Patient Monitoring for Cardiovascular Diseases

ADD application

Abstract

This document describes the first three decompositions of the attribute-driven design approach (ADD) applied to the requirements of the Patient Monitoring System. This document applies ADD strictly and as such serves as an example of how to start architectural design from a set of functional and non-functional requirements. Since the three decompositions do not cover all requirements, the end-result of this document is only a *partial* architecture of the patient monitoring system (PMS).

Contents

1 Introduction

This document describes the first three decompositions of the attribute-driven design approach (ADD) applied to the requirements of the Patient Monitoring System. This document applies ADD strictly and as such serves as an example of how to start architectural design from a set of functional and non-functional requirements. Since the three decompositions do not cover all requirements, the end-result of this document is only a *partial* architecture of the patient monitoring system (PMS).

The remainder of this document is structured as follows. Section 2 starts by repeating the priorities of the requirements of the PMS as given by our stakeholders. Section 3 provides the log for the ADD approach. Section 4 documents the partial architecture resulting from the ADD decompositions.

2 Priorities of the requirements

For completeness and clarity, this section repeats the priorities for each requirement of the PMS as given by our stakeholders. Table 1 shows the quality requirements ranked high, medium or low. Table 2 shows the functional requirements ranked as must-have, nice-to-have and maybe.

Quality requirement	Av1	Av2	Av3	M1	M2	M3	P1	P2	P3
Priority	H	L	H	L	M	M	H	H	M

Legend: H = high, M = medium, L = low

Table 1: The priorities for the quality requirements as given by our stakeholders.

Category	Functionalities	Use cases
Must-have	Authentication	*UC1*, *UC2*
	Patient registration	*UC13*, *UC14*
	Clinical model configuration	*UC9*, *UC12*
	Patient monitoring	*UC4*, *UC5*, *UC8*
	Consultation	*UC11*
	Clinical model processing	*UC10*, *UC15*
	Cardiologist notification	*UC7*
Nice-to-have	Integration with HIS	*UC16*, *UC17*
	Telemedicine callcenter	*UC7*
	Emergency notifications	*UC3*
Maybe	Trustee/Buddy	*UC3*, *UC5*, *UC6*, *UC7*, *UC8*

Table 2: The priorities for the functional requirements as given by our stakeholders.

3 Attribute-driven design documentation

This section describes the execution of the attribute-driven design approach (ADD). Each of the following subsections describes a single decomposition and each decomposition is structured according to the ADD steps specified in [1, Chapter 7]: we (1) clearly state the module that is decomposed in the run, (2) list the selected architectural drivers and why we selected them, (3) elaborate on the architectural design for each non-functional driver, thereby discussing our approach and alternatives considered, (4) describe the resulting decomposition and allocation of functionality, (5) describe the resulting interfaces, (6) describe the data types used in these interfaces and (7) verify the allocation of all requirements and refine them.

3.1 Decomposition 1: PMS (P1, Av3, UC4, UC15)

3.1.1 Module to decompose

The first run decomposes the PMS as a whole.

3.1.2 Selected architectural drivers

The selected non-functional drivers are:

- *P1*: Storage of sensor data readings
- *Av3*: Internal PMS database failure

Two use cases in the PMS rely on the storage of sensor data and are thus selected as functional drivers. These are the following use cases:

- *UC4*: send sensor data
  This use case stores the received sensor data in the raw sensor data storage.
- *UC15*: compute clinical model
  This use case reads the monitoring history from the raw sensor data storage to compute the clinical model.

Rationale. *P1* and *Av3* were chosen because both have a high priority. They where selected among the other high-priority quality requirements because they both concern the storage of raw sensor data, which is essential to the other requirements of the PMS. Therefore, we decided to address these non-functional drivers in the first decomposition.

3.1.3 Architectural design

Scheduling for *P1*. *P1* specifies that the databases storing the sensor data must be able to cope with a potentially large amount of incoming data (i.e., updates) by **scheduling** the different updates. For this we introduce the `SensorDataScheduler`. The performance scenario focuses on scheduling for performance in two modi: normal modus and overload modus. In normal modus (i.e., when the required deadlines are met), the incoming sensor data updates are written to the database in a **dynamic priority scheduling: earliest deadline first** fashion using the deadlines as described in the quality scenario. When detecting that the deadlines are no longer met, the scheduler goes into overload modus and uses a **fixed priority scheduling** policy where sensor data updates are scheduled according to the patient risk level (i.e. **semantic importance**). More precisely, updates for patients with a red risk level have priority over updates for patients with either a yellow or green risk level. The updates for patients with a red risk level will be written in a first-in/first-out fashion. The `SensorDataScheduler` sorts all incoming requests in its queue and sorts it according to its current modus, normal or overload. By monitoring the throughput of this queue the `SensorDataScheduler` knows when to switch between modi. Note that the scheduling in overload modus does not enforce explicit deadlines for updates of yellow and green risk level patients. Thus it is possible that starvation occurs for updates of yellow and green risk level patients. Moreover, in overload modus, the `SensorDataScheduler` is allowed to omit sensor data of patients with a green risk level if required.

Dedicated sensor database for *Av3*. Besides the performance of the database storing sensor data it is also important that the database remains available in the presence of software and hardware failures. First, Av3 specifies that a failure of the database storing sensor data may not affect the availability of other types of persistent data. Therefore, the storage of sensor data is done by a dedicated database, or more precisely, the components handling the storage of the sensor data do not handle other data.

Back-up database for *Av3*. Next to storing the sensor data in a dedicated database, *Av3* also requires this database to seamlessly transition to a back-up database in case a database failure occurs. Therefore, we apply redundancy, or more precisely, **passive redundancy**. As a result, we introduce three components: the `PrimaryDB` which represents the primary database, the `StandbyDB` which represents the backup database (or *standby*) and the `ReplicationManager` which manages the replication. In our passive redundancy scheme, the `ReplicationManager` accepts all requests to the database and handles these using the `PrimaryDB`. As such, all data is written to the `PrimaryDB` and the `PrimaryDB` will be used for all reads. The `PrimaryDB` will periodically push its new data to the `StandbyDB` to synchronize its state. We choose to push data once every interval of 10 minutes. This relatively small interval allows to keep the `SensorDataCache`, introduced later, rather small while still allowing the `PrimaryDB` to schedule a synchronization push on a quiet moment (i.e. when few read and write requests arrive). The `ReplicationManager` is responsible for tracking the availability of the `PrimaryDB` and when it fails, it will start using the `StandbyDB` as primary database. To prevent loss of data due to a failure in between two synchronizations, we also introduce the `SensorDataCache`. The `ReplicationManager` uses the `SensorDataCache` to store all updates received for a period the length of at least two synchronization intervals, thus 20 minutes. When the `PrimaryDB` fails, the `ReplicationManager` will bring the `StandbyDB` up to date by first writing each cached update to the `StandbyDB`. Notice that the `SensorDataCache` can discard entries older than the synchronization period since they are sure to already be backed up to the `StandbyDB` database. Also notice that the cache can

contain sensor data already written to the `StandbyDB` in the last synchronization. However, since writing a sensor data update is an idempotent operation, writing this data to the `StandbyDB` when the `PrimaryDB` has failed does not introduce incorrect data in the `StandbyDB`. Also notice that currently, two replicas are used and that at all times one will take on the role of primary database and the other of standby database. This can be extended if required with multiple standby databases for increased robustness. Such an extension would only impact the manager in that it must select one standby to become the primary database in case of failure but has no impact on the rest of the system. And finally, also notice that the *Av3* requires only the backing up of sensor data related to patients with a red or yellow risk level. We deviate from this and backup all sensor data irrelevant of risk level. This does not entail large hardware or software costs, while otherwise a failed primary database of which the data cannot be recovered would result in a loss of all sensor data concerning patients with a green risk level.

Failure detection for *Av3*. Third, *Av3* also requires a failure of the primary database to be detected within five seconds. To tackle this, we make the `ReplicationManager` introduced above responsible for detecting any such failures. For this we use the read and write operations performed on the primary database as implicit **ping/echo** messages. This means that reading and writing data is used as a ping whereas the reply from the database, the requested data in case of reading and a confirmation in case of writing, is used as echo. To assure detection within five seconds the manager sends an explicit ping message to the primary database if no read or write operation was performed for a period of four seconds. The primary database must then reply within one second. When detecting a failure the `ReplicationManager` must switch to degraded modus. Furthermore, the `ReplicationManager` will send a notification to the sub-system responsible for forwarding notifications to the intended parties (i.e. in this case the PMS system administrator should be notified).

Omitting new sensor data. Both quality scenarios include a fall back modus, respectively called overload modus (*P1*) and degraded modus (*Av3*), in case the requirements are not achieved. Both these modi allow to omit at most two (consecutive) sensor data updates for patients with a green risk level. Overload modus is determined by the `SensorDataScheduler` whereas degraded modus is determined by the `ReplicationManager` for *Av3*. Should both the `SensorDataScheduler` and the `ReplicationManager` separately be able to drop two updates, up to four sensor data updates could be omitted (i.e. two by the `SensorDataScheduler` in overload modus and two by the `ReplicationManager` in degraded modus), thereby violating both quality scenarios. To avoid this, the omission of sensor data updates must be determined in a single location. Therefore, we decided that the `ReplicationManager` informs the `SensorDataScheduler` when degraded modus is required (i.e. when the primary database fails). Thus it is always the responsibility of the `SensorDataScheduler` to determine whether a sensor data update can be omitted or not.

Reading vs writing sensor data. Computing clinical models (*UC15*) requires to retrieve a patient's monitoring history (i.e. read access to the sensor data storage). As a consequence the `SensorDataScheduler` will contain both read and write requests in its queue. Handling the read requests can interfere with achieving the deadlines for storing sensor data as required by *P1*, especially if the number of read requests grows very large (i.e. when the number of monitored patients grows large). To minimize this interference, we decided to define a deadline of five minutes for read requests, i.e. longer than all write requests. In normal modus, write requests have higher priority while there is no starvation of read requests due to the deadline. In overload modus, starvation of read requests can occur just as is the case for write requests for sensor data concerning patients with a green of yellow risk level.

Handling race conditions. By introducing the `SensorDataScheduler`, read and write requests for sensor data are handled asynchronously. However, it can be the case that new sensor data is stored and shortly after requested again. For example, for now, we presume that the `GatewayFacade` (which first receives the new sensor data) stores this data in the sensor database and launches a new risk estimation. The new sensor data is then again fetched from the sensor database during this risk estimation.

An important remark is that this flow suffers from a possible race condition where the risk estimation sub-system requests the new sensor data from the database before it is actually stored (i.e. the write operation is scheduled, but not yet executed). To remedy this race condition, the `GatewayFacade` includes the sensor data update in the request to schedule a risk estimation and passes the time at which it received the update to both the sensor data database and the risk estimation sub-system. As such, the latter can use the time-stamp when requesting other sensor data from the database in order to avoid receiving the new sensor data again.

