Patient Monitoring for Cardiovascular Diseases

The complete architecture

Abstract

The goal of this project is to design a patient monitoring system for cardiovascular diseases. In this document, we describe the final architecture, which has been designed based on the requirements from the domain analysis and the priorities of these requirements given by our stakeholders. This document provides (1) an overview of the system in terms of the main architectural decisions, (2) the context diagram of the system, (3) its main decomposition into components, (4) the decompositions of the most important of these components, (5) the deployment of these components, (6) the illustration of how the main scenarios are fulfilled by the architecture and (7) the description of all components, their interfaces and the defined data types.

Contents

1 Introduction

The goal of this project was to design a patient monitoring system for cardiovascular diseases. In this document, we describe the final architecture, which was designed based on the requirements from the domain analysis and the priorities of these requirements given by our stakeholders. Section 2 lists the architectural decisions for all non-functional requirements and discusses the final architecture. Section 3 provides and discusses the main context diagram of our architecture (i.e., the context diagram of the component-and-connector view). Section 4 provides the main decomposition of the architecture as component-and-connector diagram and discusses the main architectural decisions. Section 5 provides and discusses the more fine-grained decompositions of some of the major components in the main decomposition. Section 6 provides and discusses the deployment of the components of the component-and-connector view on physical nodes. Finally, Section 7 illustrates how our architecture accomplishes the most important functionality and data flows using sequence diagrams. Afterwards, Appendix A lists and describes all components of the component-and-connector view and their interfaces and Appendix B lists and describes the data types used in these interfaces.

For clarity, we here also repeat the priorities of the requirements as given by the stakeholders (we do not require you to do this in your report):

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.

2 Overview

This section gives a high-level but complete overview of the system: it lists the design decisions for *all* non-functional requirements and provides a discussion concerning the strong and weak points of the architecture.

2.1 Architectural decisions

In this section, we give an overview of the architectural decisions made in our architecture in order to achieve the requirements given in the domain analysis.

- **Av1: Communication channel between the patient gateway and the PMS.** *Av1* requires the system (1) to be able to detect the lack of updates from a certain gateway and (2) to be able to detect the failure of the internal communication sub-system responsible for communicating with the gateways.
  To achieve the former, the `GatewayFacade` keeps track of the next deadline for each gateway, which the gateway provides with each new sensor data update. As such, these updates are considered as implicit heartbeats. To achieve the latter, the `GatewayFacade` is pinged by the `AvailabilityMonitor`, which is deployed on another node. For more details, we refer to Section 4.1.1, Section 5.2 and Section 6.
  Employed tactics and patterns: heartbeat, ping/echo.

- **Av2: Availability of the patient record database in the HIS.** *Av2* requires the system to be able to detect the failure of the HIS or the communication channel between the HIS and the PMS and continue operation in degraded modus when failure occurs.
  To achieve this, the `HISFacade` periodically pings the HIS. In normal modus (i.e., when the HIS is functioning correctly) the `HISFacade` immediately forwards write requests to the HIS, serves read requests for patient records directly from the HIS and caches the results for 24 hours. When the HIS fails, the `HISFacade` notifies a system administrator using his client. In degraded modus (i.e., when the HIS has failed) this `HISFacade` serves read requests of patient records from its cache and temporarily buffers write requests until the HIS is available again. For more details, we refer to Section 5.1.
  Employed tactics and patterns: ping/echo, caching, buffering

- **Av3: Internal PMS database failure.** *Av3* requires the database storing the sensor data (1) to be separated from other data, (2) to have a back-up database in order to survive failures and (3) be able to seamlessly transition to the back-up database in case of failure.
  To achieve this, the `SensorDataDB` is separated from other databases, both logically and physically, and is internally replicated into a primary and a back-up database using passive repdundancy, with a cache to avoid losing data in between two synchronizations. For more details, we refer to Section 4.1.2, Section 5.4 and Section 6.
  Employed tactics and patterns: logical separation, physical separation, replication, passive redundancy, caching, state resynchronization

- **P1: Storage of sensor data readings.** *P1* requires the database storing the sensor data to process new sensor data before certain deadlines and schedule them based on the risk level of the respective patients in case these deadlines cannot be attained.
  To achieve this, the `SensorDataDB` internally queues new sensor data, monitors their processing time and schedules them based on the given deadlines. Since read requests to the database can also impact the performance of storing new data, the `SensorDataDB` also actively schedules read requests. For more details, we refer to Section 4.1.3 and Section 5.4.
  Employed tactics and patterns: queuing and scheduling, dynamic priority scheduling: earliest deadline first, fixed priority scheduling with semantic importance

- **P2: Risk estimation by clinical models.** *P2* requires the system (1) to balance the multiple risk level estimations over multiple instances and (2) schedule new risk estimations based on the throughput of the risk estimation sub-system.

To achieve the former, the `RiskEstimator` internally employs multiple risk estimation processors, replicated on multiple nodes and multiple times on each node. To achieve the latter, the `RiskEstimator` queues new risk estimation requests, monitors the throughput and schedules them according to *P2*. For more details, we refer to Section 4.1.4 and Section 5.3.
Employed tactics and patterns: queuing and scheduling of commands, dynamic priority: earliest deadline first, first-in/first-out, replication on multiple physical nodes, concurrency on a single node

- **P3: Emergency notifications.** *P3* requires emergency notifications to be verified within a hard deadline of 5 ms and achieve a minimal throughput of 4 emergency notifications per second.
To achieve this, the `GatewayFacade` receives emergency notifications on a dedicated node and the `EmergencyHandler` is also deployed on a separate node, both in order to minimize the impact of the load of the rest of the system on emergency verification and allow the emergency verification flow to be optimized independently. For more details, we refer to Section 4.1.5 and Section 6.
Employed tactics and patterns: logical separation, physical separation

- **M1: Integration of the PMS into a different HIS.** *M1* requires integrating a different HIS (1) to have low development overhead and (2) not to have impact on functionality other than patient record access, notification delivery and emergency notification delivery. To achieve this, the `HISFacade` encapsulates all communication with the HIS for the mentioned functionality. As a result, only this component has to be altered for integrating with a different HIS (as long as the functionality remains similar, of course).
Employed tactics and patterns: maintain existing interfaces using a facade/adapter

- **M2: New type of sensor in the wearable unit.** *M2* requires adding a new type of sensor in the wearable unit (1) to have low development overhead for the system in terms of the interfaces, and the resulting new clinical models and (2) not to have impact on the notification system or the interaction with the HIS.
The former is achieved by keeping the interfaces generic, i.e., by employing a generic data type consisting of the type of measurement and the value of the measurement itself. The same holds for having no impact on the interaction with the HIS. The `SensorDataDB` stores sensor data in a similar way. By storing the clinical models themselves in a database (see *M3*), new clinical models can also be added easily, without having to change the code itself. Having no impact on the notification system is achieved by the way these functionalities are realized: processing sensor data does not interfere with sending notifications.
Employed tactics and patterns: anticipate expected changes

- **M3: New clinical model/optimized clinical model.** *M3* requires adding or updating a clinical model without down-time, not to affect ongoing risk estimations and not to take longer than 30 minutes.
All three requirements are achieved by storing the clinical models themselves in a database and fetching these models (and their configuration for each individual patient) at run-time.
Employed tactics and patterns: defer binding time using database storage

- **Multi-tiered architecture and facades.** While it is not an explicit requirement, we chose to structure the complete architecture as three coarse tiers: a presentation tier with externally available components, a business tier containing business logic and a database tier. These tiers provide modifiability of each individual tier. As a result of the presentation tier, we also chose to apply facades to all external interfaces. Of these facades, the `GatewayFacade` and `HISFacade` contribute to the realization of the non-functional requirements *Av1*, *Av2* and *P3*. However, also employing facades for all other external interfaces encapsulates the internal components, interfaces and functionality so that they can vary without changing the external interfaces. Moreover, the facades also allow to encapsulate perimeter security such as transmission encryption, input validation and demilitarized zones.
Employed tactics and patterns: multi-tiered architecture, facade

- **Caching of clinical models and their configurations.** While it is not an explicit requirement, we chose to cache clinical models and their configurations for each individual patient close to the

`RiskEstimator`. Since the assignment of clinical models and their configurations can be presumed to change infrequently, this cache is expected to avoid remote traffic for the majority of requests to the `OtherDataDB` and speed up the process of evaluating the clinical models as such. Because of the infrequent updates and the need for cache consistency, the cache has to be explicitly invalidated by the `OtherDataDB` (instead of working with time-outs). For more details, we refer to Section 5.3 and Section 6.
Employed tactics and patterns: caching, explicit invalidation

2.2 Discussion

In this document, we describe the architecture of the PMS, which was designed based on the requirements from the domain analysis and the priorities of these requirements given by our stakeholders. In terms of the requirements, there were no hard trade-offs. The requirements were handled based on their relative priorities, but in the end, all requirements were compatible and are supported in the final architecture.

In terms of availability, we tackled the requirements given by our stakeholders. As such, no sensor data will be lost as the result of the failure of a single node and the system will continue to function in the presence of failure of the HIS. However, as the architects of this system, we want to state that there are still single points of failure in our architecture. For example, none of the facades are replicated and as a result, the failure of a single node can make the system unavailable for a large number of the end-users. Similarly, databases of data other than the sensor data are not replicated and as a result, the failure of a single database node can lead to the loss of, for example, all administrative data of patients. Recovering this data without a back-up is nearly impossible and will lead to large economical costs for our company.

In terms of performance and scalability, the largest bottlenecks in the system are able to scale out. As a result, this architecture can be expected to support a large number of users when the system grows. As architects of this system, we believe that the first limits to scalability will be the end-user facades, which are now located on a single node. However, we believe that replicating these facades in order to scale out to a larger number of users will not lead to high costs, since both are stateless because of the separate authentication handler and session database.

As such, we believe that this architecture supports all given requirements and will be able to support a growing number of end-users, but that more care should be given to availability in order for the system and our company to survive in the long term.

