(DeviceID device) throws NoSuchRMMException`
    * Effect: The information concerning the RMM is returned. This information includes the type of RMM, the alarmrecipients and the actuator that should handle alarms for this RMM.
    * Exceptions:
      · NoSuchDeviceException: Thrown when the given RMM does not exist.
  - `SLA sla(CustomerID customer) throws NoSuchCustomerException`
    * Effect: The SLA of the given customer is returned.
    * Exceptions:
      · NoSuchCustomerException: Thrown when the given customer does not exist.

A.12 CustomerOverview

Responsabilities The CustomerOverview

- Handles generation of customer overviews;

Super component ThirdPartyCommunication

Sub components None

Provided interfaces

- Overview
  - `CustomerOverview generateOverview(CustomerID customer) throws NoSuchCustomerException`
    * Effect: An overview of the customer's consumption, contact information for notifications, parameters for anomaly detection, etc., is returned.
    * Exceptions:
      · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
      · NoSuchCustomerException: Thrown when the given customer does not exist.

A.13 MeasurementArrivalMonitor

Responsabilities The MeasurementArrivalMonitor

- Monitors for missing measurements;
- Notifies the operators when three consecutive measurements do not arrive in time;
- Keeps track of how long a failure of the communication channel remains;

Super component RMMCommunication

Sub components None

Provided interfaces

- Ping
- Measurement Arrival
  - `void configurationUpdate(DeviceID device, DeviceConfiguration configuration) throws NoSuchDeviceException`
    - Effect: The timer for this device is altered according to the new configuration.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given RMM does not exist.
  - `void measurement(DeviceID device) throws NoSuchDeviceException`
    - Effect: The timer that was waiting for this device to send a measurement is reset.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given RMM does not exist.

A.14 MeasurementCache

Responsabilities The MeasurementCache

- Caches replies to queries on the MeasurementDB;

Super component MeasurementDBManager

Sub components None

Provided interfaces

- Cache
  - `void store(MeasurementQuery query, MeasurementReply reply)`
    - Effect: The query and reply are stored in the cache.
    - Exceptions: None
  - `MeasurementReply request(MeasurementQuery query) throws CacheMissException`
    - Effect: The reply that belongs to the query is returned.
    - Exceptions:
      - CacheMissException: Thrown when the reply to the given query is not stored in the cache.
- Overload
  - `void setOverloadedMode()`
    - Effect: The ReplicaManager goes into overloaded modus: it is now allowed to return stale cached information.
    - Exceptions: None
  - `void setNormalMode()`
    - Effect: The ReplicaManager goes into normal modus: the cache can only be consulted when the information is not stale.
    - Exceptions: None

A.15 MeasurementDB

cf. assignment

A.16 MeasurementDBManager

Responsabilities The MeasurementDBManager

- Provides access to the MeasurementDBs;
- Manages the replicas of the MeasurementDBs;
- Notifies the operators when one of the MeasurementDBs fails;

Super component *ReMeS*

Sub components MeasurementCache, MeasurementManager, ReplicaManager, RequestScheduler

Provided interfaces

- MeasurementMgmt

A.17 MeasurementManager

Responsabilities The MeasurementManager

- Provides a single point of communication for requests/replies to/from the MeasurementDB;
- Decides when to go to overloaded modus by monitoring the time needed for handling requests;

Super component MeasurementDBManager

Sub components None

Provided interfaces

- MeasurementMgmt
  - `MeasurementReply request(MeasurementQuery query) throws InvalidQueryException`
    - Effect: Looks up the SLA for the query and hands it to the RequestScheduler. The origin of the query is cached until a MeasurementReply arrives that can be passed to the originator of the query.
    - Exceptions:
      - InvalidQueryException: Thrown when the given query is invalid.
  - `void store(MeasurementTrame measurement)`
    - Effect: Looks up the SLA for the query and hands it to the RequestScheduler.
    - Exceptions: None
- Reply
  - `void reply(MeasurementReply reply, MeasurementQuery query)`
    - Effect: Communicates the reply to the appropriate recipients.
    - Exceptions: None

A.18 MeasurementProcessor

Responsabilities The MeasurementProcessor

- Detects anomalies in incoming measurements by comparing them to history data.