Alternatives considered

Alternatives for passive redundancy. An alternative to the use of passive redundancy would have been the use of **active redundancy**. With active redundancy, all replicas respond to events in parallel and

as a result, all replicas are in sync. Active redundancy would keep the sensor database available without resynchronization. However, active replication has a negative impact on the write performance since the data must be consistently and atomically written to each instance of the database. Passive redundancy does not introduce this extra latency since the data is initially written to a single (primary) database and synchronized to the standby database at a later point. Considering that the quality scenario (*P1*) concerns only the writing of sensor data we did not select active redundancy.

Alternatives for the race condition. In general, there are three alternatives to remedy the race condition mentioned above. (1) The first is to include the sensor data update in the request to schedule a risk estimation. In this manner the risk estimation sub-system is sure to have the update when it executes the risk estimation. Care must be taken when the monitoring history is retrieved, since this can already contain the sensor data update resulting in duplicate data (i.e. the sensor data update sent with the risk estimation request and the sensor data update already stored in the database). This can be avoided by sending a time-stamp along with the risk estimation request. The risk estimation sub-system retrieves the monitoring history up until the time-stamp, thus without the new sensor data. (2) The second alternative remedies the race condition in a different manner. Here the `SensorDataScheduler` sends the risk estimation request to the designated sub-system. By sending this request only after the sensor data update has been stored the race condition is avoided. This alternative results in a **pipe-and-filter** system where each component performs its task and notifies the next component when he is finished. (3) Finally, the third alternative is to remedy the race condition by having the `SensorDataScheduler` know about this work flow and have it guarantee that the read request will never be scheduled before the write request While this alternative encapsulates the complexity of the race condition, its disadvantages are that it limits the possibilities of the scheduler and makes it more specific to our application. As mentioned above, we selected the first as best option for our architecture: It does not suffer from race conditions while still providing the flexibility to easily change the work flow when needed.

3.1.4 Instantiation and allocation of functionality

This section describes the components which instantiate the selected tactics and patterns, how the related functionality is allocated and how the components are deployed on physical nodes. In addition, we illustrate the behavior of these components using sequence diagrams in order to clarify how the sensor data is stored and processed.

Decomposition. Figure 1 shows the components resulting from the decomposition in this run. Extra attention is required concerning the `PrimaryDB` and `StandbyDB`. These components are actually two instances of the same storage component. Therefore, they provide and require the same interfaces. But depending on their run-time role, i.e. primary or standby database (cf. the decisions concerning *Av3* above), a specific set of these is activated. This is indicated by having both component instances provide and require the same interfaces, but differing the connections between components dependent on the role of the instance.

Furthermore, we instantiated `GatewayFacade`, `PatientStatusDB` and `NotificationHandler` to fully achieve the selected drivers. The `PatientStatusDB` stores the risk level of each patient. We assigned this responsibility to a separate database to be able to deploy it optimally in a fine-grained matter later on.

The responsibilities of the resulting components are as follows:

GatewayFacade Responsible for communication with the patient gateways. This consists mainly of accepting incoming sensor data updates and forwarding them to the `SensorDataScheduler`. Subsequently this component translates the given gateway id to the id of the associated patient and sends the sensor data update to the sub-system responsible for scheduling and processing this update (*UC4*).

NotificationHandler Responsible for communication with the PMS system administrators in case of failure of the sensor data database (*Av3*).

OtherFunctionality1 Encapsulates requirements not tackled in this run and which cannot be assigned to other introduced components.

PatientStatusDB Responsible for storing the status (i.e. the risk level) of patients.

PrimaryDB Responsible for the actual storage of the sensor data. This component responds to both read and write requests concerning sensor data (*Av3*).

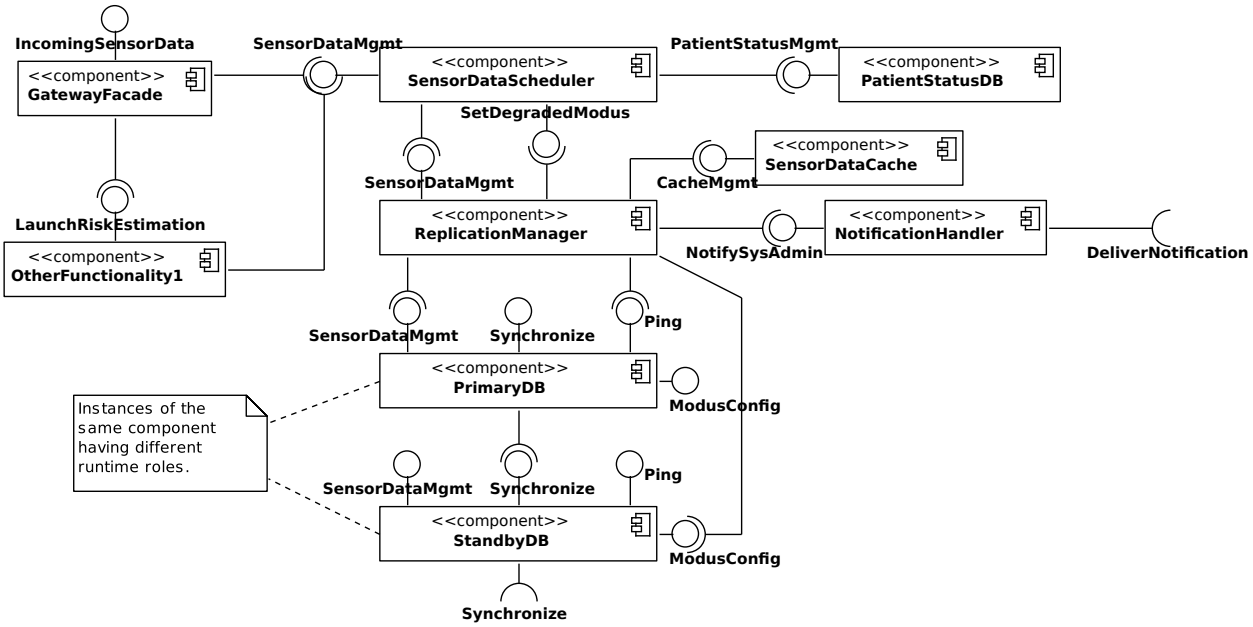

Figure 1: The main component-and-connector diagram resulting from the first decomposition. The dangling interfaces of the `PrimaryDB` and `StandbyDB` are used to illustrate that the `PrimaryDB` and `StandbyDB` are identical in terms of interfaces, but that at each point in time only one is used as primary database and the other as standby. The other dangling interfaces are provided or used by components external to the PMS.

StandbyDB Responsible for backing up the sensor data stored on the primary database. This means periodically synchronizing its content with the primary database (*Av3*).

SensorDataCache Responsible for storing the sensor data written in between synchronizations of the primary and standby databases (*Av3*).

SensorDataScheduler Responsible for scheduling all operations to be performed on the sensor data database in its queue and monitoring whether the deadlines are met. If not the scheduler switches from normal modus to overload modus and changes its scheduling policy accordingly (*P1*).

ReplicationManager Responsible for handling the requests pushed by the `SensorDataScheduler` and managing the replicas of the sensor data database (i.e. the `PrimaryDB` and `StandbyDB`), including checking their availability and if necessary switching to a different primary database (*Av3*). In case of a failure, a request to notify a system administrator is forwarded to the `NotificationHandler` (*Av3*).

Deployment. Following the decomposition of the system into new components, Figure 2 shows how the components are deployed over several physical nodes. Note that the `PrimaryDB` and `StandbyDB` databases are deployed on different nodes. Similarly to the roles of these components, the roles of the nodes on which they are deployed can change at run-time. Furthermore, most components are deployed on an `OtherFunctionalityNode`, which indicates that their deployment is not essential for achieving the architectural drivers of this decomposition. The deployment of these components will be decided in a future decomposition.

Behavior. Next to the decomposition and deployment, the following paragraphs clarify the behavior of the architecture resulting from this first run. Figures 3 and 4 respectively show how sensor data is written and read when the corresponding operation reaches the head of the `SensorDataScheduler`'s queue. As shown, new sensor data are first stored in the `SensorDataCache` before they are written to the data in the `PrimaryDB` database and read requests are only executed on the data in the `PrimaryDB` database after which the `SensorDataScheduler` sends the data to the original requester. Finally, Figure 5 shows what happens when a `PrimaryDB` database failure is detected by the `ReplicationManager`: The `StandbyDB` will first be brought to a sufficiently fresh state by writing the sensor data updates arrived since its last synchronization from the cache. The `ReplicationManager` then assigns the role of primary database to the `StandbyDB`. When restarted, the former `PrimaryDB` database is assigned the role of standby database and it synchronizes its state with the current primary database.

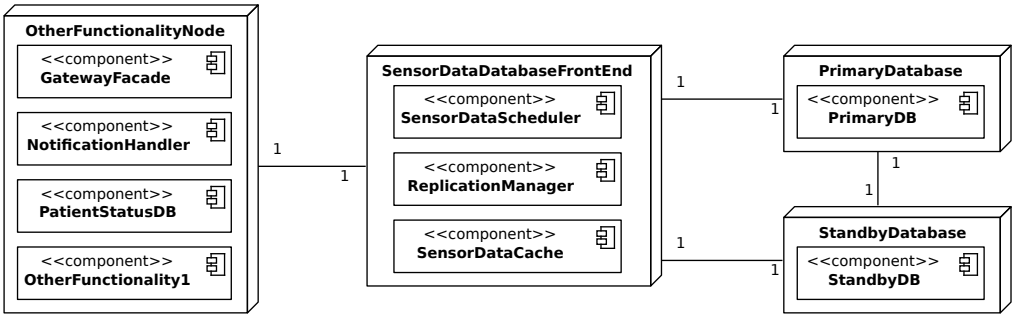

Figure 2: The main deployment diagram resulting from the first decomposition.

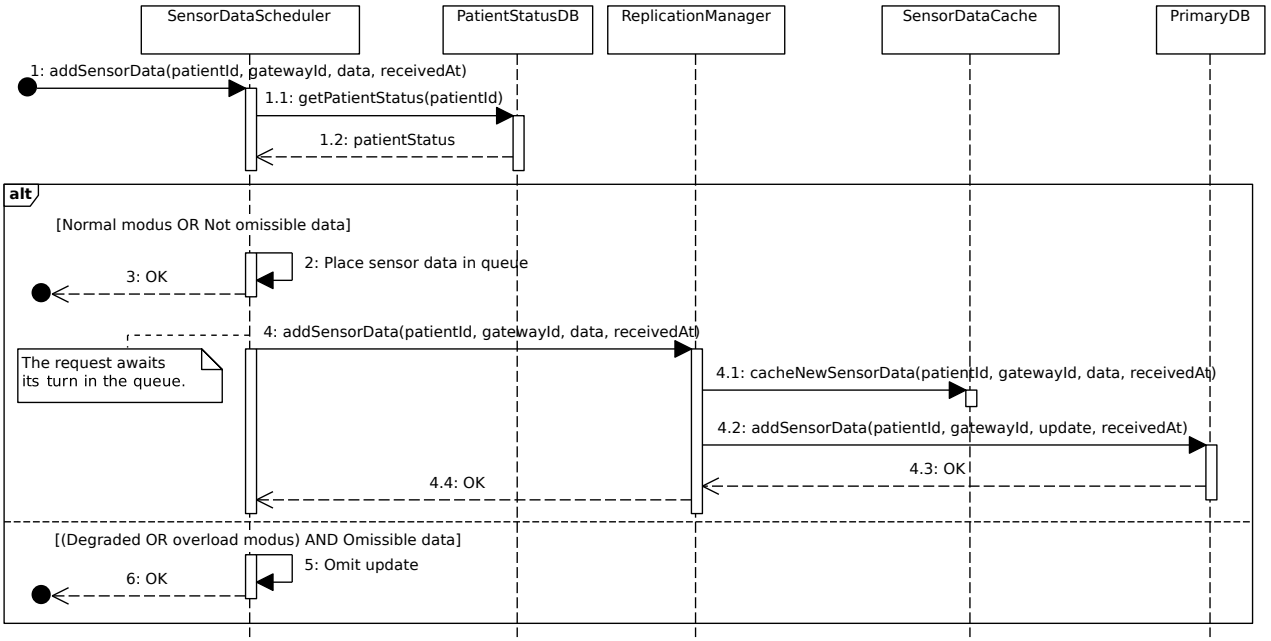

Figure 3: Illustration of how the sensor data database handles writes.