3 Context diagram

The context diagram of the component-and-connector view is given in Figure 1. As shown, five distinct types of external components communicate with the system: (i) the `Gateway` represents the gateway devices close to each monitored patient, (ii) the `HIS` represents the hospital information system (which can be simplified into a single component from the point of view of the PMS), (iii) the `SysAdminDevice` represents the device of a system administrator used to receive notifications sent by the PMS (e.g., an e-mail endpoint), (iv) the `HospitalUsersClient` represents the client of users employed at the hospital (i.e., cardiologists and nurses) and (v) the `OtherUsersClient` represent the clients of users not employed at the hospital (i.e., telemedicine operators, patients, trustees, home caretakers and GPs). Notice that each external interface is provided by a facade component and that each type of external component is served by a separate facade component.

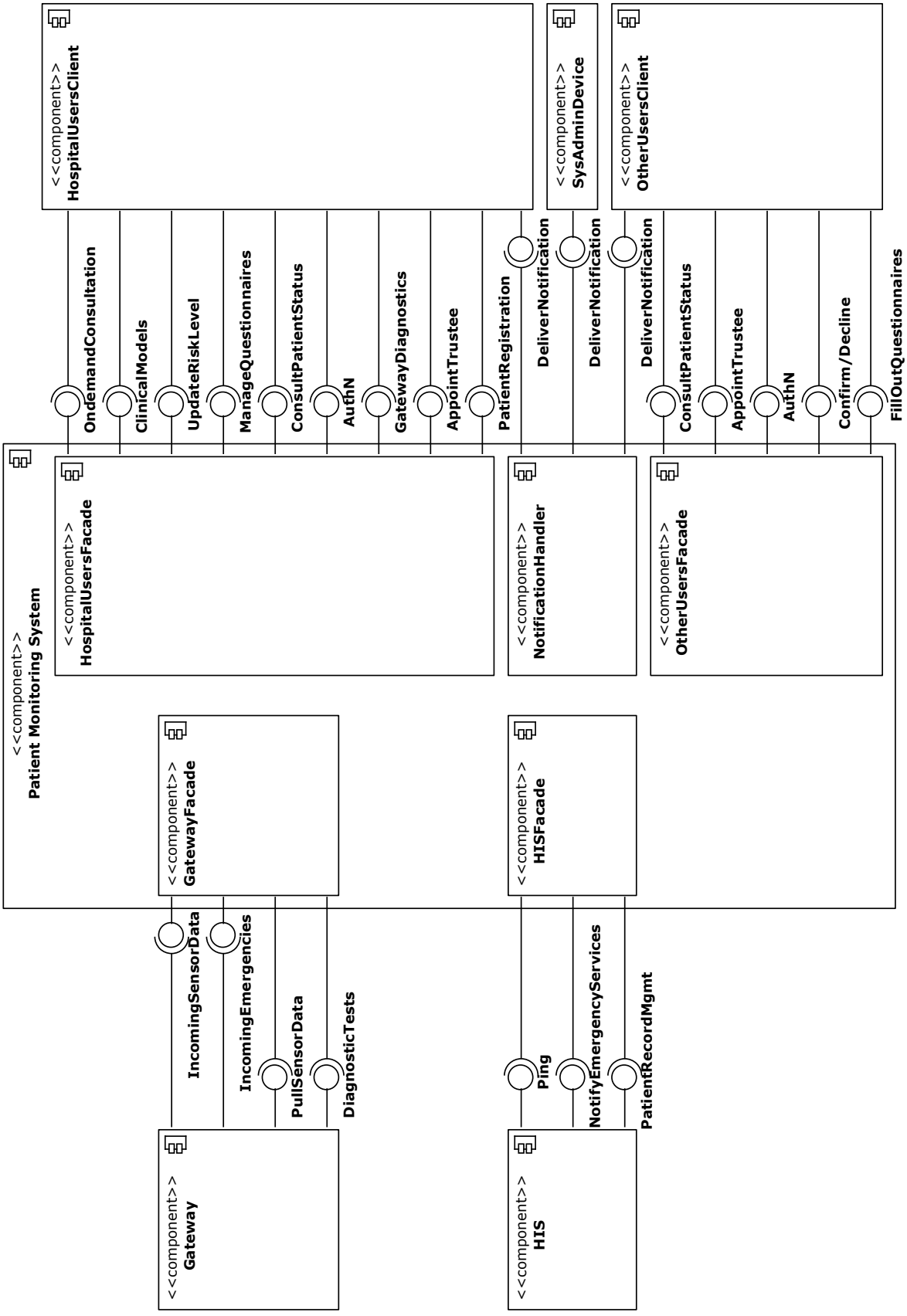

Figure 1: The context diagram of the component-and-connector view.

4 Main component diagram

The main component-and-connector view is shown in Figure 2. As shown, the complete architecture is largely structured as three coarse **tiers**: (i) the presentation tier containing the facades, which are externally available, (ii) the business tier containing the components handling the business logic (e.g., the `PatientStatusOverviewHandler` which generates the status reports end the `RiskEstimator` which executes the risk models) and (iii) the database tier containing the components storing the actual data. Notice that we do not employ the tiers strictly, i.e., we still allow direct connections between components of the presentation tier and the database tier.

The `HISFacade` is further decomposed in Section 5.1, the `GatewayFacade` in Section 5.2, the `RiskEstimator` in Section 5.3 and the `SensorDataDB` in Section 5.4. The descriptions of all components in Figure 2, their sub-components and their interfaces are given in Section A.

Note that Figure 2 does not show the connections of the NotifySysAdmin interface for readability reasons, as is also noted in the diagram itself.

4.1 Main architectural decisions

Figure 2 provides the overall description of the architecture and illustrates the most important architectural decisions for achieving the required qualities. Here we document how our architecture fulfills the most important qualities (based on the priorities given by our stakeholders), i.e., *Av1: Communication channel between the patient gateway and the PMS*, *Av3: Internal PMS database failure*, *P1: Storage of sensor data readings* and *P2: Risk estimation by clinical models*. In addition, we also highlight *P3: Emergency notifications* (because of the importance of this quality to critical patients) and the use of facades. Notice that the most important quality attribute scenarios can be encapsulated cleanly because there is no or little interaction between them. However, *P3* and the modifiability scenarios are more cross-cutting and affect multiple components.

4.1.1 Av1: Communication channel between the patient gateway and the PMS.

Firstly, *Av1* requires the system to be able to detect the failure of the communication channel between the patient gateway and the PMS. To achieve this, the `GatewayFacade` monitors the arrival of new sensor data based on the current schedule of each gateway, i.e., the `GatewayFacade` treats each sensor data update as an implicit **heartbeat** from the `Gateway`. The gateway sends a time-stamp indicating when its next update is scheduled together with each sensor data update. Thus the `GatewayFacade` must only monitor whether each patient gateway achieves its next deadline without having to keep track of the transmission rate of each gateway. In case an update is missing, the `GatewayFacade` notifies a system administrator within five minutes from the expected deadline and logs the time of the communication failure.

Secondly, *Av1* requires the system to be able to differentiate between failure of the communication channel and failure of the facade itself. Therefore, the `GatewayFacade` can be pinged to check for liveliness. This ping is performed every two minutes by the `ComponentAvailabilityMonitor` located outside of the `GatewayFacade`. In case of failure, the `ComponentAvailabilityMonitor` notifies a system administrator via the `NotificationHandler` and keeps track of the time of failure.

Thirdly, *Av1* requires a system administrator to be notified in case of failure. As explained above, this is handled by the `GatewayFacade` and the `ComponentAvailabilityMonitor` using the `NotificationHandler`.

The `GatewayFacade` is further decomposed in Section 5.2.

Alternatives considered

Alternative for the bookkeeping of the gateway deadlines. In our architecture, the gateways themselves share their next deadline when sending a sensor data update to the PMS. An alternative would have been to also maintain the deadlines (i.e., the transmission schedules) of all gateways in the PMS (i.e., in the `GatewayFacade`). In that case, the PMS would check the availability of the gateways and the telecom infrastructure using **Ping/Echo** instead of using the sensor data updates as implicit heartbeats. However, it is clear that the server-side bookkeeping of the gateway deadlines would duplicate

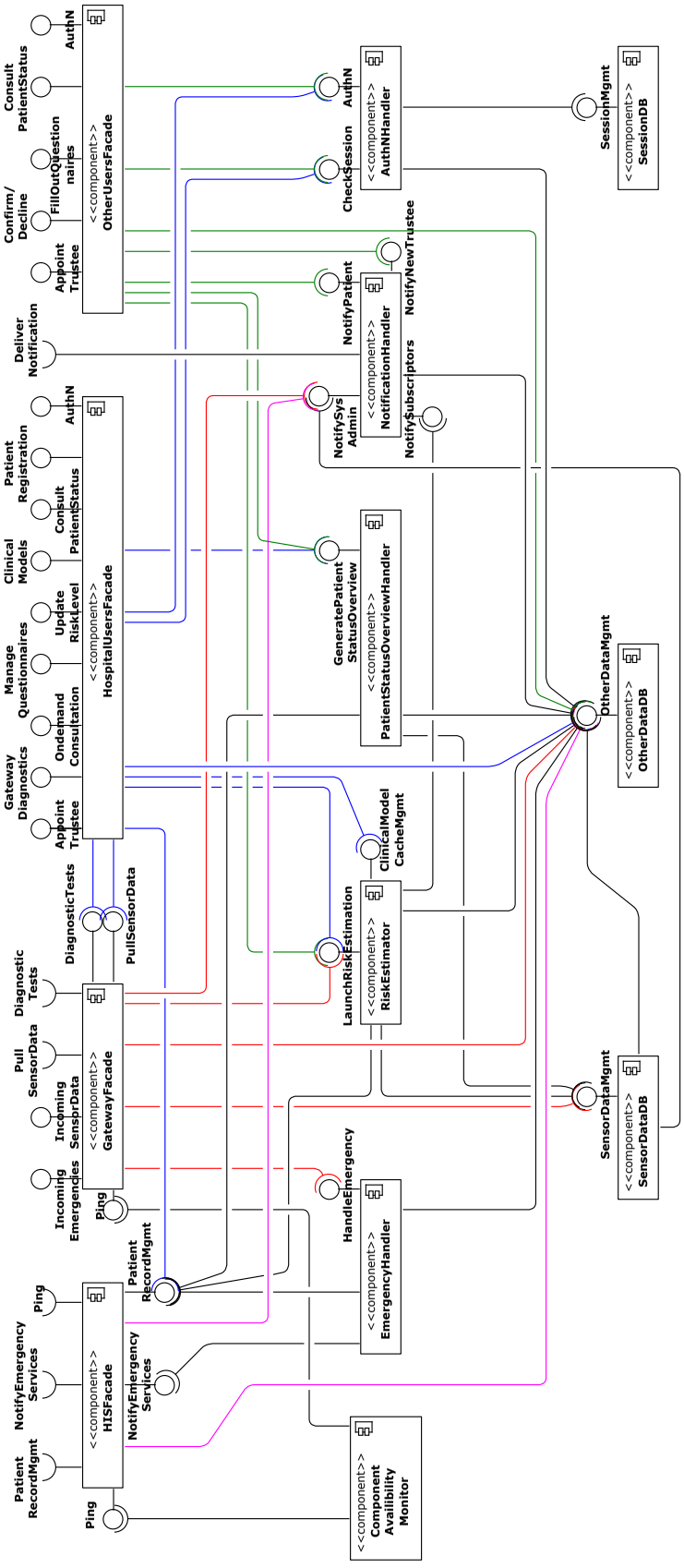

Figure 2: The main component-and-connector view. As shown, the complete architecture is largely structured as three coarse tiers: a tier of front-end nodes hosting facades, a tier of business nodes hosting business logic such as clinical model processing and a tier of databases.