Super component AnomalyDetector

Sub components None

A.19 MessageCache

Responsabilities The MessageCache

- Caches unacknowledged messages;

Super component ThirdPartyCommunication

Sub components None

Provided interfaces

- Cache
  - `void cache(Message message)`
    - Effect: The message is cached.
    - Exceptions: None
  - `Message fetch(MessageID message)`
    - Effect: The message is fetched from the cache.
    - Exceptions: None
  - `void ack(MessageID message)`
    - Effect: The message is marked for removal from the cache because reception has been acknowledged.
    - Exceptions: None

A.20 NotificationHandler

Responsabilities The NotificationHandler

- Handles all notifications to and from third parties via alternative communication channels;

Super component *ReMeS*

Sub components None

Provided interfaces

- Confirmation
  - `void confirmAlarmRecipientChange(CustomerID customer, AlarmRecipient recipient) throws NoSuchCustomerException`
    - Effect: The customer is notified that his alarm recipient agrees to receive alarms for him.
    - Exceptions:
      - NoSuchCustomerException: Thrown when the given customer does not exist.
  - `void confirmDeviceAssociation(CustomerID customer, DeviceID device) throws NoSuchCustomerExce NoSuchDeviceException`
    - Effect: The customer is notified that a new RMM has been associated to his account.
    - Exceptions:
      - NoSuchCustomerException: Thrown when the given customer does not exist.
      - NoSuchDeviceException: Thrown when the given device does not exist.
  - `void requestConfirmationAlarmRecipientAppointment(CustomerID customer, AlarmRecipient recipient) throws NoSuchCustomerException`
    - Effect: The AlarmRecipient is asked to confirm that he is willing to receive alarms (and requests for confimation of alarms) for the given customer.
    - Exceptions:

        - NoSuchCustomerException: Thrown when the given customer does not exist.
    - `void requestConfirmationRegistration(CustomerID customer, RegistrationID registration) throws NoSuchCustomerException`
        - Effect: The customer is asked for a confirmation of his registration.
        - Exceptions:
            - NoSuchCustomerException: Thrown when the given customer does not exist.

- EmergencyNotification
    - `void notifyAlarmRecipients(AlarmMessage alarm, DeviceID device)`
        - Effect: The alarm recipients are looked up in de CustomerDB and notified of the alarm.
        - Exceptions: None
    - `void notifyLowBatteryStatus(AlarmMessage alarm, DeviceID device) throws NoSuchDeviceException`
        - Effect: The customer is notified of the low power status of the RMM.
        - Exceptions:
            - NoSuchDeviceException: Thrown when the given device does not exist.
    - `void notifyEmergencyServices(EmergencyMessage alarm)`
        - Effect: The emergency services are notified of the alarm.
        - Exceptions: None

- InboundCommunication
    - `void confirmAlarmRecipientAppointment(AlarmRecipientAppointmentID araID) throws NoSuchRegistr`
        - Effect: The registration corresponding to the RegistrationID is confirmed.
        - Exceptions:
            - NoSuchRegistrationException: Thrown when no registration with a matching ID is in progress.
    - `void confirmRegistration(RegistrationID registration) throws NoSuchRegistrationException`
        - Effect: The registration corresponding to the RegistrationID is confirmed.
        - Exceptions:
            - NoSuchRegistrationException: Thrown when no registration with a matching ID is in progress.

A.21 OperatorNotificationHandler

Responsabilities The OperatorNotificationHandler

- Notifies the *ReMeS* operators;

Super component *ReMeS*

Sub components None

Provided interfaces

- Notification
    - `void notifyDeviceMalfunction(DeviceID device, AlarmMessage alarm) throws NoSuchDeviceExceptio`
        - Effect: A *ReMeS* operator is notified of a malfunctioning device, e.g. many false alarms, sporadic communication, etc.
        - Exceptions:
            - NoSuchDeviceException: Thrown when the given device does not exist.
    - `void notifyMissingMeasurement(DeviceID device, AlarmMessage alarm) throws NoSuchDeviceExcepti`
        - Effect: A *ReMeS* operator is notified of a missing measurement.