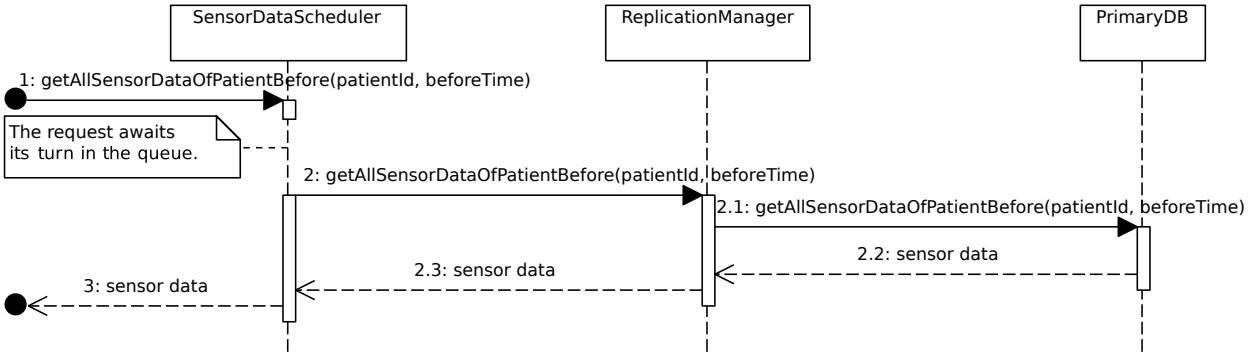

Figure 4: Illustration of how the sensor data database handles reads.

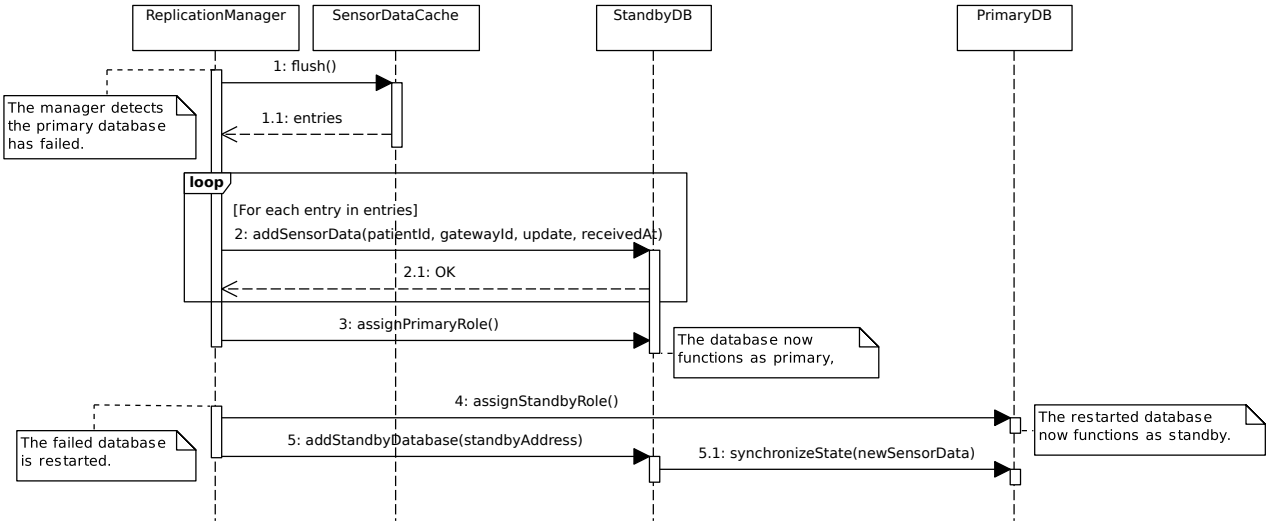

Figure 5: Illustration of the behavior when the `PrimaryDB` fails.

3.1.5 Interfaces for child modules

This section describes the interfaces assigned to the components detailed in the above section. Per interface, we list its methods by means of its syntax. Per method, we also list its effect (mostly in terms of subsequent actions) and possible exceptions. The data types used in these interfaces are defined in the following section.

GatewayFacade

- IncomingSensorData
  - `void sendSensorData(GatewayId id, SensorDataPackage sensorData)`
    - Effect: Sends a sensor data update `sensorData` to the system for the patient gateway identified by `id`.
    - Exceptions: None

NotificationHandler

- NotifySysAdmin
  - `void notifySysAdmin(NotificationMessage msg)`
    - Effect: Sends the given message to a PMS system administrator.
    - Exceptions: None

OtherFunctionality1

- LaunchRiskEstimation
  - `void launchRiskEstimation(PatientId id, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: Schedules a risk estimation for the patient identified by `id` for the newly arrived sensor data `sensorData`. The `receivedAt` time-stamp avoids that the risk estimation retrieves a duplicate of the `sensorData` when retrieving the monitoring history.
    - Exceptions: None

PatientStatusDB

- PatientStatusMgmt
  - `PatientStatus getPatientStatus(PatientId patientId) throws NoSuchPatientException`
    - Effect: Returns the status of the patient corresponding to `patientId`.
    - Exceptions:
      - NoSuchPatientException: Thrown if no patient with the given id exists.

PrimaryDB

- ModusConfig
  - `void addStandbyDatabase(StandbyAddress standbyAddr)`
    - Effect: Adds the database located at `standbyAddr` as standby database to which new data must be synchronized.
    - Exception: None
  - `void assignPrimaryRole()`
    - Effect: Assigns the callee the role of primary database.
    - Exceptions: None
  - `void assignStandbyRole()`
    - Effect: Assigns the callee the role of standby database.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: Returns an echo to the caller indicating that the callee is available.
    - Exceptions: None
- SensorDataMgmt
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: Stores sensor data package `sensorData` sent by the gateway identified by `gatewayId` belonging to the patient identified by `patientId`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId id, TimeStamp untilTime)`
    - Effect: Returns the monitoring history up until `untilTime` belonging to the patient identified by `id`. If no such patient exists or if no data exists for that is, an empty map is returned, but no error is thrown.
    - Exceptions: None
- Synchronize
  - `void synchronizeState(List<Tuple<PatientId, GatewayId, SensorDataPackage, TimeStamp>> sensorData)`
    - Effect: Writes all sensor data and meta-data in `sensorData` to the database.
    - Exceptions: None

ReplicationManager

- SensorDataMgmt
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: Stores the sensor data package `sensorData` sent by the gateway identified by `gatewayId` belonging to the patient identified by `patientId` in the `PrimaryDB`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId id, TimeStamp untilTime)`
    - Effect: Fetches the monitoring history up until `untilTime` belonging to the patient identified by `id` from the `PrimaryDB` and returns it. If no such patient exists or if no data exists for that is, an empty map is returned, but no error is thrown.
    - Exceptions: None

SensorDataCache

- CacheMgmt
  - `void cacheNewSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: Adds the sensor data update `sensorData` along with its meta-data to the cache.
    - Exceptions: None

- `List<Tuple<PatientId, GatewayId, SensorDataPackage, TimeStamp>> flush()`
  - Effect: Returns a list containing all entries in the `SensorDataCache`, sorted by age with the eldest entry first and removes these entries from the cache.
  - Exceptions: None

SensorDataScheduler

- SensorDataMgmt
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: Schedules the given sensor data package `sensorData` sent by the gateway identified by `gatewayId` belonging to the patient identified by `patientId` to be stored in the `PrimaryDB`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId id, TimeStamp untilTime)`
    - Effect: Schedules the read request for the monitoring history up until `untilTime` belonging to the patient identified by `id` to be executed by the `PrimaryDB` and returns the result afterwards. If no patient with given id exists or if no data exists for that id, an empty map is returned, but no error is thrown.
    - Exceptions: None
- SetDegradedModus
  - `void setDegradedModus()`
    - Effect: Switches the scheduler to degraded modus.
    - Exceptions: None
  - `void setNondegradedModus()`
    - Effect: Switches off degraded modus. The scheduler will determine whether it must go into normal or overload modus.
    - Exceptions: None

StandbyDB

- ModusConfig: See `PrimaryDB::`ModusConfig
- Ping: See `PrimaryDB::`Ping
- SensorDataMgmt: See `PrimaryDB::`SensorDataMgmt
- Synchronize: See `PrimaryDB::`Synchronize

SystemAdministratorDevice

- DeliverNotification
  - `void deliverNotifification(NotificationMessage msg)`
    - Effect: Send the given notification message to a PMS system administrator.
    - Exceptions: None

3.1.6 Data type definitions

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

Echo A message used as response to a ping request.

GatewayId A data element uniquely identifying a patient gateway.

NotificationMessage A data element containing a message, sender and receiver.

PatientId A data element uniquely identifying a patient monitored by the PMS.

PatientStatus A data element containing the status of a patient. Currently this only contains the risk level of a patient and a PatientId identifying the corresponding patient.

SensorDataPackage A data element containing a value for each sensor in the wearable unit.

StandbyAddress Data element representing the network address through which a standby database can be reached.

TimeStamp A data element representing a time-stamp.

3.1.7 Verify and refine

The requirements not tackled in this decomposition must be mapped to the components that resulted from decomposing the PMS. The following shows per component which remaining requirements are assigned to it. If requirements are split, a letter is added to its name (e.g. *Av1**a***) to indicate that the component will only satisfy a part of the original requirement. If requirements are split, it will also be indicated what part of the original requirement each new requirement contains. If new derived requirements are introduced they will receive a new identifier and are marked as a new requirement, e.g. *UC18* for `GatewayFacade`.

GatewayFacade

- *UC3a*: send emergency notification
  Receive emergency notification and forward for processing.
- *UC18*: Translate gateway identifiers to patient identifiers (*new use case*)
  This functionality is derived from the need to associate gateways to patients when receiving sensor data.
- *Av1a*: Communication channel between the patient gateway and the PMS
  Detect failure of patient gateway, the telecom infrastructure and internal communication components.
- *M2b*: New type of sensor in the wearable unit
  PMS must be able to accept the new type of sensor data.