the transmission schedules at the gateway and at the `GatewayFacade`, increasing the complexity of the system by having to maintain consistency. Our current solution avoids this complexity.

4.1.2 Av3: Internal PMS database failure.

Firstly, *Av3* requires the sensor database to fail independently of other databases in the system. Therefore, this database is separated both logically and physically (see the deployment view in Section 6) from the others.

Secondly, *Av3* requires the system to maintain a backup database and seamlessly switch to this database in case the main database fails. Therefore, the `SensorDataDB` internally consists of a primary database and a back-up database (see Section 5.4). The primary database and the back-up synchronize using **passive redundancy**, i.e., the primary processes all reads and writes by itself and periodically pushes new data to the back-up (or *standby*) to synchronize. In order to avoid losing the data written to the primary after the last synchronization, the `SensorDataDB` keeps this data in a cache as well and flushes this to the back-up upon failure. Note that while *Av3* only requires the sensor data of patients with yellow and red risk levels to be backed up, our solution also backs up the sensor data of patients with green risk level.

Thirdly, *Av3* allows the system to omit gateway updates of patients with green risk levels in case of overload in degraded modus. Therefore, the `SensorDataDB` internally queues new sensor measurements. If needed, this queue can omit new measurements of patients with green risk levels.

The `SensorDataDB` is further decomposed in Section 5.4.

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 is known to have 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.

Sharding the sensor data. As mentioned above, *Av3* only requires backing up the sensor data of patients with yellow and red risk levels. As such, the sensor data could be **sharded** over two databases: one for patients with green risk levels and one for the other patients. However, sharding the data would lead to a larger number of nodes and, more importantly, a more complex architecture since patients can change risk level and their data would then be distributed or will have to be moved from the one database to the other. Moreover, sharding the sensor data would allow sensor data of patients with green risk level to be lost in case of hard failure of the primary database, while backing up all sensor data would actually not introduce large hardware costs. Therefore, we chose not to shard the sensor data and back up the sensor data of all patients, i.e., also back up the sensor data of patients with green risk level.

4.1.3 P1: Storage of sensor data readings.

Firstly, *P1* requires the system to monitor how long it takes to store new sensor data readings into the database and go into overload modus if needed. Therefore, the `SensorDataDB` queues all new sensor data readings, tracks their processing time and sorts all elements in the queue as required by *P1*. If needed, the queue can also drop entries for patients with a green risk level.

In normal modus (i.e., when the required deadlines are met), the incoming sensor data updates are written to the database according to a **dynamic priority scheduling: earliest deadline first** policy using the deadlines as described in the quality scenario. In overload modus, new sensor data readings are stored according to 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. This means that starvation can occurs for sensor data updates belonging to patients with a yellow or green

risk level. To fetch an entry from the queue, the other component pops the top of the queue, while the queue itself determines the sequence of the elements in the queue.

Notice that next to storing new sensor data readings, the `SensorDataDB` is also read by other components. Because reading the database also affects the performance of storing new sensor data readings, read requests are also scheduled by the queue. Because explicit deadlines are given for storing new sensor data readings, these are given priority over read requests. However, to avoid starvation of read requests, they are also taken into account by the scheduler using an implicit deadline of five minutes (longer than any storage request).

Secondly, notice that it can be the case that new sensor data is stored and shortly after requested again. For example, for now, we assume 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.

Finally, also notice that the deadlines imposed by *P1* could be interpreted as end-to-end, while the current architecture does not take into account the time that new sensor data readings spend in the `GatewayFacade`. As such, the current architecture assumes this time to be as low as possible.

The `SensorDataDB` is further decomposed in Section 5.4.

Alternatives considered

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 `SensorDataDB` 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 it is finished. (3) Finally, the third alternative is to remedy the race condition by making the `SensorDataDB` aware of the workflow of storing and reading sensor data for risk estimations 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 workflow when needed.

Distributing read requests. In order to improve the read performance of the sensor data database, read requests could be distributed over the multiple replicas of the database. This way, the maximal read throughput of the database would scale with the number of replicas. However, as discussed for *Av3*, we chose to opt for passive redundancy with periodic synchronization. This suits the requirements given by our stakeholders, favors the write performance and leads to a rather simple architecture. However, since the replicas are not up-to-date, they cannot be used for handling read requests. Because the read performance of the sensor data database is not an explicit requirement, we still opted for periodic synchronization.

4.1.4 P2: Risk estimation by clinical models.

Firstly, *P2* requires the risk estimation to be balanced over multiple instances. Therefore, the `RiskEstimator` internally replicates the processing sub-component in order to process (potentially different) risk estimations in parallel. These processing nodes asynchronously pop risk estimation jobs (or *commands*, since this is an instance of the **Command** pattern) from a queue. Since risk estimations do not read data written by other risk estimations, there are no race conditions and there is no need for concurrency control.

Replicating the risk estimation processors can be achieved on two levels. First, multiple instances can be instantiated on **multiple physical nodes**. As a result, multiple risk estimations can be performed concurrently. Second, we also introduce **concurrency on a single node** by having each risk estimation node also host multiple instances. 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 may 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.

Secondly, *P2* requires the system to track the processing time of risk estimations and go into overload modus when needed (i.e., when more than 20 risk estimations per minute are required). In overload modus, the jobs in the queue are scheduled based on the risk level of the respective patients. Therefore, the `RiskEstimator` continuously monitors the throughput of incoming jobs and changes the scheduling algorithm of the internal queue when needed. In normal modus, the internal queue will return the jobs in a **first-in/first-out** order. In overload modus, the queue will return using **dynamic priority: earliest deadline first**. 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.

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.

Alternatives considered

Dynamic scaling. 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 required 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 by dynamically 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.

Caching input data. Depending on the exact risk estimation model used, a risk estimation can require large amounts of input data, possibly from multiple databases and possibly from external databases. Our current architecture incorporates caching of the clinical models and their configurations (see Section 5.3).

However, it does not yet incorporate solutions to improve the performance of fetching this required data in order to avoid this extra complexity, which was not explicitly required. Should this pose a problem in the future, the architecture can be extended. For example, in order to speed up the complete risk estimation process, data can be fetched and cached asynchronously when a risk estimation job enters the queue instead. As another example, external database with possibly large response times could asynchronously be cached or duplicated locally. Notice that depending on the consistency requirements of these data, race conditions could complicate both examples.

4.1.5 P3: Emergency notifications.

P3 requires emergency notifications to be processed within a strict deadline of five milliseconds and achieve a throughput of at least four emergency notifications per second. To achieve this, the emergency clinical models are light-weight and do not rely on the patient's history. Moreover, to make sure that this deadline is maintained even in high load of the rest of the system, the processing of emergency notifications is separated from the rest of the system both logically and physically (e.g., the `EmergencyHandler` is deployed on a dedicated node, see the deployment view in Section 6). For this reason, the Emergencies interface of the `GatewayFacade` is provided by a separate component (see the decomposition of this component in Section 5.2), so that the emergency flow is separate from the start and can even be deployed on separate networking infrastructure. If needed, the emergency flow can also be autonomously upgraded with newer (i.e., better) hardware for improved latency and the `EmergencyHandler` can be replicated for improved throughput. Notice that replicating the `EmergencyHandler` is fairly easy since it does not require other data and does not write any data, thereby avoiding the need for concurrency control.

Alternatives considered

Handling emergencies in the normal sensor data flow. An alternative to separating the processing of emergency notifications from the processing of the normal sensor data would have been to merge both flows. The emergency notifications could then be given priority over normal sensor data in every scheduler in order to process them as quickly as possible. However, this would only increase the complexity of the normal sensor data flow, while its load would still affect the performance of processing emergency notifications and the strict deadline of five milliseconds would not be guaranteed. We argue that the life-threatening nature of the emergencies justifies the extra hardware cost of the separate emergency flow and opted for this.

5 Key decompositions

5.1 HISFacade

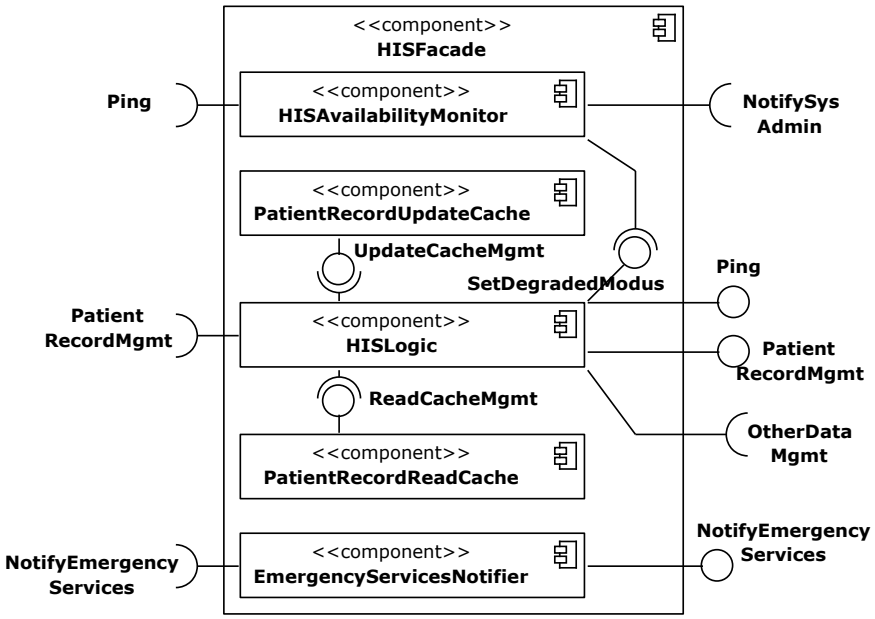

Figure 3: Decomposition of the `HISFacade`.