        * Exceptions:
            · NoSuchDeviceException: Thrown when the given device does not exist.
    - `void notifyUnavailableComponent(SubsystemID subsystem, AlarmMessage alarm) throws NoSuchSubsy`
        * Effect: A *ReMeS* operator is notified of an availability problem in the subsystem.
        * Exceptions:
            · NoSuchSubsystemException: Thrown when the given subsystem does not exist.
    - `void notifyUnavailableThirdParty(ThirdPartyID thirdParty, AlarmMessage alarm) throws NoSuchThirdPartyException`
        * Effect: A *ReMeS* operator is notified of an availability problem with a third party.
        * Exceptions:
            · NoSuchThirdPartyException: Thrown when the given third party does not exist.

A.22 ReplicaManager

Responsabilities The ReplicaManager

- Manages active replication of MeasurementDBs;
- Provides communication with the MeasurementDBs;
- Notifies the operator when one of the databases fails;

Super component MeasurementDBManager

Sub components None

Provided interfaces

- Request
    - `void query(MeasurementQuery query, SLA sla) throws InvalidQueryException`
        * Effect: The query is forwarded to one or more MeasurementDBs.
        * Exceptions:
            · InvalidQueryException: Thrown when the given query is invalid.
- Overload
    - `void setOverloadedMode()`
        * Effect: The ReplicaManager goes into overloaded modus: it is now allowed to return stale cached information.
        * Exceptions: None
    - `void setNormalMode()`
        * Effect: The ReplicaManager goes into normal modus: the cache can only be consulted when the information is not stale.
        * Exceptions: None

A.23 RequestScheduler

Responsabilities The RequestScheduler

- Schedules the requests according to the currently active mode;

Super component MeasurementDBManager

Sub components None

Provided interfaces

- Request
  - `void query(MeasurementQuery query, SLA sla) throws InvalidQueryException`
    - Effect: The query is scheduled.
    - Exceptions:
      - InvalidQueryException: Thrown when the given query is invalid.
- Overload
  - `void setOverloadedMode()`
    - Effect: The RequestScheduler goes into overloaded modus and alters its scheduling strategy approriately.
    - Exceptions: None
  - `void setNormalMode()`
    - Effect: The RequestScheduler goes into normal modus and alters its scheduling strategy approriately.
    - Exceptions: None

A.24 RMMCommunication

Responsabilities The RMMCommunication

- Provides all communication with the RMMs;
- Acknowledges all incoming trames;
- Monitors for missing measurements;
- Notifies the operators when three consecutive measurements do not arrive in time;
- Notifies the operators when one of the internal components fails;
- Keeps track of how long a failure of the communication channel remains;
- Handles alarms;

Super component *ReMeS*

Sub components Alarm handler, CommunicationAvailabilityMonitor, MeasurementArrivalMonitor

Provided interfaces

- Send Trame
- AlertMgmt

A.25 RMMFacade

Responsabilities The RMMFacade

- Acknowledges the incoming trames;
- Forwards the incoming measurements to the MeasurementArrivalMonitor, the AnomalyDetector and the MeasurementDBManager;
- Forwards the incoming alarms to the AlarmHandler;

Super component RMMCommunication

Sub components None

Provided interfaces

- AlertMgmt cf. AlarmHandler
- Ping
- RMMControl
  - `void actuate(DeviceID device, ActuatorState state) throws NoSuchDeviceException`
    - Effect: The given device is actuated in accordance with the required state.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given device does not exist.
  - `void requestMeasurements(DeviceID device) throws NoSuchDeviceException`
    - Effect: The given device is asked for recent measurements, this allows the AnomalyDetector to gather more recent information in case of an uncertain anomaly.
    - Exceptions:
      - NoSuchDeviceException: Thrown when the given device does not exist.
  - `void configure(DeviceID device, DeviceConfiguration configuration) throws InvalidConfiguratio NoSuchDeviceException`
    - Effect: A ConfigurationTrame is sent to the device.
    - Exceptions:
      - InvalidConfigurationException: Throw when the given configuration is invalid.
      - NoSuchDeviceException: Thrown when the given device does not exist.
- Send Trame
  - `void alert(AlarmTrame lowPriorityAlarm)`
    - Effect: The alarm is acknowledged and forwarded to the AnomalyDetector for validation.
    - Exceptions: None
  - `void confirm(TrameID trame)`
    - Effect: The trame corresponding to the given TrameID is confirmed.
    - Exceptions: None
  - `void lowBatteryAlert(PowerStatus status)`
    - Effect: *ReMeS* is notified of the low power status of the RMM.
    - Exceptions: None
  - `ConfigurationTrame initialConfigurationRequest()`
    - Effect: The initial configuration of the RMM is looked up and sent.
    - Exceptions: None
  - `void measurement(MeasurementTrame measurement)`
    - Effect: The measurement is processed, acknowledged and forwarded to the MeasurementDB-Manager.
    - Exceptions: None
  - `void priorityAlert(AlarmTrame highPriorityAlarm)`
    - Effect: The alarm is acknowledged and forwarded to the AlarmHandler to be handled and to the AnomalyDetector for validation.
    - Exceptions: None