NotificationHandler

- *UC7*: notify
- *Av1b*: Communication channel between the patient gateway and the PMS
  Send notifications concerning failures to the PMS system administrators.
- *Av2b*: Availability of the patient record database in the hospital Information System (HIS)
  Send notifications concerning failures to the PMS system administrators.

OtherFunctionality1

- *UC1*: log in
- *UC2*: log off
- *UC3b*: send emergency notification
  Process the emergency notification and forward for storage in the patient record. If necessary contact the emergency services.
- *UC5*: fill out questionnaire
- *UC6*: appoint trustee
- *UC8*: consult patient status
- *UC9*: configure a clinical model
- *UC10*: update risk level
- *UC11*: perform on-demand consultation
- *UC12*: select clinical models for patient
- *UC13*: register patient
- *UC14*: unregister patient
- *UC15a:* compute clinical model
  Retrieve all required information, compute clinical model and if needed change risk level and inform interested parties. Note that the rest of UC15 has been solved by other components.
- *UC16*: consult patient record
- *UC17*: update patient record
- *Av2a*: Availability of the patient record database in the hospital Information System (HIS)
  Detect failure of the HIS and disruptions in the communication channel and store older copy of patient records.

- *P2*: Risk estimation by clinical models
- *P3*: Emergency notifications
- *M1*: Integration of the PMS into a different Hospital Information System
- *M2a*: New type of sensor in the wearable unit
  Client software must be updated and clinical models must be added/updated to cope with the new type of information.
- *M3*: New clinical model/optimized clinical model

PrimaryDB

- None

ReplicationManager

- None

SensorDataCache

- None

SensorDataScheduler

- None

StandbyDB

- None

3.2 Decomposition 2: OtherFunctionality1 (P2, UC15a, UC9, UC12)

3.2.1 Module to decompose

In this run we further decompose the `OtherFunctionality1` component. Note that the `OtherFunctionality1` component is treated as any other component concerning its decomposition.

3.2.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *P2*: Risk estimation by clinical models

Three use cases are related to the risk estimation in that they either trigger a risk estimation or perform the actual computations. The following use cases are thus the functional drivers of this decomposition.

- *UC9*: configure a clinical model
  Triggers a risk estimation.
- *UC12*: select clinical models for patient
  Triggers a risk estimation.
- *UC15a*: compute clinical model
  Performs the actual risk estimation.

Rationale. *P2* was chosen because it concerns the performance of the risk estimation for the patients monitored by the PMS. This requirement has high priority since risk estimation is a central aspect of the system. Therefore, we tackle it early on in the architectural design. With respect to *Av1* (which is also of high priority), we reason that *Av1* is already further refined and decomposed and that therefore, *P2* can have a larger architectural impact. Therefore, *P2* should be handled before *Av1*.

Risk level	Deadline (minutes)
red	2
yellow	5
green	5

Table 3: Deadlines for the initiation of risk estimations per risk level in overload modus.

3.2.3 Architectural design

Scheduling for *P2*. Since estimating the risk levels of patients is crucial to the PMS, the responsible sub-system should have high performance and be easily scalable. *P2* focuses on **scheduling** the incoming requests for risk estimations and clearly states the requirement for two modi: a normal modus and overload modus. Therefore, we introduce two components: the `RiskEstimationScheduler` and the `RiskEstimationProcessor`. The `RiskEstimationScheduler` is responsible for scheduling the different risk estimations and enforcing the two scheduling modi. More precisely, it queues all incoming requests for risk estimations and sorts this queue according to the current modus. The `RiskEstimationProcessor` is responsible for performing the actual risk estimations and asynchronously pops a risk estimation job from the `RiskEstimationScheduler`, one after the other.

The `RiskEstimationScheduler` determines when to switch between the two scheduling modi by monitoring the throughput of its queue. In normal modus the scheduler will schedule risk estimation requests in a **first-in/first-out** fashion. In overload modus (i.e. when the throughput exceeds 20 risk estimations per minute) a **dynamic priority scheduling** policy based on **earliest deadline first** is used. For this, we assign deadlines to the risk estimations based on the level of the corresponding patients according to *P2*: risk estimations for patients with red risk level have priority over risk estimations for patients with yellow and green risk levels. The resulting deadlines are shown in Table 3. Note that scheduling based on deadlines avoids starvation of lower risk level estimations.

Load balancing for *P2*. Next to scheduling, *P2* requires the PMS to balance the load over multiple instances of the risk estimation sub-system. Therefore, we **replicate** the `RiskEstimationProcessor`. This does not require any specific functionality in the `RiskEstimationScheduler` since the `RiskEstimationProcessors` pop risk estimation jobs from the scheduler themselves (as opposed to having the `RiskEstimationScheduler` actively initiate new risk estimations on the `RiskEstimationProcessors`). This also allows to easily add or remove replicas.

Replicating the `RiskEstimationProcessor` can be achieved on two levels. First, multiple instances of the `RiskEstimationProcessor` can be instantiated on **multiple physical nodes**. As a result, multiple risk estimations can be performed concurrently. Note that this concurrency does not introduce race conditions since the risk models do not read data written by other risk estimations.

Second, we also introduce **concurrency on a single node** by having each risk estimation node also host multiple instances of the `RiskEstimationProcessor`. The advantage of introducing concurrency on a single node is that it allows another risk estimation to be executed once the first is blocked for I/O. Since a risk estimation could require considerable amounts of data (e.g. monitoring history, current patient status and patient record) it can be expected that the risk estimations will be waiting for retrieving this data for considerable amounts of time. As a result, it can also be expected that the throughput improvement of single-node concurrency will be substantial.

As a remark, notice that one must be careful not to initiate too many risk estimations on a single node. This might cause the risk estimation queue to be emptied, while the large number of risk estimations are then contending for processing time on the heavy occupied risk estimation nodes, increasing the duration of each single risk estimation and decreasing the overall throughput. Furthermore, each initiated risk estimation requires local storage on its node to store the data it requires throughout computation.

As another remark, notice that the introduced replication allows the PMS to support the minimum throughput of 20 risk estimations per minute by scaling out the number of processing nodes. The number of nodes can be estimated using statistical methods on the expected arrival rate of sensor data for the expected number of monitored patients. However, *P2* also imposes a hard deadline of 10 minutes for the initialization of each job. While the statistical estimation of the number of processing nodes allows to minimize the chances of failing to keep this deadline, a fixed number of nodes can strictly speaking not handle any peak load. To achieve this, the architecture could be extended with dynamic scaling out the number of processor nodes for peak loads. For now, we decided to avoid this complexity in this architecture, but this can be reconsidered in the future when needed.

Multiple risk estimations for a single patient. It is important to note that a single risk estimation for a single patient can encompass multiple clinical models. To maximize the parallelization, we

consider each computation of a clinical model as a separate job. Thus, the `RiskEstimationScheduler` will not schedule a risk estimation for a patient as a whole, but will divide the risk estimation in smaller jobs, one for each clinical model assigned to the patient. The `RiskEstimationScheduler` will group the jobs belonging to the same risk estimation for the same patient in the queue and will not interleave these groups by jobs belonging to other risk estimations.

It is important to notice that each of the jobs only provides a partial result concerning the risk estimation as a whole. Therefore, the different partial results, possibly computed by different instances of the sub-system, have to be combined in order to acquire the final result of a risk estimation. Therefore, we introduce the `RiskEstimationCombiner` which is responsible for combining the partial results. We decided not to replicate the `RiskEstimationCombiner` to avoid the resulting management complexity and because this combination should be achievable with little resources.

Databases for the required data. Similar to the first decomposition, we instantiate separate database components for the different types of data required by this decomposition. As such, we instantiate the `ClinicalModelDB`, which is responsible for storing the clinical models and their configuration for each patient. The `PatientStatusDB` was already introduced in the previous decomposition. And for now, we do not instantiate a database for the questionnaires, reasoning that we have not tackled any core requirements related to them, but define the `OtherFunctionality2` component to provide the necessary interfaces to the `RiskEstimationProcessor`.

Alternatives considered

Alternatives for distributing risk estimation jobs As mentioned above, we chose to have the `RiskEstimationProcessor`s actively pop risk estimation jobs from the `RiskEstimationScheduler`. The alternative would be to have the `RiskEstimationScheduler` push jobs to the multiple replicas. The disadvantage of this scheme is that it requires the `RiskEstimationScheduler` to be reconfigured every time the number of replicas changes. Moreover, having the `RiskEstimationProcessor`s ask for new risk estimations themselves avoids notifying the `RiskEstimationScheduler` once they have completed a risk estimation.

Alternatives for scheduling the sets of jobs. As mentioned above, the `RiskEstimationScheduler` will group the jobs belonging to the same risk estimation in the queue and will not interleave these groups by jobs belonging to other risk estimations. The advantage of this approach is that the multiple jobs for a single risk estimation are scheduled closely after each other, thereby avoiding additional delay for the concerned patient. However, this approach does have the disadvantage that the same input data (e.g. sensor data) may be requested multiple times for the same risk estimation (e.g. when multiple jobs require the monitoring history). This could be mitigated by retrieving and caching all required data before the first job of a risk estimation starts. However, this also introduces possible race conditions if the data changes between retrieval and execution of the jobs, which cloud result in executing jobs based on outdated data and lead to incorrect or irrelevant results. These problems could be mitigated using cache invalidation which in turns adds considerable complexity to the PMS since each sub-system providing data for risk estimations must be able to invalidate the cache. Furthermore there is the problem of handling already started jobs when their data is reported to be outdated while they are executing. We decided to avoid this extra complexity and retrieve the data required for each job when the execution of the job starts. If this becomes a performance issue in the future, these solutions can be reconsidered.

3.2.4 Instantiation and allocation of functionality

This section describes the components which instantiate the selected tactics and patterns, how the related functionality is allocated and how the components are deployed on physical nodes. Unless stated otherwise the responsibilities assigned in the first decomposition are unchanged.

Decomposition. Figure 6 shows the component-and-connector diagram resulting from this decomposition. For clarity, we only show the decomposition of `OtherFunctionality1` with its directly connected components. Notice that with respect to Figure 1, `OtherFunctionality1` in Figure 6 employs three additional interfaces: NotifySubscribers, PatientRecordMgmt and PatientStatusMgmt, which were not foreseen in the first decomposition.

The responsibilities of the resulting components are as follows:

ClinicalModelDB Responsible for storing the clinical models available in the PMS as well as the patient-specific configurations of these models.

GatewayFacade See `GatewayFacade` in decomposition 1.