The decomposition of the `HISFacade` is presented in Figure 3. As shown, the `HISFacade` is decomposed into five submodules: the `HISLogic`, the `HISAvailabilityMonitor`, the `PatientRecordReadCache`, the `PatientRecordUpdateBuffer` and the `EmergencyServicesNotifier`. The `HISLogic` is responsible for interacting with the `HIS` for the patient record. Since this `HISLogic` contains the main responsibilities of the `HISFacade`, the `HISLogic` is also pinged for availability. The `HISAvailabilityMonitor` is responsible for monitoring the availability of the `HIS` (and the communication channel to it) according to *Av2*. The `PatientRecordReadCache` provides a cache of patient records from the `HIS` for use in degraded modus (i.e., when the `HIS` cannot be reached). The entries of this cache are invalidated after 24 hours. If a patient record update arrives for which the patient record is in the cache, the cache entry is updated as well. The `PatientRecordUpdateBuffer` provides a buffer for temporarily storing patient record updates in degraded modus. Finally, the `EmergencyServicesNotifier` is responsible for notifying the emergency services. It is a separate sub-component in order to be able to deploy it separately such that it is not influenced by the load of the rest of the `HISFacade` and emergency notifications are always forwarded as quickly as possible.

Failure of the `HIS` (for *Av2*) is handled as such: The `HISAvailabilityMonitor` continuously checks the availability of the `HIS` by pinging it every 25 seconds. In case of failure, the `HISAvailabilityMonitor` notifies the system administrator, keeps track of how long this failure occurs and puts the `HISLogic` into degraded modus. In that modus, the `HISLogic` will respond to patient record requests from the `PatientRecordReadCache` and will store patient record updates in the `PatientRecordUpdateBuffer`. When the `HIS` becomes available again, the `HISLogic` flushes the update buffer in a first-in/first-out order. In normal modus, the `HISLogic` forwards patient record updates to the `HIS` immediately, serves read requests by directly contacting the `HIS` every time and puts the result of every read into the `PatientRecordReadCache` for use in degraded modus.

5.2 GatewayFacade

The decomposition of the `GatewayFacade` is presented in Figure 4. As shown, the `GatewayFacade` is decomposed into three sub-modules: the `GatewayLogic`, the `DeadlineChecker` and the `GatewayEmergenciesFacade`. The `GatewayLogic` is responsible for communicating with the gateway devices,

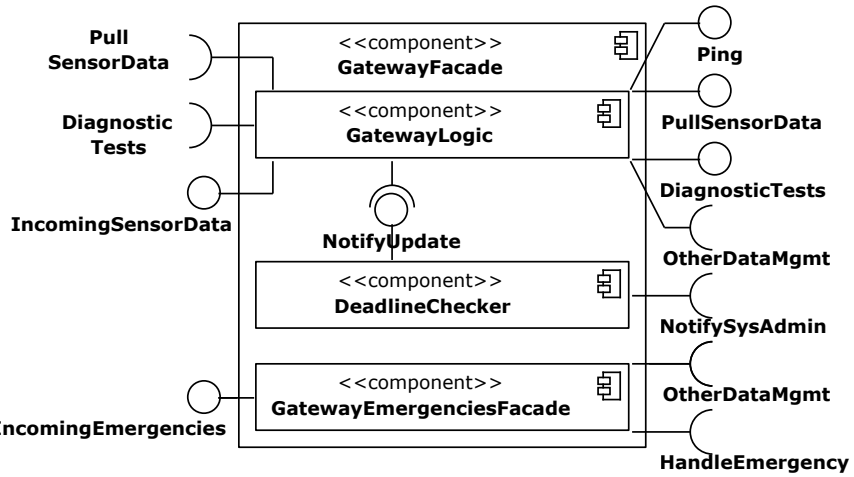

Figure 4: Decomposition of the `GatewayFacade`.

storing the new sensor data and launching risk level estimations. The `DeadlineChecker` is responsible for monitoring the transmission deadlines of the different gateways and notifying the system administrators in case a gateway fails to comply to its deadline. On each newly received sensor data package, the `GatewayLogic` notifies the `DeadlineChecker` and also passes the deadline of the next package as given by the gateway. Finally, the `GatewayEmergenciesFacade` is responsible for receiving and passing incoming emergency notifications. It is a separate sub-component in order to be able to deploy it separately such that it is not influenced by the load of the rest of the `GatewayFacade` and emergency notifications are always forwarded as quickly as possible.

5.3 RiskEstimator

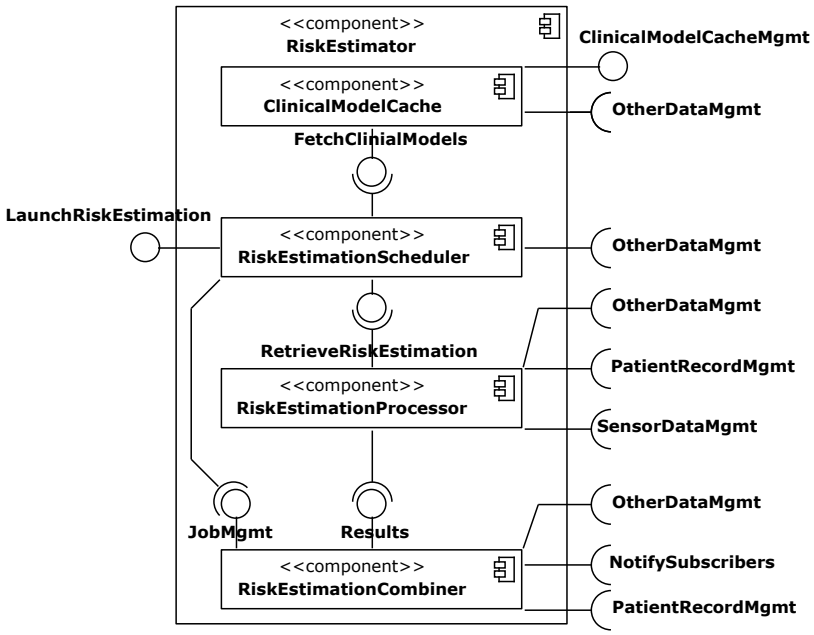

Figure 5: Decomposition of the `RiskEstimator`.

The decomposition of the `RiskEstimator` is presented in Figure 5. As shown, the `RiskEstimator` is decomposed into four sub-modules: the `ClinicalModelCache`, the `RiskEstimationScheduler`, the `RiskEstimationProcessor` and the `RiskEstimationCombiner`. The `RiskEstimationScheduler` ac-

cepts new requests for risk estimations (which are given as the patient id for which the risk level should be estimated with optionally a sensor data update and time-stamp of arrival if the risk estimation was triggered by the arrival of new sensor data). Afterwards, the `RiskEstimationScheduler` will fetch the assigned clinical risk models and their configurations from the `OtherDataDB` using the `ClinicalModelCache`. This `ClinicalModelCache` caches the clinical models assigned to the different patients and their specific configurations. Since the assignment of clinical models and their configurations can be presumed to change infrequently, this cache is expected to avoid remote traffic for the majority of requests to the `OtherDataDB` and speed up the process of evaluating the clinical models as such. Because the clinical models and their configuration are crucial to correct risk estimations, the cache has to be kept consistent. For this, the `ClinicalModelCache` can explicitly be invalidated. After fetching the assigned risk models, the `RiskEstimationScheduler` will create a job for each risk model to be evaluated, put these into a queue, monitor their completion time and notify the `RiskEstimationCombiner` of the different risk models that should be computed for that patient. The `RiskEstimationProcessor` (which can be deployed multiple times on different machines) asynchronously fetches these single clinical model computation jobs from the `RiskEstimationScheduler`, which returns the next job depending on the current modus. In normal modus, the `RiskEstimationScheduler` returns the jobs in a **first-in/first-out** manner. In overload modus (i.e., if the deadlines of *P1* cannot be reached), the `RiskEstimationScheduler` returns jobs using **dynamic priority scheduling** based on the deadlines stated in *P1*. During the computation of a clinical model, the `RiskEstimationProcessor` fetches all required data from the `SensorDataDB`, the `HIS` and the `OtherDataDB`. Afterwards, the `RiskEstimationProcessor` returns its results to the `RiskEstimationCombiner`. When the `RiskEstimationCombiner` has received the results of all required risk estimations for a certain patient, it combines these, notifies the subscribers of that patient if the risk level is estimated to change and propagates the new sensor data and the results of the risk estimation to the patient record if needed.