A.26 ThirdPartyCommunication

Responsabilities The ThirdPartyCommunication

- Communicates with third parties;
- Authenticates third parties;
- Caches unacknowledged messages;
- Repeatadly attempts sending messages that are not acknowledged;
- Notifies an operator when third parties remain unavailable for an appropriate period of time;

Super component *ReMeS*

Sub components MessageCache, ThirdPartyFacade

Provided interfaces

- Authentication
- CustomerMgmt
- DeviceMgmt
- Dispatch Invoice

A.27 ThirdPartyFacade

Responsabilities The ThirdPartyCommunication

- Communicates with third parties;
- Authenticates third parties;
- Repeatadly attempts sending messages that are not acknowledged;
- Notifies an operator when third parties remain unavailable for an appropriate period of time;

Super component ThirdPartyCommunication

Sub components None

Provided interfaces

- Authentication
  - `Secret authenticate(Credential credential) throws InvalidCredentialException`
    - Effect: The credentials of the user are verified and if they are valid the user receives a secret to be used in all communication in the current session.
    - Exceptions:
      - InvalidCredentialException: Thrown when the credentials are invalid.
  - `void logOff(Secret secret)`
    - Effect: The secret is invalidated, effectively ending the session.
    - Exceptions: None
- ConsumptionPrediction
  - `ConsumptionPrediction requestConsumptionPrediction(Secret uisSecret, TimePeriod timePeriod) throws AutenticationFailure, NoPredictionPossibleException`

    - Effect: *ReMeS* sends a request to the ConsumptionPredictor to calculate the consumption prediction.
    - Exception:
        - AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        - NoPredictionPossibleException: Throws when the request cannot be fulfilled.

- CustomerInterface
  - `void configureDevice(Secret customerSecret, CustomerID customer, DeviceID device, DeviceConfi configuration) throws AuthenticationFailure, NoSuchCustomerException, NoSuchDeviceException`
    - Effects: Checks whether the configuration is reasonable and configures the device accordingly.
    - Exceptions:
        - AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        - NoSuchCustomerException: Thrown when the given customer does not exist.
        - NoSuchDeviceException: Thrown when the given device does not exist (all RMMs that are ready to be installed are registered in *ReMeS*).
  - `void configureProfile(Secret customerSecret, CustomerID customer, Profile changedProfile) throws AuthenticationFailure, InvalidProfileException, NoSuchCustomerException`
    - Effects: Checks whether the customer's changes to the profile are within reason and saves them. Then the customer's overview is updated accordingly.
    - Exceptions:
        - AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        - InvalidProfileException: Thrown when the changes to the profile are unacceptable.
        - NoSuchCustomerException: Thrown when the given customer does not exist.
  - `void confirmRegistration(RegistrationID registration) throws NoSuchRegistrationException`
    - Effect: The registration corresponding to the RegistrationID is confirmed.
    - Exceptions:
        - NoSuchRegistrationException: Thrown when no registration with a matching ID is in progress.
  - `void controlActuator(Secret customerSecret, CustomerID customer, DeviceID device, ActuatorSta actuatorState) throws AuthenticationFailure, NoSuchCustomerException, NoSuchDeviceException`
    - Effects: Changes the state of the actuator as long as it is not unsafe, e.g. no validated gas alarm has been received.
    - Exceptions:
        - AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        - NoSuchCustomerException: Thrown when the given customer does not exist.
        - NoSuchDeviceException: Thrown when the given device does not exist (all RMMs that are ready to be installed are registered in *ReMeS*).
  - `CustomerOverview generateOverview(Secret customerSecret, CustomerID customer) throws AuthenticationFailure, NoSuchCustomerException`
    - Effect: An overview of the customer's consumption, contact information for notifications, parameters for anomaly detection, etc., is returned.
    - Exceptions:
        - AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        - NoSuchCustomerException: Thrown when the given customer does not exist.
  - `Secret customerRegistration()`
    - Effect: A secret is returned with an appropriate clearance level, i.e. customers will not have the clearance to register without confirmation by a callcenter operator. This secret allows to register a new customer.
    - Exceptions: None
  - `RegistrationID registrationInformation(Secret registrationSecret, CustomerInformation customerInfo) throws AuthenticationFailure, DuplicateUsernameException, WeakPasswordException`