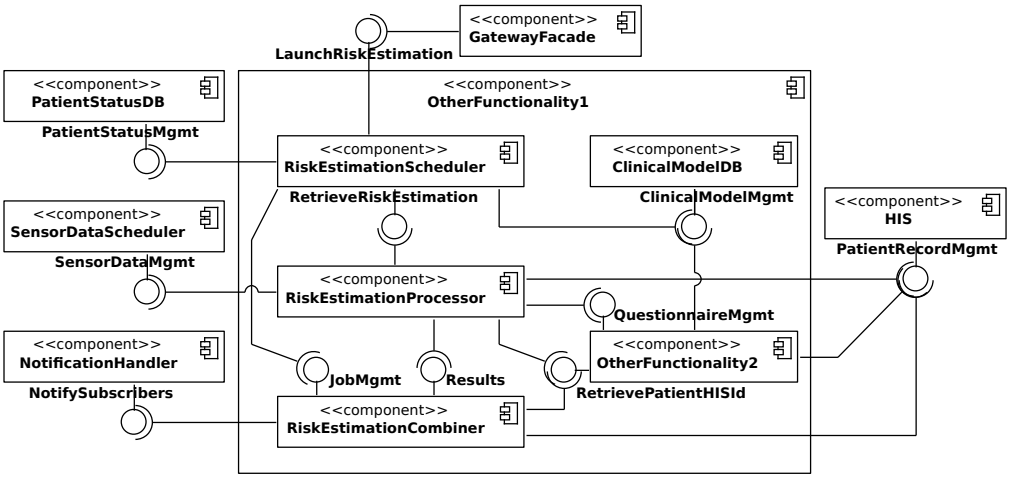

Figure 6: The decomposition of the `OtherFunctionality1` component resulting from the second decomposition.

NotificationHandler See `NotificationHandler` in decomposition 1. Furthermore, this component has the responsibility to notify interested parties of the results of a risk estimation (*UC15*).

OtherFunctionality2 Responsible for the remaining functional and non-functional requirements.

PatientStatusDB See `PatientStatusDB` in decomposition 1.

PrimaryDB See `PrimaryDB` in decomposition 1.

ReplicationManager See `ReplicationManager` in decomposition 1.

RiskEstimationCombiner Responsible for combining the results of the different clinical model computation jobs belonging to the same risk estimation. Furthermore, this component determines whether a notification must be sent and/or the patient record must be updated.

RiskEstimationProcessor Responsible for computing the clinical models to estimate the risk level op patients.

RiskEstimationScheduler Responsible for scheduling risk estimations and monitoring the throughput of the risk estimations.

SensorDataScheduler See `SensorDataScheduler` in decomposition 1.

StandbyDB See `StandbyDB` in decomposition 1.

Deployment. Following the decomposition of the system into new components, Figure 7 shows how the components are deployed over several nodes. As in the first decomposition only components whose deployment influence the achieving of the architectural drivers are explicitly deployed in a way satisfying the drivers. The deployment of the components in the `OtherFunctionalityNode` will be decided in later decompositions. The most important aspect of the deployment is that there are multiple `RiskEstimator` nodes and that each node hosts multiple `RiskEstimationProcessor`s. Note that the `ClinicalModelDB` is deployed on the `RiskEstimationFrontEnd`. The data contained in this databases is important for the risk estimation thus deploying it in this subsystem minimizes any latencies.

3.2.5 Interfaces for child modules

This section describes the interfaces assigned to the components detailed in the above section. For each interface is detailed who provides it and what resources the interface provides. The data types used in these interfaces are defined in the following section.

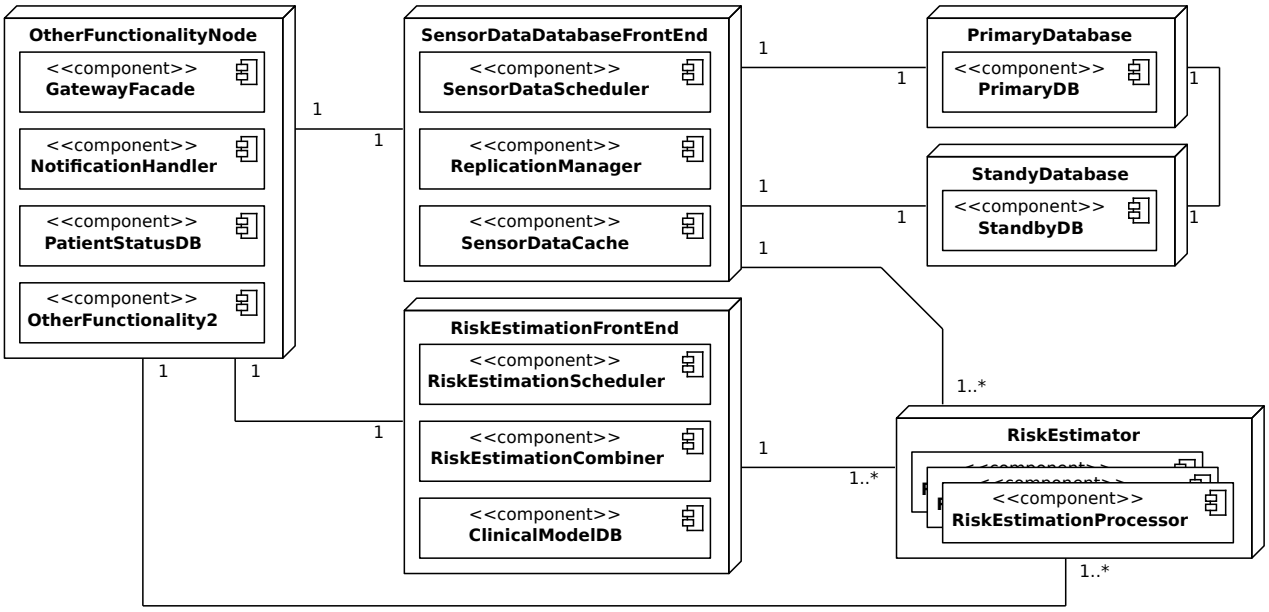

Figure 7: The main deployment diagram resulting from the second decomposition.

ClinicalModelDB

- ClinicaModelMgmt
  - `List<ClinicalModel> getAvailableClinicalModels()`
    - Effect: Returns all clinical models available in the PMS.
    - Exceptions: None
  - `List<ClinicalModel> getAssignedClinicalModels(PatientId id)`
    - Effect: Returns all clinical models assigned to the patient identified by `id`.
    - Exceptions: None
  - `Map<ClinicalModelId, Map<ClinicalModelParameterKey, ClinicalModelParametervalue>> getAssignedClinicalModelsAndConfigForPatient(PatientId id)`
    - Effect: Returns all clinical models and their configuration for the patient identified by `id`.
    - Exceptions: None
  - `void assignClinicalModel(PatientId patientId, ClinicalModelId clinicalModelId)`
    - Effect: Assigns the clinical model identified by `clinicalModelId` to the patient identified by `patientId`. The assigned clinical model is initiated using its default configuration.
    - Exceptions: None
  - `void updateClinicalModelConfiguration(PatientId patientId, ClinicalModelId modelId, ClinicalModelParameterKey key, ClinicalModelParametervalue value)`
    - Effect: Updates the configuration of the clinical model with identifier `modelId` assigned to the patient identifier by `patientId` with the given key-value pair.
    - Exceptions: None

GatewayFacade

See `GatewayFacade` in decomposition 1.

HIS

- PatientRecordMgmt
  - `PatientRecord getPatientRecord(HISPatientId id)`
    - Effect: Returns the patient record of the patient identified by `id`.
    - Exceptions: None

NotificationHandler

- NotifySubscribers
  - `void notifySubscribersOfPatient(PatientId id, NotificationMessage msg)`
    - Effect: Notifies the parties subscribed for updates concerning the patient identified by `id`.
    - Exceptions: None
- NotifySysAdmin: See `NotificationHandler::`NotifySysAdmin in decomposition 1.

OtherFunctionality2

- QuestionnaireMgmt
  - `List<Questionnaire> getQuestionnairesForPatient(PatientId id)`
    - Effect: Returns a list of all questionnaires filled out for the patient identified by `id`.
    - Exceptions: None
- RetrievePatientHISId
  - `HISPatientId getHISPatientIdForPatienId(PatientId id) throws NoSuchPatientException`
    - Effect: Returns the identifier used by the HIS of the patient identified by the given internal `id`.
    - Exceptions:
      - NoSuchPatientException: Thrown if no patient with the given `id` exists.

PatientStatusDB

See `PatientStatusDB` in decomposition 1.

PrimaryDB

See `PrimaryDB` in decomposition 1.

ReplicationManager

See `ReplicationManager` in decomposition 1.

RiskEstimationCombiner

- JobMgmt
  - `void addJobSet(RiskEstimationiId id, List<ClinicalModelJobId> jobIds)`
    - Effect: Adds a set of identifiers for jobs belonging to a single risk estimation identified by `id`. The partial results of each clinical model computation job identified by an element in `jobIds` have to be combined in order to find the final result of the risk estimation as a whole.
    - Exceptions: None
- Results
  - `void addClinicalModelJobResult(ClinicalModelJobId jobId, ClinicalModelJobResult result)`
    - Effect: Sends the `result` of the performed clinical model computation identified `jobId` to the `RiskEstimationCombiner` for combination with the other partial results belonging to the same risk estimation.
    - Exceptions: None

RiskEstimationProcessor

- None

RiskEstimationScheduler

- LaunchRiskEstimation
  - `void launchRiskEstimation(PatientId id, SensorDataPackage data, TimeStamp receivedAt)`
    - Effect: Schedules a risk estimation for the patient identified by `id` due to the arrival new sensor data package `data`. The `receivedAt` parameter indicates when `data` was received by the PMS.
    - Exceptions: None
  - `void launchRiskEstimation(PatientId id, Questionnaire quest, TimeStamp receivedAt)`
    - Effect: Schedules a risk estimation for the patient identified by `id` due to the availability of a new filled in questionnaire `quest` which arrived at the PMS on the `receivedAt` time-stamp.
    - Exceptions: None
  - `void launchRiskEstimation(PatientId id)`
    - Effect: Schedules a risk estimation for the patient identified by `id` due to a change in its clinical models or clinical model configurations.
    - Exceptions: None
- RetrieveRiskEstimation
  - `ClinicalModelJob getNextJob()`
    - Effect: Returns the next clinical model computation job that must be performed (i.e. the first job in the queue).
    - Exceptions: None

SensorDataScheduler

See `SensorDataScheduler` in decomposition 1.

StandbyDB

See `StandbyDB` in decomposition 1.

SubscriberDevice

- DeliverNotification
  - `void deliverNotification(NotificationMessage msg)`
    - Effect: Sends a `msg` to a party subscribed to medical updates.
    - Exceptions: None

SystemAdministratorDevice

See `SystemAdministratorDevice` in decomposition 1.

3.2.6 Data type definitions

ClinicalModel A data element representing a clinical model containing a number of `ClinicalModelParameterKey` and `ClinicalModelParametervalue` pairs.

ClinicalModelId A data element uniquely identifying a clinical model.

ClinicalModelJob A data element representing a request to compute a single clinical model. This contains a `ClinicalModelJobId`, `ClinicalModelId` and `PatientId` respectively uniquely identifying the risk estimation, clinical model and patient this job belongs to. If the risk estimation this job belongs to is triggered by the arrival of a sensor data package this sensor data is also contained in this data element.

ClinicalModelJobId A data element uniquely identifying a ClinicalModelJob.

ClinicalModelParameterKey A data element representing a parameter in a clinical model that can be configured.

ClinicalModelParametervalue A data element representing the value of a parameter in a clinical model.

Echo See `Echo` in decomposition 1.

GatewayId See `GatewayId` in decomposition 1.

HISPatientId A data element uniquely identifying a patient monitored by the PMS in the HIS.