5.4 SensorDataDB

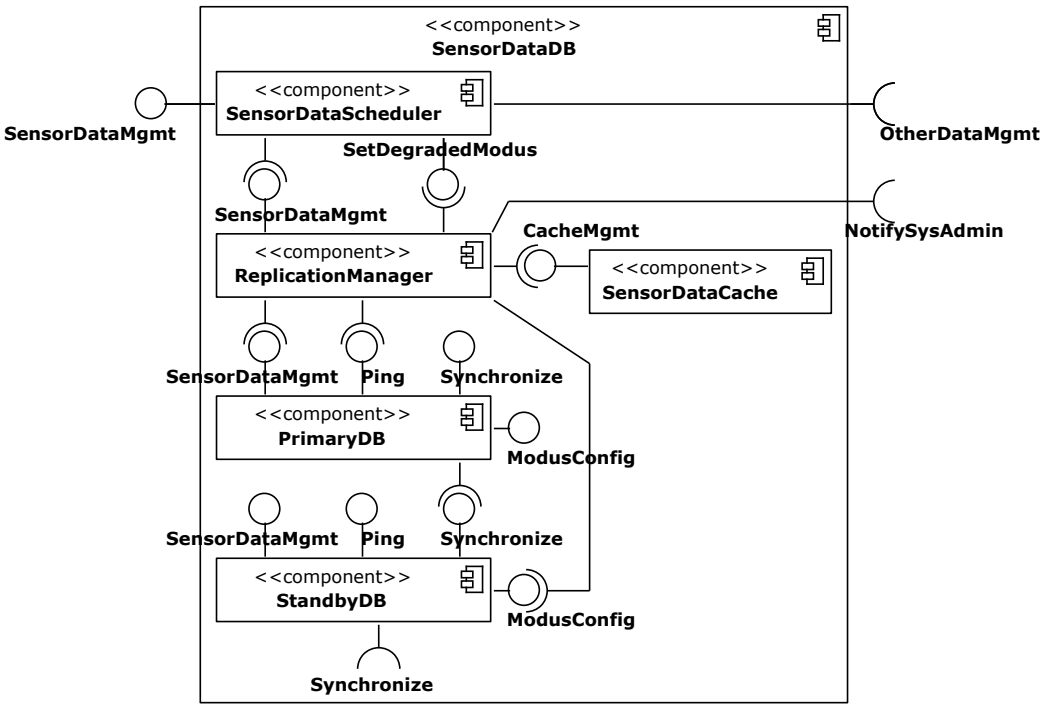

Figure 6: Decomposition of the `SensorDataDB`. Dangling internal interfaces represent available interfaces which are not connected at all times.

The decomposition of the `SensorDataDB` is presented in Figure 6. As shown, the `SensorDataDB` is decomposed into five sub-modules: the `SensorDataScheduler`, the `ReplicationManager`, the `PrimaryDB`,

the StandbyDB and the SensorDataCache. The SensorDataScheduler is responsible for taking in all requests (reads and writes), queuing them, monitoring their completion time and scheduling them according to the current modus of the database in terms of availability and performance. The PrimaryDB is responsible for processing all reads and writes. The StandbyDB represents the back-up as required by *P1* and the PrimaryDB periodically pushes its new sensor data in order to synchronize the StandbyDB its state. The SensorDataCache is responsible for temporarily storing new sensor data written to the PrimaryDB, in order to avoid losing the data written to it after the last synchronization with the StandbyDB. This cache should therefore be able to store sensor data for at least the synchronization period of the StandbyDB. The ReplicationManager is responsible for managing the whole replication scheme. Firstly, it passes read requests to the PrimaryDB and passes write requests to both the PrimaryDB and the SensorDataCache. The ReplicationManager also checks the availability of the PrimaryDB by checking the results of all requests sent to it and pinging it if necessary, so as to check it at least every 4 seconds. In case of failure, the ReplicationManager notifies a system administrator, changes the StandbyDB into the PrimaryDB, flushes the SensorDataCache to the new PrimaryDB in order to synchronize it and starts using it for reads and writes. Note that the SensorDataCache can contain sensor data which the PrimaryDB already pushed to the StandbyDB. However, every sensor data package is identified by a combination of patient identifier, gateway identifier and the time-stamp at which the GatewayFacade received the data. Because the SensorDataCache also stores these, repeating the write will overwrite the existing, but identical sensor data in the new PrimaryDB, ensuring correctness.

The SensorDataScheduler schedules requests according to the current modus of the database. In normal modus, the SensorDataScheduler passes on the requests earliest-deadline-first according to the deadlines stated in *P1*. In degraded modus (i.e., when the primary has failed according to *Av3*), the SensorDataScheduler gives sensor data readings of patients with red risk level priority over others (thereby allowing starvation). In both degraded modus and overload modus (i.e., in case the database cannot comply to the deadlines stated in *P1*), new sensor data packages of patients with green risk levels can be omitted, but not more than twice consecutively in total. In order to guarantee not to omit more sensor data packages, the scheduling for *P1* and *Av3* is combined into a single component, i.e., the SensorDataScheduler.

Notice that the SensorDataScheduler reasons about both read and write requests. This is necessary because the read requests sent to the SensorDataScheduler influence the performance of the writes, which are subject to strict performance requirements according to *P1*. To incorporate the reads and at the same time avoid starvation, reads are given longer deadlines than writes for patients with green risk levels. Similarly, in overload modus, they are handled at lower priority than writes for patients of red risk levels. By relating these deadlines to the sources of the read requests, the impact of these different sources can also be managed. For example, it can be expected that in certain situations (e.g., an epidemic), patients could be checking their status much more frequently than usual. In order to avoid that the OtherUsersFacade dominates the performance of the SensorDataDB, requests from this source can be given longer deadlines and lower priorities than others.

6 Deployment view

The context diagram of the deployment view is given in Figure 7. As shown, four nodes are connected to the outside world: the EmergenciesNode, the GatewayFacadeNode, the HISFacadeNode and the UserFacadesNode. Notice that there will be multiple (if not a large number) of GatewayNodes, HospitalUsersTerminals and OtherUsersTerminals communicating with the PMS simultaneously.

The entire deployment diagram of the PMS is presented in Figure 8. As shown, the components are deployed in three tiers, similarly as they were decomposed in Figure 2: a tier of front-end nodes hosting facades and supporting components, a tier of business nodes hosting business logic such as clinical model processing, and a tier of databases. Notice that the deployment diagram employs both top-level components as sub-components of further decompositions. If a top-level component is deployed on a node, this means that all its sub-components are deployed there as well. Also notice that for readability reasons, we only show multiplicities differing from 1. Finally, also notice that we did not specify protocols for internal connections.

In general, the deployment of the components in the different tiers is chosen in order to separate components with differing non-functional requirements, but group components as much as possible in order to minimize the total number of required nodes. The front-end tier contains the EmergenciesNode, the GatewayFacadeNode, the HISFacadeNode and the UserFacadesNode. The EmergenciesNode hosts the components responsible for incoming and outgoing communication about emergencies: the GatewayEmergenciesFacade takes in the emergency notifications sent by the gateways and the EmergencyServicesNotifier sends out emergency notifications to the emergency services. These components are deployed separately from the rest in order to avoid their load influencing the performance of the processing of emergencies. The GatewayFacadeNode hosts the GatewayLogic (which is responsible for all the communication with the gateways, except from the emergency notifications) and the supporting DeadlineChecker. The HISFacadeNode hosts the HISLogic and all components supporting this, plus the NotificationHandler. The UserFacadesNode hosts all facades for human end-users, plus the AuthNHandler and SessionDB since these are thightly coupled to the facades. The components communicating with the HIS, the gateways and the end-users are separated from each other in order to avoid their load influencing each others performance. Each of these nodes can be replicated if needed, e.g., in case the single GatewayFacadeNode is not be able to support the number of gateways.

The business tier contains the EmergencyHandlerNode, the RiskEstimationMgmtNode, one or multiple RiskEstimationProcessorNodes and the PatientStatusOverviewNode. The EmergencyHandlerNode hosts the EmergencyHandler. This component is deployed separately in order to guarantee that emergencies are handled as fast as possible by decoupling its performance from the rest of the system. The RiskEstimationMgmtNode hosts the components involved in managing risk estimations and the RiskEstimationProcessorNodes each host multiple RiskEstimationProcessors. By replicating the RiskEstimationProcessor on multiple nodes, the system can scale out in order to achieve the hard deadline of 10 minutes for any risk level estimation job in the presence of a growing number of jobs per minute (as required by *P2*). By also replicating the RiskEstimationProcessor on each single RiskEstimationProcessorNode, multiple risk estimations can run concurrently, allowing each node to be fully occupied (e.g., when a certain estimation makes a database call, the RiskEstimationProcessorNode can switch to processing another in order to avoid unnecessary waiting time). Finally, the PatientStatusOverviewNode hosts the rest of the business logic, which is mainly the PatientStatusOverviewHandler.

The database tier contains the SensorDataDBMgmtNode, the PrimaryNode, the StandbyNode and the OtherDBsNode. The SensorDataDBMgmtNode hosts the components involved in managing the PrimaryDB and the StandbyDB. The PrimaryNode and StandbyNode respectively host the PrimaryDB and StandbyDB. Finally, the OtherDBsNode hosts all other database components, which are presumed to be small enough to be combined onto a single node.

A property of the final deployment diagram is the large number of connections between the nodes. This does not benefit the readability of this diagram. However, the large number of connections is mainly caused by (i) the large number of connections to the OtherDBsNode, which hosts all data except from the sensor data, and (ii) the large number of connections to the HISFacadeNode, which hosts the communication to the HIS and the NotificationHandler, which are used by a large number of other components. As such, we believe that the large number of connections results from valid architectural decisions.

Non-functional requirements

The deployment shown in Figure 8 is crucial for achieving some of the non-functional requirements. More specifically:

1. For *Av1* it is important that the `ComponentAvailabilityMonitor` is deployed on another node as the components which it monitors so that those can fail independently from the `ComponentAvailabilityMonitor`. As such, the `ComponentAvailabilityMonitor` is deployed on another node as the `GatewayLogic` and the `HISLogic`. For more details, refer to Section 4.1.1.

2. For *Av3*, it is important that the `SensorDataDB` (or more precisely, all of its sub-components) is deployed on another node as the other databases in order to allow it to fail independently of the other types of persistent data. For more details, refer to Section 4.1.2.

3. For *Av3*, it is important that the `ReplicationManager`, the `PrimaryDB` and the `StandbyDB` are each deployed on separate nodes, so that the `PrimaryDB` and `StandbyDB` can fail without bringing the whole `SensorDataDB` down. For more details, refer to Section 4.1.2.

4. For *P2*, it is important that the `RiskEstimationProcessor` is replicated on multiple nodes (as explained above). In essence, it is not required that the management components (i.e., the `RiskEstimationScheduler` and the `RiskEstimationCombiner`) are located on a separate node as the `RiskEstimationProcessor`s. However, we opted for this in order to make the multiple `RiskEstimationProcessorNode`s identical, thereby simplifying the architecture. For more details, refer to Section 4.1.4.