    * Effect: The customerInfo is added to the tentative registration.
    * Exceptions:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        · DuplicateUsernameException: Thrown when the given username already exists.
        · WeakPasswordException: Thrown when the entropy of the password is too small, e.g. the password is too short, the password is a common english word, the password does not contain enough different characters.
  - `void removeDevice(Secret customerSecret, DeviceID device) throws AuthenticationFailure, NoSuchDeviceException`
    * Effect: The given device is removed from the customer's account.
    * Exception:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        · NoSuchDeviceException: Thrown when the given RMM does not exist.
  - `void setRecipients(Secret customerSecret, DeviceID device, AlarmRecipients recipients) throws AuthenticationFailure, NoSuchDeviceException`
    * Effect: The alarm recipients are set for the given device.
    * Exceptions:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        · NoSuchDeviceException: Thrown when the given device does not exist.
  - `void unregisterCustomer(Secret registrationSecret, CustomerID customer) throws Authentication NoSuchCustomerException`
    * Effect: The customer corresponding to the ID is unregistered by *ReMeS*.
    * Exceptions:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        · NoSuchCustomerException: Thrown when the given customer does not exist.

- CustomerMgmt
  - `void approveRegistration(Secret callcenterSecret, RegistrationID registration) throws AuthenticationFailure, NoSuchRegistrationException`
    * Effect: The registration corresponding to the RegistrationID is approved.
    * Exceptions:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        · NoSuchRegistrationException: Thrown when no registration with a matching ID is in progress.
  - `Secret callcenterRegistration(Secret callcenterSecret) throws AuthenticationFailure`
    * Effect: A secret is returned with an appropriate clearance level. This secret allows to register a new customer.
    * Exceptions:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
  - `void confirmRegistration(Secret callcenterSecret, RegistrationID registration) throws AuthenticationFailure, NoSuchRegistrationException`
    * Effect: The registration corresponding to the RegistrationID is confirmed.
    * Exceptions:
        · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
        · NoSuchRegistrationException: Thrown when no registration with a matching ID is in progress.
  - `RegistrationID registrationInformation(Secret registrationSecret, CustomerInformation customerInfo) throws AuthenticationFailure, DuplicateUsernameException, WeakPasswordException`
    * Effect: The customerInfo is added to the tentative registration.
    * Exceptions:

· AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
· DuplicateUsernameException: Thrown when the given username already exists.
· WeakPasswordException: Thrown when the entropy of the password is too small, e.g. the password is too short, the password is a common english word, the password does not contain enough different characters.

– `void unregisterCustomer(Secret registrationSecret, CustomerID customer) throws Authentication NoSuchCustomerException`
  * Effect: The customer corresponding to the ID is unregistered by *ReMeS*.
  * Exceptions:
    · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
    · NoSuchCustomerException: Thrown when the given customer does not exist.

- DeviceMgmt
  – `void associateDevice(Secret technicianSecret, CustomerID customer, DeviceID device, DeviceID actuator) throws Authenticationfailure, NoSuchCustomerException, NoSuchDeviceExcepti`
    * Effect: The given device is associated with the given customer. If the optional argument for the actuator is given it is associated to the given RMM.
    * Exceptions:
      · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
      · NoSuchCustomerException: Thrown when the given customer does not exist.
      · NoSuchDeviceException: Thrown when the given device does not exist (all RMMs that are ready to be installed are registered in *ReMeS*).