ClinicalModelJobResult This data element represents the result of a single `ClinicalModelJob`. This contains whether the risk level of the relevant patient should change and if so to which risk level.

NotificationMessage See `NotificationMessage` in decomposition 1.

PatientId See `PatientId` in decomposition 1.

PatientRecord This data element represents a patient record containing all medical information for a patient.

Questionnaire This data element represents a questionnaire containing one or more questions with their corresponding answers.

RiskEstimationiId A data element representing uniquely identifying each risk estimation.

RiskEstimationResult A data element representing the result of risk estimation indicating whether the risk level should change, and if so to which risk level.

SensorDataPackage See `SensorDataPackage` in decomposition 1.

StandbyAddress See `StandbyAddress` in decomposition 1.

TimeStamp See `TimeStamp` in decomposition 1.

3.2.7 Verify and refine

This section maps the requirements not yet tackled to the components defined so far. If needed a requirement is split in multiple parts, indicated by the addition of a letter after its identifier.

ClinicalModelDB

- *UC14b*: unregister patient
  Remove clinical model configurations for unregistered patient.
- *M3*: New clinical model/optimized clinical model

GatewayFacade

- *UC3a*: send emergency notification
  Receive emergency notification and forward for processing.
- *UC18*: Translate gateway identifiers to patient identifiers
  This functionality is derived from the need to associate gateways to patients when receiving sensor data.
- *Av1a*: Communication channel between the patient gateway and the PMS
  Detect failure of patient gateway, the telecom infrastructure and internal communication components.
- *M2b*: New type of sensor in the wearable unit
  PMS must be able to accept the new type of sensor data.

NotificationHandler

- *UC7*: notify
- *Av1b*: Communication channel between the patient gateway and the PMS
  Send notifications concerning failures to the PMS system administrators.
- *Av2b*: Availability of the patient record database in the hospital Information System (HIS)
  Send notifications concerning failures to the PMS system administrators.

OtherFunctionality2

- *UC1*: log in
- *UC2*: log off
- *UC3b*: send emergency notification
  Process the emergency notification and forward for storage in the patient record. If necessary contact the emergency services.
- *UC5*: fill out questionnaire
- *UC6*: appoint trustee
- *UC8a*: consult patient status
  Look up patient status and patient record and compile overview for the main actor.
- *UC9a*: configure a clinical model
  Provide interface to cardiologist to request and configure clinical models.
- *UC10*: update risk level
- *UC11*: perform on-demand consultation
- *UC12a*: select clinical models for patient
  Provide interface to the main actor to access and select the available clinical model.
- *UC13*: register patient
- *UC14a*: unregister patient
  Provide means to unregister a patient and log event.
- *UC16*: consult patient record
- *UC17*: update patient record
- *UC19*: Translate internal patient identifiers to HIS patient identifiers (*new use case*)
  This functionality is derived from the need identify patients at the HIS.
- *Av2a*: Availability of the patient record database in the hospital Information System (HIS)
  Detect failure of the HIS and disruptions in the communication channel and store older copy of patient records.
- *P3*: Emergency notifications
- *M1*: Integration of the PMS into a different Hospital Information System
- *M2a*: New type of sensor in the wearable unit
  Client software must be updated and clinical models must be added/updated to cope with the new type of information.

PatientStatusDB

- *UC14c*: unregister patient
  Remove the patient status for the unregistered patient.

PrimaryDB

- None

ReplicationManager

- None

RiskEstimationCombiner

- None

RiskEstimationProcessor

- None

RiskEstimationScheduler

- None

SensorDataScheduler

- None

StandbyDB

- None

3.3 Decomposition 3: GatewayFacade (Av1a)

3.3.1 Module to decompose

In this run we further decompose the `GatewayFacade` component introduced in the first run.

3.3.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *Av1a*: Communication channel between the patient gateway and the PMS
  Detect failure of patient gateway, the telecom infrastructure and internal communication components.

No functional requirements are sufficiently related to the selected non-functional drivers. Therefore, this decomposition has no non-functional drivers.

Rationale. *Av1a* has high priority since it is part of *Av1* split up at the end of the first decomposition. *Av1a* is the last high priority non-functional requirement for the selected `GatewayFacade` component.

3.3.3 Architectural design

Internal failure detection for *Av1a*. To detect the failure of an internal communication component we selected **Ping/Echo**. The `GatewayLogic` component, responsible for the communication with the patient gateway (i.e. the responsibilities of the former `GatewayFacade`), is periodically pinged by the `ComponentAvailabilityMonitor`. If the `GatewayLogic` has failed, the `ComponentAvailabilityMonitor` notifies a system administrator. Note that Av1 does not specify the exact interval for this ping. Also note that the monitoring component has to deployed separately from the component it monitors.

External failure detection for *Av1a*. In order to detect whether a patient gateway or (a part of) the telecom infrastructure has failed we choose **heartbeat**. Considering that a patient gateway already periodically sends sensor data updates we can use these messages as implicit heartbeats. We can further simplify the administration required on the PMS side by having the gateway send, along with the sensor data update, a time-stamp indicating when its next sensor data update should be expected. This allows the PMS to monitor whether the next sensor data update of each patient gateway arrives on time without keeping track of the transmission rates of all patient gateways. Now the PMS only stores the time-stamp for the next sensor data update in the `DeadlineChecker` and monitors whether any of them are overdue. The `GatewayLogic` will forward the piggybacked deadlines received with each sensor data update to the `DeadlineChecker`.

Alternatives considered

Alternatives for ping/echo The failure of an internal component can also be done using a **heartbeat**. This would require both the monitored and monitoring component to keep track of respectively when to sen and receive a heartbeat. This complicates the addition of any monitored components since they would require this extra logic.

Alternatives for heartbeat. An alternative for detecting the failure of patient gateways or a telecom channel is **Ping/Echo**. This would require that the PMS itself keeps track of the transmission rates of each patient gateway and pings each gateway in time. This requires considerable administration and scales difficult to a (very) large number of gateways.

3.3.4 Instantiation and allocation of functionality

This section describes the components which instantiate the selected tactics and patterns, how the related functionality is allocated and how the components are deployed on physical nodes. Unless stated otherwise the responsibilities assigned in the previous decompositions remain unchanged.

Decomposition. Figure 8 shows the component-and-connector diagram of the resulting architecture. For clarity, we only show the decomposition of the `GatewayFacade` with its directly connected components. Notice that with respect to Figure 6, the `GatewayFacade` in Figure 8 employs an additional interface NotifySysAdmin which was not foreseen in the first decomposition.

The responsibilities of the resulting components are as follows:

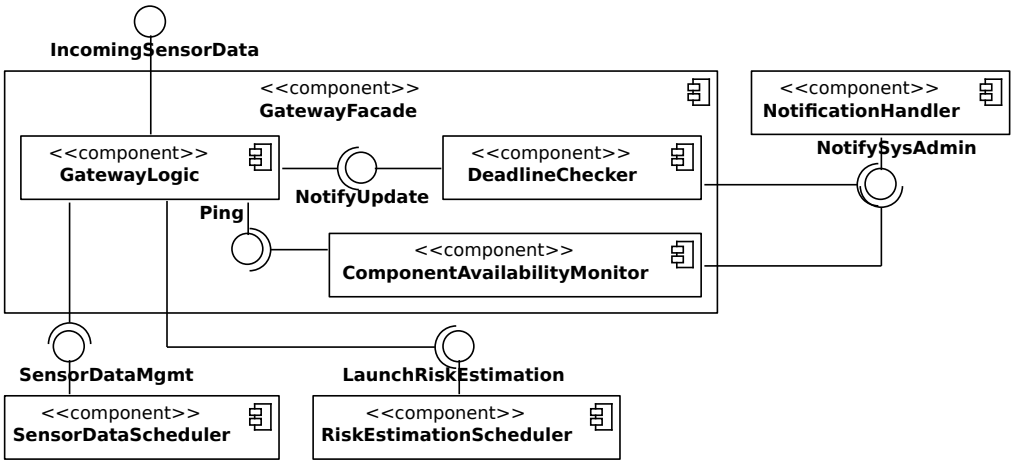

Figure 8: The main component-and-connector diagram resulting from the third decomposition.

ClinicalModelDB See `ClinicalModelDB` in decomposition 2.

ComponentAvailabilityMonitor Responsible for checking the availability of monitored components by sending ping messages to them. If a monitored component does not respond in time, it sends a notification to a PMS system administrator (*Av1a*).

DeadlineChecker Responsible for storing the time-stamps on which the next sensor data update from each patient gateway is expected. This component monitors whether any deadline has been exceeded and if so it requests to send a notification to a PMS system administrator (*Av1a*)

GatewayLogic Responsible for communication with patient gateways. This consists of handling incoming sensor data updates and requesting on-demand updates. Incoming sensor data is first forwarded to the `SensorDataScheduler` after which the update will be forwarded to the `RiskEstimationScheduler`. Furthermore, this component determines to which patient each arrived sensor data update belongs.

NotificationHandler See `NotificationHandler` in decomposition 2..

OtherFunctionality2 See `OtherFunctionality2` in decomposition 2.

PatientStatusDB See `PatientStatusDB` in decomposition 1.

PrimaryDB See `PrimaryDB` in decomposition 1.

ReplicationManager See `ReplicationManager` in decomposition 1.

RiskEstimationCombiner See `RiskEstimationCombiner` in decomposition 2.

RiskEstimationProcessor See `RiskEstimationProcessor` in decomposition 2.

RiskEstimationScheduler See `RiskEstimationScheduler` in decomposition 2.

SensorDataScheduler See `SensorDataScheduler` in decomposition 1.

StandbyDB See `StandbyDB` in decomposition 1.

Deployment. Following the decomposition of the system into new components, Figure 9 shows how the components in the architecture are deployed. Note that the new components are deployed separate from previously existing components. The `GatewayLogic` and `DeadlineChecker` are deployed separate to avoid impact on non-related components when a large number of sensor data updates arrive.

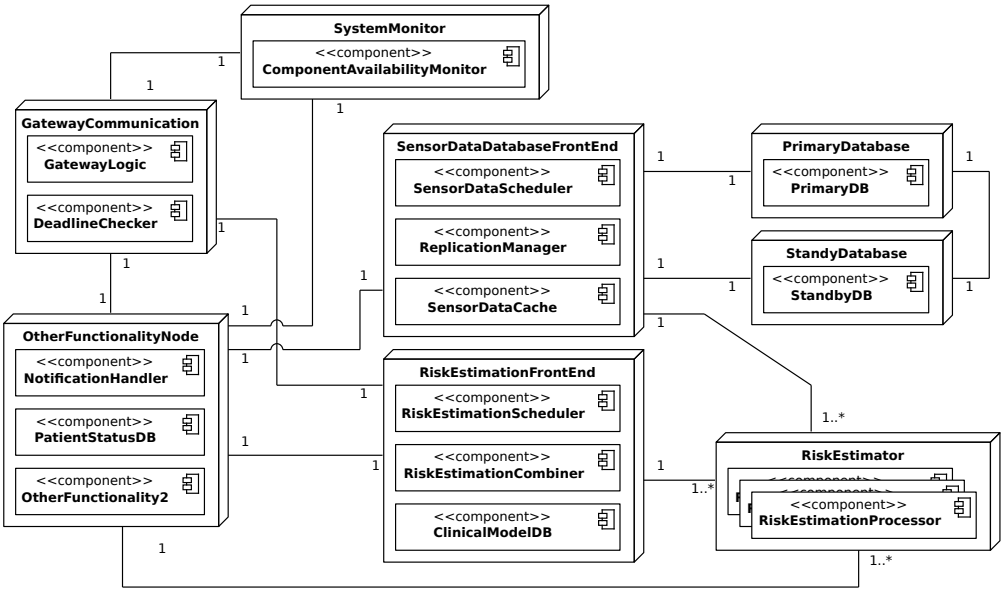

Figure 9: The main deployment diagram resulting from the third decomposition.