5. For performance reasons, it is important that the `ClinicalModelCache` is deployed on the same node as the `RiskEstimationScheduler`, thereby avoiding inter-node communication.

Alternatives considered

Alternatives for the bundling of components. As mentioned above, the deployment of the components in the different tiers is chosen in order to separate components with differing non-functional requirements, but group components as much as possible in order to minimize the total number of required nodes. A first (extreme) alternative is to actually deploy all components which can be deployed on the same node based on the non-functional requirements on a single node. This would minimize the deployment cost while still supporting the main requirements. However, the load of the different sub-systems of the PMS would heavily influence each other's performance. A second (extreme) alternative is to deploy each component on a separate node. While this deployment would clearly decouple the loads of all sub-systems, it would clearly lead to high hardware costs, while under-utilizing each individual node and leading to large latencies for some operations because of the need for cross-node communication. We believe that our deployment strikes the balance between both extremes.

Alternatives for the different tiers. An alternative to the use of the different physical tiers in the deployment diagram is grouping components of different logical tiers on a single physical node. The advantage of the different physical tiers is the ability to specialize the physical nodes for their individual roles, e.g., the database nodes require extensive persistent storage, while the clinical model processors will focus more on processing power. In case that nodes in the infrastructure of the PMS are all similar (for example, large data centers are known to use a small number of commodity hardware configurations), mixing different logical tiers on a single physical node has the advantage of optimally utilizing the different hardware resources of each node, i.e., each node will host some processing-oriented components and some storage-oriented components.

Alternatives for the `EmerencyHandlerNode`. As shown, the `EmergencyHandler` is deployed on a separate node, mainly in order to guarantee that emergencies are handled as fast as possible by decoupling its performance from the rest of the system. As a result, the `EmergencyHandler` is now also decoupled from the `GatewayEmerenciesFacade` and `EmergencyServicesNotifier`. This decouples the verification of emergencies from receiving new emergency notifications and the sending of notifications to the emergency services. However, this decoupling also leads to two network hops in this verification,

increasing its latency. If this latency should be avoided, the `EmergencyHandler` can be deployed on the `EmergenciesNode` as well.

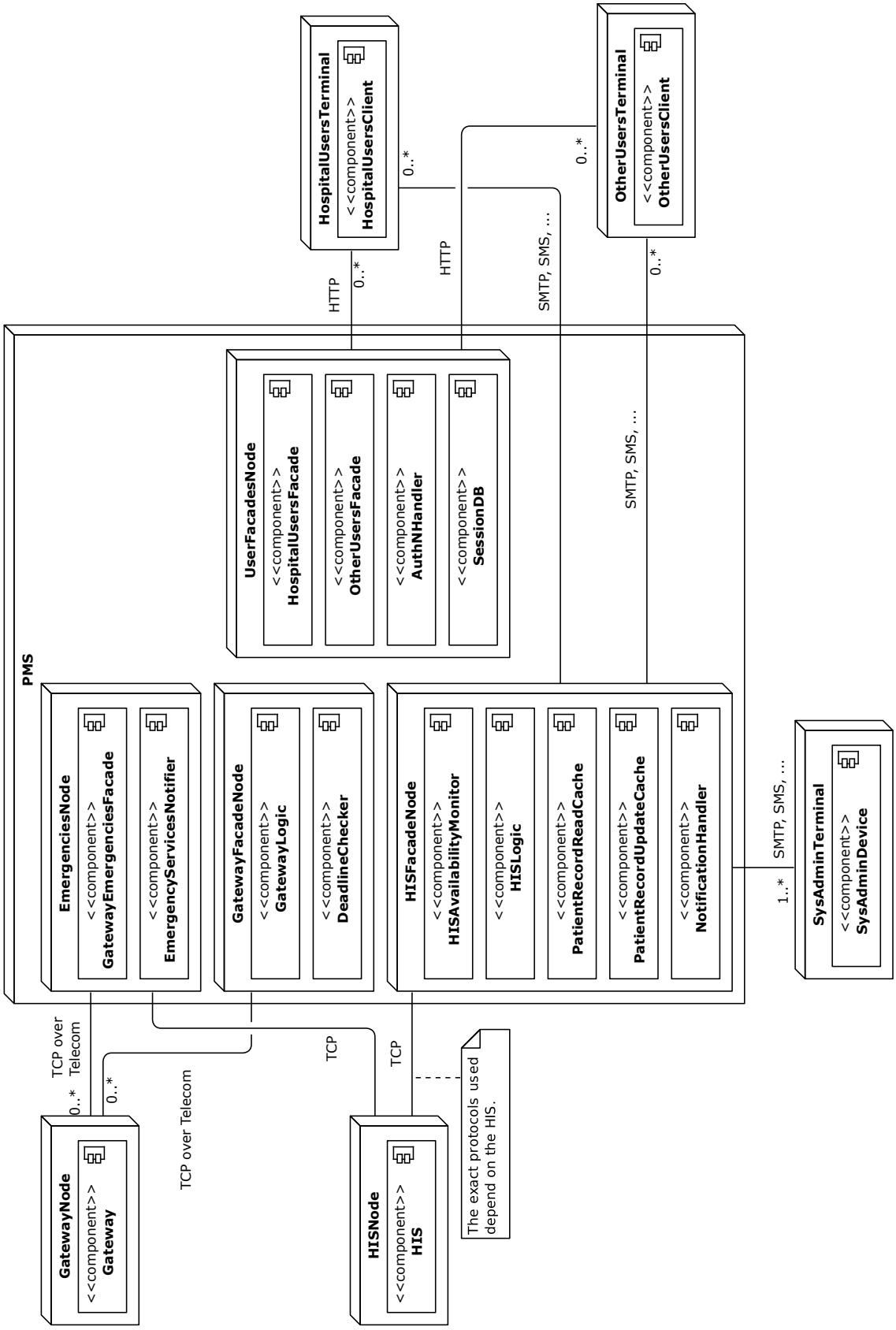

Figure 7: Context diagram of the deployment view.

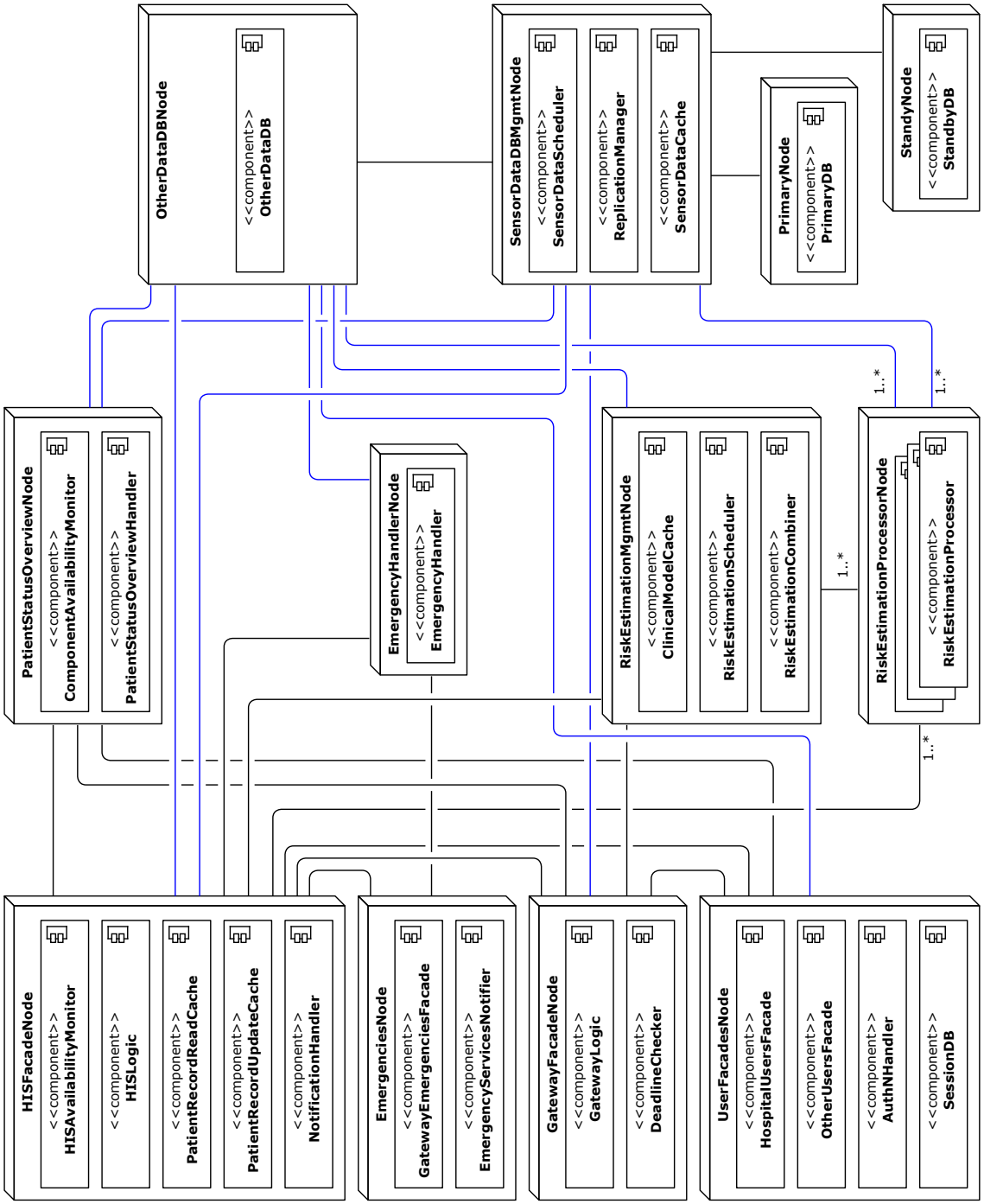

Figure 8: Deployment diagram. For readability reasons, we only show multiplicities differing from 1.

7 Scenarios

In this section, we illustrate the internal behavior of the system for the most important data flows, i.e., the processing of new sensor data, the processing of emergency notifications, an on-demand patient status consultation and a normal patient status consultation. Note that these diagrams employ the most fine-grained components instead of their super-components for clarity. Notice the difference between synchronous and asynchronous messages.