- Dispatch Invoice
  – `void dispatchInvoice(Invoice invoice)`
    * Effect: The Invoice is forwarded to the Third Party Billing Service. If the Third Party Billing Service is unavailable, forwarding is retried in an appropriate fashion and an operator is notified if necessary.
    * Exceptions: None
  – `BillingInformation requestBillingInformation(UISID uisID, UISCustomerID customer, TimePeriod timePeriod) throws NoSuchUISException, NoSuchCustomerException`
    * Effect: The ThirdPartyCommunication contacts the UIS for the information needed by the Billing component.
    * Exceptions:
      · NoSuchUISException: Thrown when the given UIS does not exist.
      · NoSuchCustomerException: Thrown when the given customer does not exist.

- InvoiceConfirmation
  – `void markInvoicePaid(Secret secret, InvoiceID invoice) throws AuthenticationFailure, NoSuchInvoiceException`
    * Effect: The given invoice is marked as paid.
    * Exceptions:
      · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
      · NoSuchInvoiceException: Thrown when the given invoice does not exist.

- Research
  – `StatisticalReply statisticalQuery(Secret researchSecret, StatisticalQuery query) throws AuthenticationFailure, InvalidQueryException`
    * Effect: The given query is processed and anwsered.
    * Exceptions:
      · AuthenticationFailure: Thrown when the clearance of the given secret does not suffice.
      · InvalidQueryException: Thrown when the given query is unvalid.

B Defined data types

- `ActuatorState`: The state of an actuator, e.g. a valve can be open or closed, etc.
- `AlarmMessage`: A message to be sent to AlarmRecipients in case of an anomaly.
- `AlarmRecipient`: Represents an alarm recipient.
- `AlarmRecipients`: A collection of AlarmRecipients.
- `AlarmRecipientAppointmentID`: Unique identifier contained in the confirmation request to an appointed AlarmRecipient.
- `AlarmTrame`: Representation of an alarm from an RMM.
- `BillingInformation`: Represents information needed by the Billing component, e.g. frequency of billing, type of contract, etc.
- `ConfigurationTrame`: Represents the configuration trame of an RMM.
- `ConsumptionPrediction`: Contains a prediction of consumption for a certain TimePeriod.
- `Credential`: Used to identify users of *ReMeS*.
- `CustomerID`: Unique identifier of a customer.
- `CustomerInformation`: Contains information about a customer, e.g. SLA, associated devices, billing information, contact information, etc.
- `CustomerOverview`: Contains all the information offered to users, e.g. associated devices, consumption, alarm recipient configuration, etc.
- `DeviceConfiguration`: Represents an RMM configuration.
- `DeviceID`: Unique identifier of an RMM.
- `DeviceInformation`: Contains all pertinent information about a device, e.g. associated actuators, alarm recipients, type of communication channel, type of measured utility, etc.
- `EmergencyMessage`: A message to be relayed to the Emergency Services in case of emergency.
- `Invoice`: Represents an invoice in such a way that it can be communicated to the Third Party Billing Service.
- `InvoiceID`: Unique identifier of an invoice.
- `MeasurementTrame`: Representation of a measurement from an RMM.
- `MeasurementQuery`: Representation of a query to a database.
- `MeasurementReply`: Representation of a reply from a database.
- `Message`: Represents a message sent to third parties.
- `MessageID`: Identifies a message sent to third parties.
- `PowerStatus`: Represents the power status of a device.
- `Profile`: Contains a user's profile information, e.g. measurement frequency, username, credentials, etc.
- `RegistrationID`: Unique identifier of an ongoing registration, a new customer receives this after starting the registration procedure. This allows the user to identify himself to *ReMeS* to confirm the registration without requiring credentials that can only be validated after registration.
- `Secret`: A token provided to users through the login procedure that indicates their appropriate clearance level. This token expires a short while after the last activity.
- `SLA`: Representation of a Service Level Agreement.

- `StatisticalQuery`: A statistical query for research.
- `StatisticalReply`: Reply containing the results of a StatisticalQuery.
- `SubsystemID`: A unique identifier of a *ReMeS* subsystem, e.g. used to indicate availability problems.
- `TimePeriod`: Representation of a period of time.
- `ThirdPartyID`: Identifies third parties.
- `TrameID`: Unique identifier of a trame.
- `UISCustomerID`: An identifier of a customer that can be communicated to the UIS.
- `UISID`: Unique identifier of a utility company.