3.3.5 Interfaces for child modules

This section describes the interfaces assigned to the components detailed in the above section. For each interfaces is detailed who provides it and what resources the interface provides. The data types used in these interfaces are defined in the following section.

ClinicalModelDB

See `ClinicalModelDB` in decomposition 2.

ComponentAvailabilityMonitor

- None

DeadlineChecker

- NotifyUpdate
  - `void updateReceived(GatewayId id, TimeStamp nextDeadline)`
    - Effect: The `DeadlineChecker` will stop waiting for the previous deadline for the gateway identified by `id` and will start waiting for the next deadline for this gateway specified by `nextDeadline`.
    - Exceptions: None

GatewayLogic

- IncomingSensorData
  - `void sendSensorData(GatewayId id, SensorDataPackage sensorData, TimeStamp nextDeadline)`
    - Effect: Sends a sensor data update `sensorData` to the system for the patient gateway identified by `id`. The `nextDeadline` time-stamp represents the piggybacked heartbeat identifying when the next sensor data update from this gateway is to be expected.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: Returns an echo to the caller indicating that the callee is available.
    - Exceptions: None

HIS

See `HIS` in decomposition 2.

NotificationHandler

See `NotificationHandler` in decomposition 2.

OtherFunctionality2

See `OtherFunctionality2` in decomposition 2.

PatientStatusDB

See `PatientStatusDB` in decomposition 1.

PrimaryDB

See `PrimaryDB` in decomposition 1.

ReplicationManager

See `ReplicationManager` in decomposition 1.

RiskEstimationCombiner

See `RiskEstimationCombiner` in decomposition 2.

RiskEstimationProcessor

See `RiskEstimationProcessor` in decomposition 2.

RiskEstimationScheduler

See `RiskEstimationScheduler` in decomposition 2.

SensorDataScheduler

See `SensorDataScheduler` in decomposition 1.

SubscriberDevice

See `SubscriberDevice` in decomposition 2.

SystemAdministratorDevice

See `SystemAdministratorDevice` in decomposition 1.

StandbyDB

See `StandbyDB` in decomposition 1.

3.3.6 Data type definitions

ClinicalModel See `ClinicalModel` in decomposition 2.

ClinicalModelId See `ClinicalModelId` in decomposition 2.

ClinicalModelJob See `ClinicalModelJob` in decomposition 2.

ClinicalModelJobId See `ClinicalModelJobId` in decomposition 2.

ClinicalModelJobResult See `ClinicalModelJobResult` in decomposition 2.

ClinicalModelParamterKey See `ClinicalModelParameterKey` in decomposition 2.

ClinicalModelParametervalue See `ClinicalModelParametervalue` in decomposition 2.

Echo See `Echo` in decomposition 1.

GatewayId See `GatewayId` in decomposition 1.

HISPatientId See `HISPatientId` in decomposition 2.

NotificationMessage See `NotificationMessage` in decomposition 1.

PatientId See `PatientId` in decomposition 1.

PatientRecord See `PatientRecord` in decomposition 2.

Questionnaire See `Questionnaire` in decomposition 2.

RiskEstimationId See `RiskEstimationiId` in decomposition 2.

RiskEstimationResult See `RiskEstimationResult` in decomposition 2.

SensorDataPackage See `SensorDataPackage` in decomposition 1.

StandbyAddress See `StandbyAddress` in decomposition 1.

TimeStamp See `TimeStamp` in decomposition 1.

3.3.7 Verify and refine

This section maps the requirements not yet tackled to the components defined so far. If needed a requirement is split in multiple parts, indicated by the addition of a letter after its identifier.

ClinicalModelDB

- *UC14b*: unregister patient
  Remove clinical model configurations for unregistered patient.
- *M3*: New clinical model/optimized clinical model

ComponentAvailabilityMonitor

- None

DeadlineChecker

- None

GatewayLogic

- *UC3a*: send emergency notification
  Receive emergency the notification and forward for processing.
- *UC18*: Translate gateway identifiers to patient identifiers
  This functionality is derived from the need to associate gateways to patients when receiving sensor data.
- *M2b*: New type of sensor in the wearable unit
  PMS must be able to accept the new type of information

NotificationHandler

- *UC7*: notify
- *Av1b*: Communication channel between the patient gateway and the PMS
  Send notifications concerning failures to the PMS system administrators.
- *Av2b*: Availability of the patient record database in the hospital Information System (HIS)
  Send notifications concerning failures to the system administrators.

OtherFunctionality2

- *UC1*: log in
- *UC2*: log off
- *UC3b*: send emergency notification
  Process the emergency notification and forward for storage in the patient record. If necessary contact the emergency services.
- *UC5*: fill out questionnaire
- *UC6*: appoint trustee
- *UC8a*: consult patient status
  Look up patient status and patient record and compile overview for the main actor.
- *UC9a*: configure a clinical model
  Provide interface to cardiologist to request and configure clinical models.
- *UC10*: update risk level
- *UC11*: perform on-demand consultation
- *UC12a*: select clinical models for patient
  Provide interface to the main actor to access and select the available clinical model.
- *UC13*: register patient
- *UC14a*: unregister patient
  Provide means to unregister a patient and log event.
- *UC16*: consult patient record
- *UC17*: update patient record
- *UC19*: Translate internal patient identifiers to HIS patient identifiers
- *Av2a*: Availability of the patient record database in the hospital Information System (HIS)
  Detect failure of the HIS and disruptions in the communication channel and store older copy of patient records.
- *P3*: Emergency notifications
- *M1*: Integration of the PMS into a different Hospital Information System
- *M2a*: New type of sensor in the wearable unit
  Client software must be updated and clinical models must be added/updated to cope with the new type of information.

PatientStatusDB

- *UC14c*: unregister patient
  Remove the patient status for the unregistered patient.

PrimaryDB

- None

ReplicationManager

- None

RiskEstimationCombiner

- None

RiskEstimationProcessor

- None

RiskEstimationScheduler

- None

SensorDataScheduler

- None

StandbyDB

- None

4 Resulting partial architecture

This section describes the architecture of the Patient Monitoring System (PMS) resulting from the attribute-driven design (ADD) process described throughout the above sections. This architecture is only *partial* (i.e. not complete), since the design process was not completed yet. Therefore, the resulting architecture also still contains an `OtherFunctionality2` component.

The rest of this section is structured as follows. First, we provide the context diagram of the PMS. Second, we show the structure of the system in terms of its components using a component-and-connector view. Finally, we show the allocation of these components to physical nodes using the deployment view. We do not repeat the list of components, their responsibilities or their interfaces since they are identical to the result the third decomposition and can be found there.

4.1 Context diagram

Figure 10 shows the context diagram for the component-and-connector view. This figure shows the system as a whole, the external components it interfaces with and the internal components that are used by the external components.

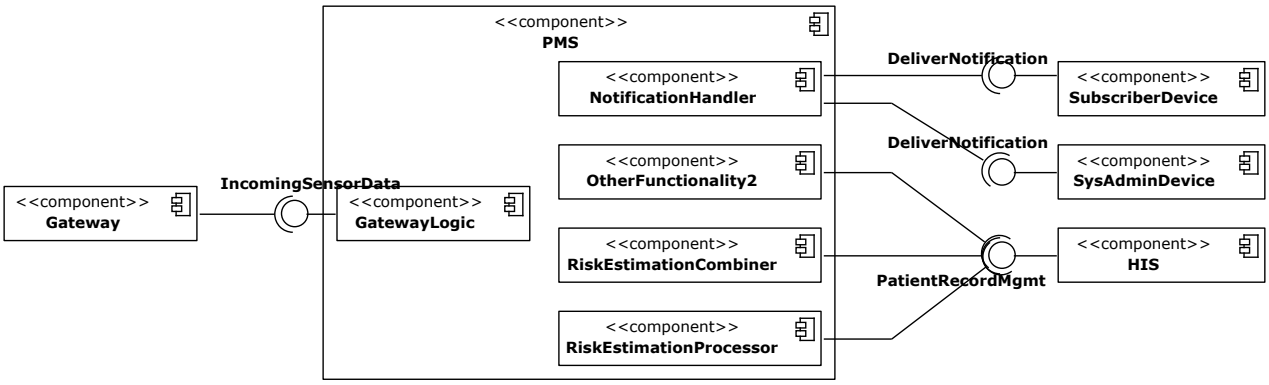

Figure 10: The context diagram for component-and-connector view of the final architecture.

4.2 Component-and-connector view

Figure 11 shows the primary diagram of the component-and-connector view. For readability reasons, we chose to bundle the components related to storing sensor data into the new super-component `SensorDataDB` and the components related to risk estimations into the new super-component `RiskEstimator` and keep the decomposition of the `GatewayFacade` as shown in the third decomposition. Their respective decompositions are shown in Figure 12, Figure 13 and Figure 14.

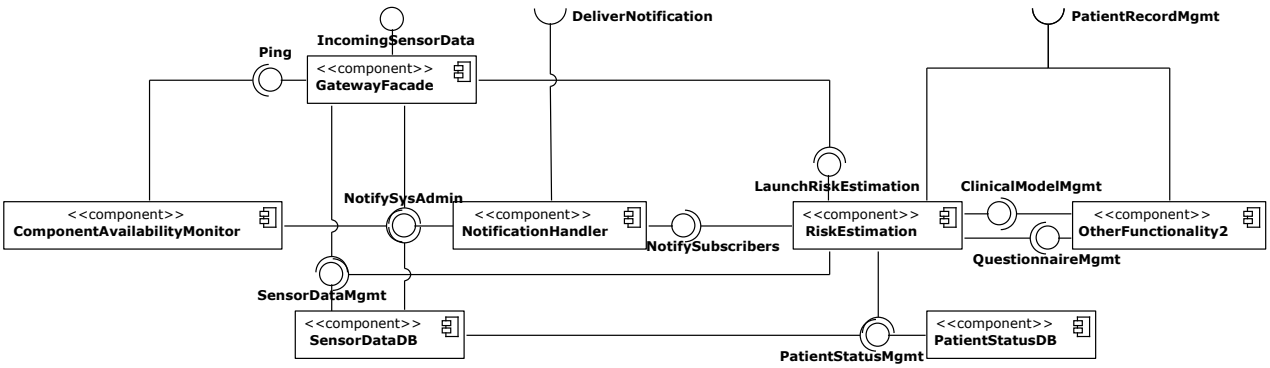

Figure 11: The component-and-connector diagram of the final architecture. Dangling interfaces are provided or employed by components external to the PMS (see Figure 10).