7.1 Sensor data arrival

Figure 9 shows the behavior of the system when a sensor data update arrives. As shown, the `GatewayLogic` accepts the update, updates the deadline for the corresponding gateway and forwards the newly arrived sensor data to the `SensorDataScheduler` for storage. The actual storage is detailed in Figure 10. Finally, the sensor data is send to the `RiskEstimationScheduler` to execute a risk estimation, detailed in Figure 11.

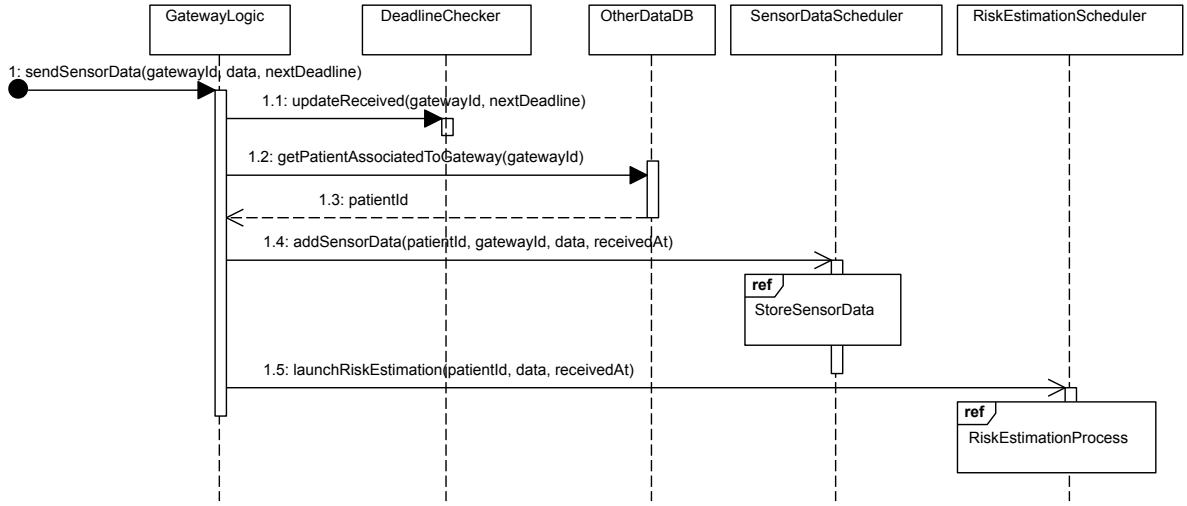

Figure 9: The behaviour of the system when a sensor data update arrives.

Figure 10 shows how sensor data is processed by the `SensorDataDB`. Note that the confirmation is send to the caller after scheduling or omitting the sensor data. The confirmation thus does not indicate that the sensor data has actually been written to the database.

Figure 11 shows how the risk estimation following the arrival of new sensor data is scheduled and executed. First, the patient status and all clinical models and their configurations for the concerned patient are retrieved. Note that if one or more clinical models in the `ClinicalModelCache` have been invalidated, all relevant clinical models are renewed through the `OtherDataDB`. For each clinical model a job is scheduled in the queue of the `RiskEstimationScheduler`, after which the `RiskEstimationCombiner` is notified of this new set of jobs to be combined when they are completed. The combined result is written to the `OtherDataDB` as latest estimation for the concerned patient. Based on this result the subscribers are notified (detailed in Figure 13) and/or the patient record is updated (detailed in Figure 14).

Figure 12 shows the execution of a single clinical model job for a certain patient. The `RiskEstimationProcessor` starts with retrieving the next job from the scheduler. Then it retrieves the data required for the computation of the clinical model at hand. Note that the required data can differ depending on the clinical model in question. After computation the result is send to the `RiskEstimationCombiner` and the `RiskEstimationProcessor` retrieves its next job.

Figure 13 shows how subscribers are notified as a consequence of risk estimation.

Figure 14 shows how the results of a risk estimation, if needed, are sent to the patient. Important here is that the conversion from `patientId`, used internally by the PMS, to the `hisPatientId`, used by the `HIS`, is done through the `OtherDataDB`. Furthermore, depending on the availability of the `HIS` the

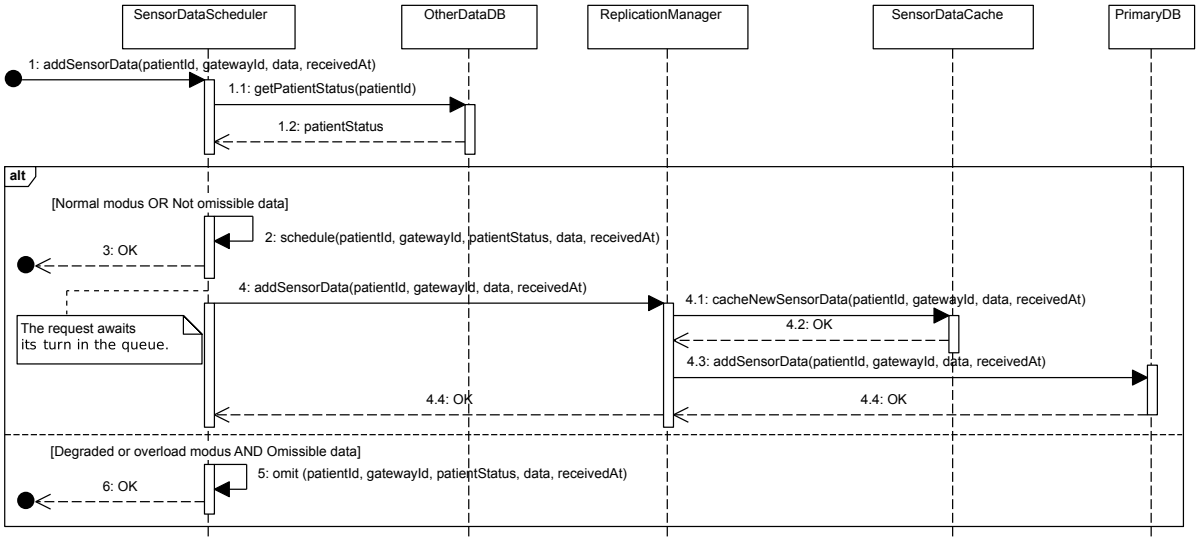

Figure 10: StoreSensorData: The storage of new sensor data in the `SensorDataDB`.

result is send to the `HIS` immediately or stored in the `PatientRecordBuffer` and send later (not shown here).

Figure 15 shows how sensor data is retrieved from the `SensorDataDB`.

Figure 16 shows how the patient record is retrieved from the `HIS`. If the `HIS` is unavailable a cached, possibly outdated, version of the patient record is returned.

7.2 Emergency notification

Emergency notifications are handled mostly by components separate from the main functionality. Figure 17 shows how an incoming emergency notification is processed. After retrieving the corresponding patient identifier the emergency sensor data is forwarded to the `EmergencyHandler` for verification. If the emergency is confirmed, the emergency services at the hospital are notified with sufficient details about the concerned patient using the `HIS`. Furthermore the emergency is written to the patient record at the `HIS`. If the emergency notification turns out to be benign it is only written in the patient record as such.

Figure 18 shows the behaviour concerning writing to the patient record. Important here to note is that depending on the availability of the `HIS` the update is written directly to the patient record or first stored in a buffer and forwarded when the `HIS` becomes available again (not shown here).

7.3 On-demand patient consultation

Figure 19 shows the general flow when a cardiologist requests to perform an on-demand consultation of a patient. First, the system verifies whether the session of the user is valid (detailed in Figure 20), as performed for each request initiated by a human user. Subsequently the patient's gateway is contacted through the `GatewayLogic` to send its current sensor data. This sensor data is processed in the same manner as periodic sensor data updates. Important in this flow is that the `HospitalUsersFacade` will actively poll the `OtherDataDB` to know when the risk estimation, initiated by pulling the sensor data, is completed. This is done by checking the time-stamp of the last estimated patient status in the database, which will be updated on completion of the risk estimation. When the risk estimation is complete the system will construct a patient status overview and present this to the cardiologist.

Figure 20 shows how it is verified whether a given session is valid. The `AuthNHandler` verifies through the `SessionDB` whether each incoming session identifier belongs to an existing session. If so the corresponding session attributes are returned to the caller, otherwise an error is returned.

Figure 21 shows how a patient status overview in generated. Important to note is that this overview is tailored to the user requesting it. In this case, the overview will contain detailed medical information