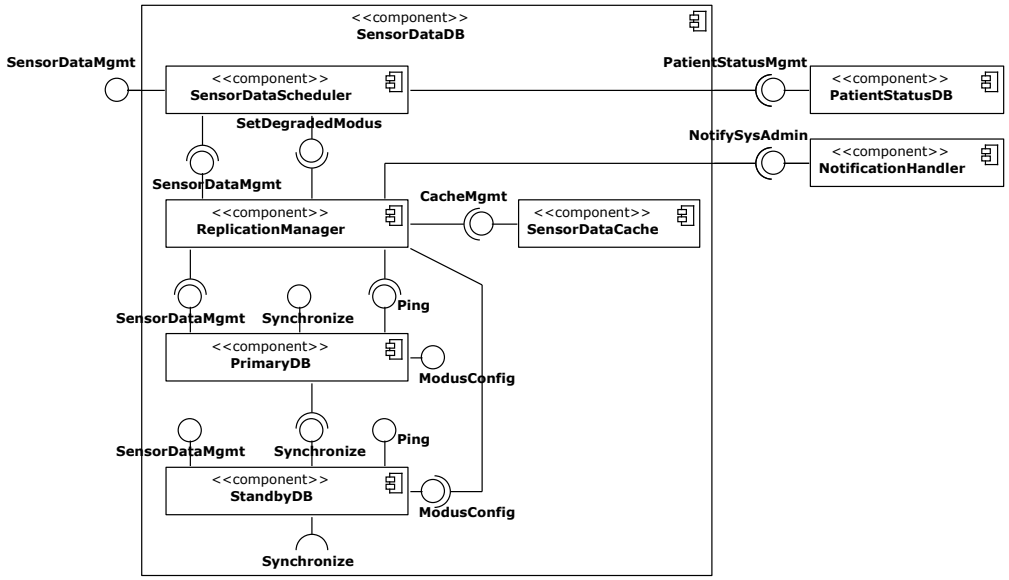

Figure 12: The decomposition of the `SensorDataDB` of Figure 11.

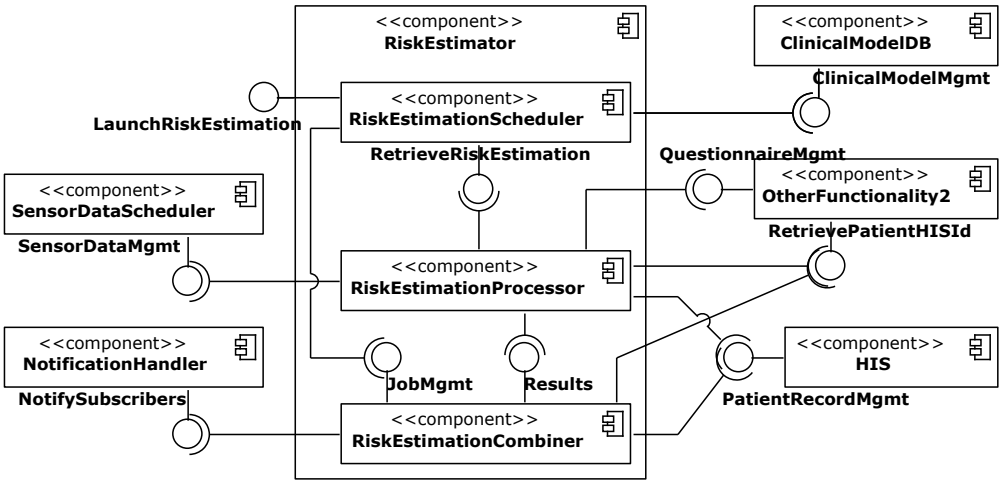

Figure 13: The decomposition of the `RiskEstimator` of Figure 11.

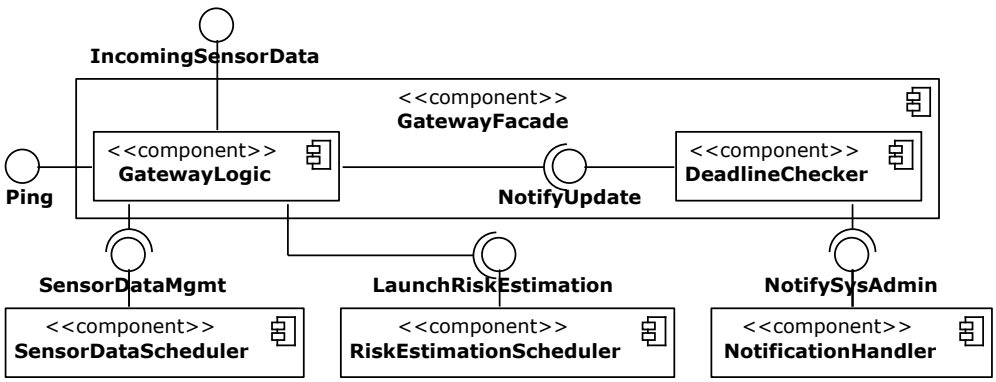

Figure 14: The decomposition of the `GatewayFacade` of Figure 11.

4.3 Deployment view

Figure 15 shows the context diagram for the deployment view. This figure shows that communication between the patient gateway and the PMS happens through the telecom infrastructure over TCP. The communication with the HIS happens over TCP. The PMS communicates with the `SubscriberDevice` and `SysAdminDevice` using SMTP, SMS or another protocol. Notice that the choices to employ TCP were made because the exact protocols used will depend on the HIS and the Gateway (which are components that we do not control). These protocols were not known in the design of this architecture. However, in general, it is best to specify the protocol as precisely as possible.

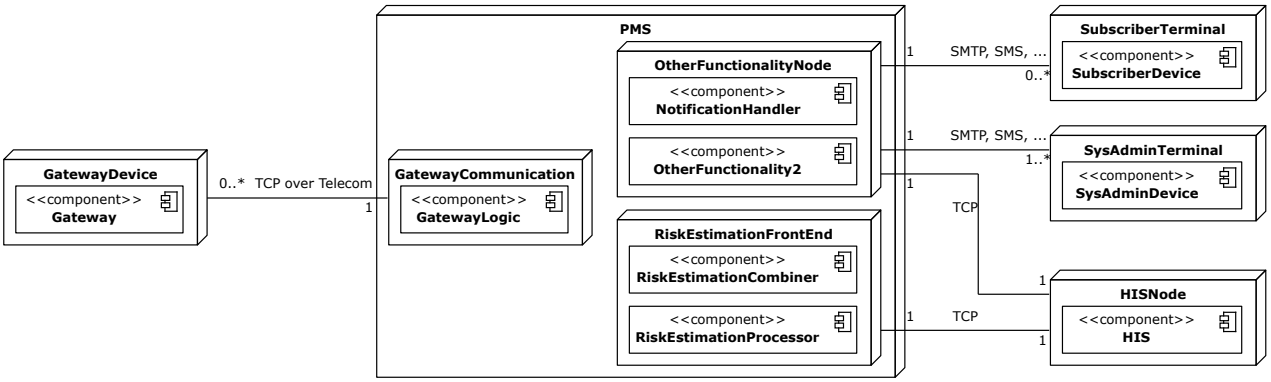

Figure 15: The context diagram for deployment view of the final architecture.

Figure 16 shows how each component is allocated to physical nodes. The `SensorDataDB` is deployed over several nodes. The `SensorDataDatabaseFrontEnd` hosts the scheduling and replication management of the databases and serves as the access point to sensor data to the rest of the system. The `PrimaryDatabase` and `StandbyDatabase` represent the actual database respectively having the primary and standby roles. See the first decomposition for more information. The `RiskEstimator` is deployed over multiple nodes to allow scaling the sub-system up when needed. The `RiskEstimationFrontEnd` is the access point to the risk estimation sub-system and manages the risk estimation requests from the rest of the system. The `RiskEstimationProcessor` can be replicated as required by the system load. The `GatewayFacade` is deployed separately to avoid its failure affects the availability of other parts of the system as explained in the third decomposition.

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 2nd ed.* Addison-Wesley, 2009.

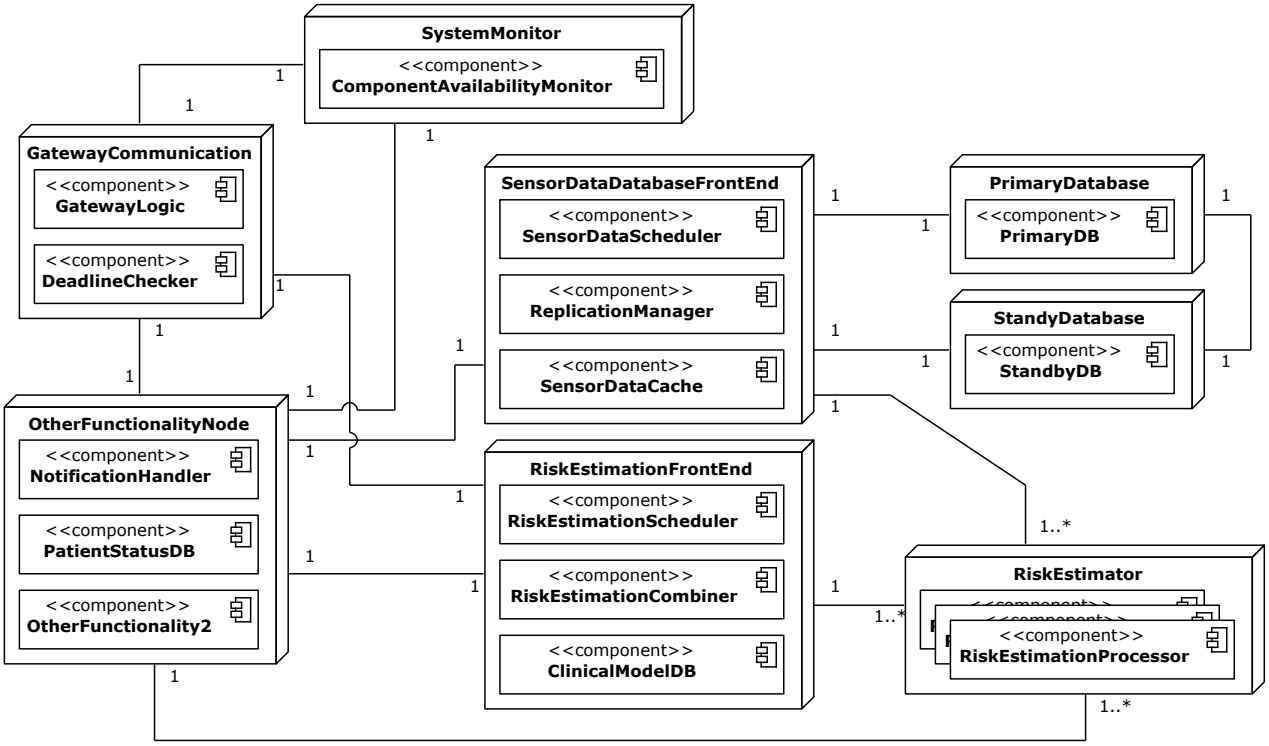

Figure 16: The component-and-connector diagram of the final architecture.