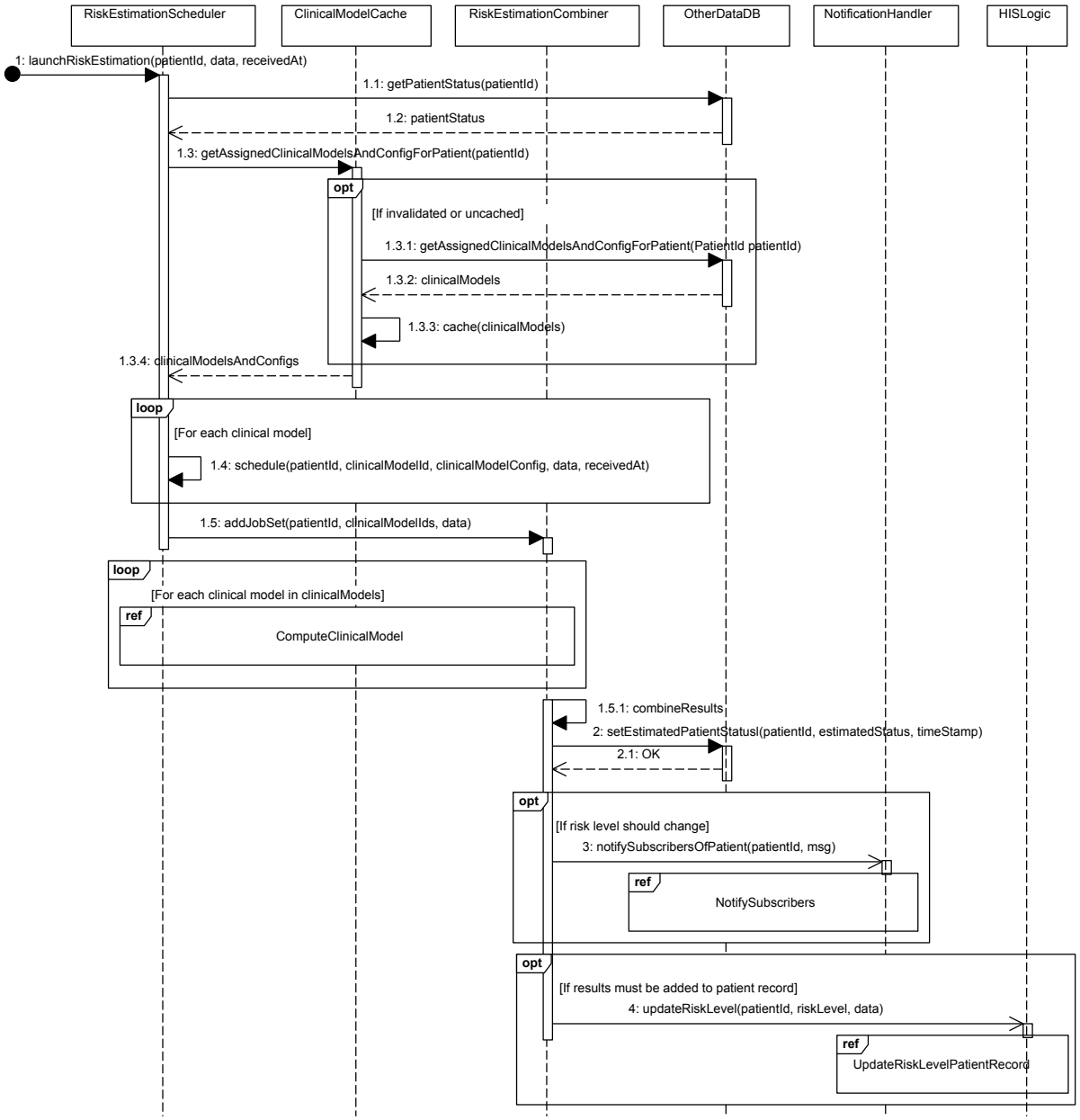

Figure 11: RiskEstimationProcess: The scheduling and execution of a risk estimation triggered by the arrival of new sensor data.

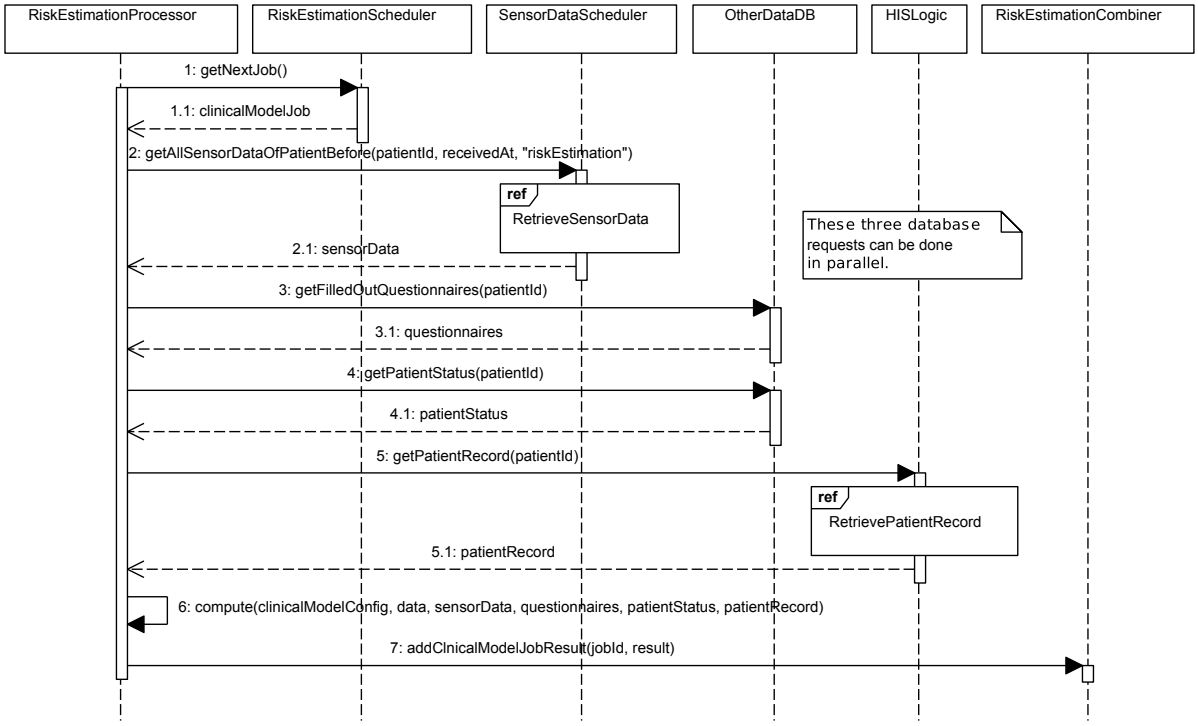

Figure 12: ComputeClinicalModel: The computation of a single clinical model.

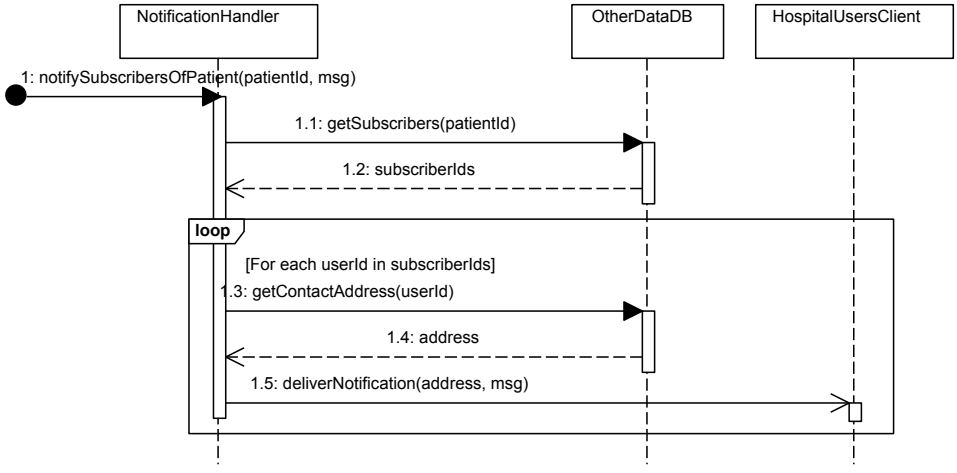

Figure 13: NotifySubscribers: Notifying subscribers for a patient of important changes concerning the patient.

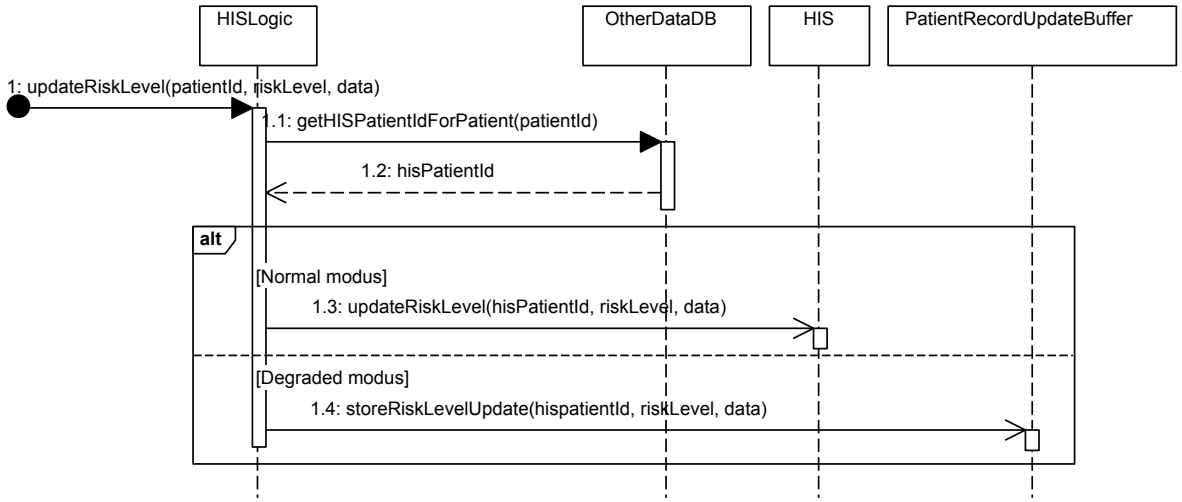

Figure 14: UpdateRiskLevelPatientRecord: Update the risk level in the patient record at the HIS.

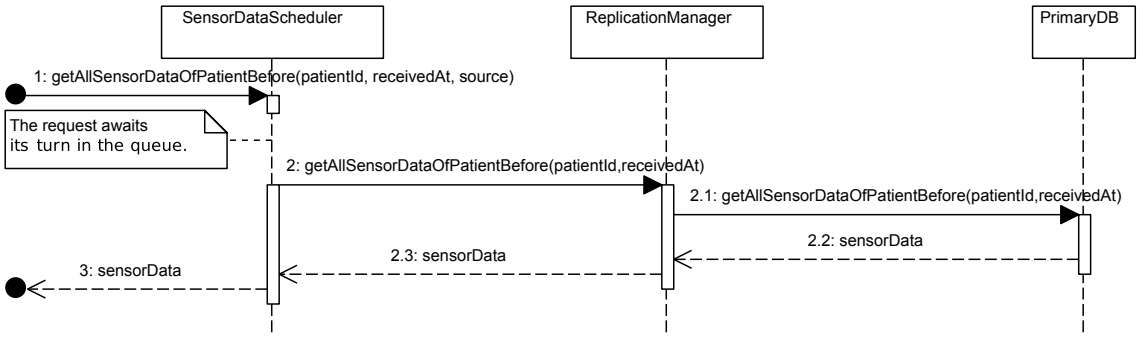

Figure 15: RetrieveSensorData: Retrieve sensor data from the `SensorDataDB`.

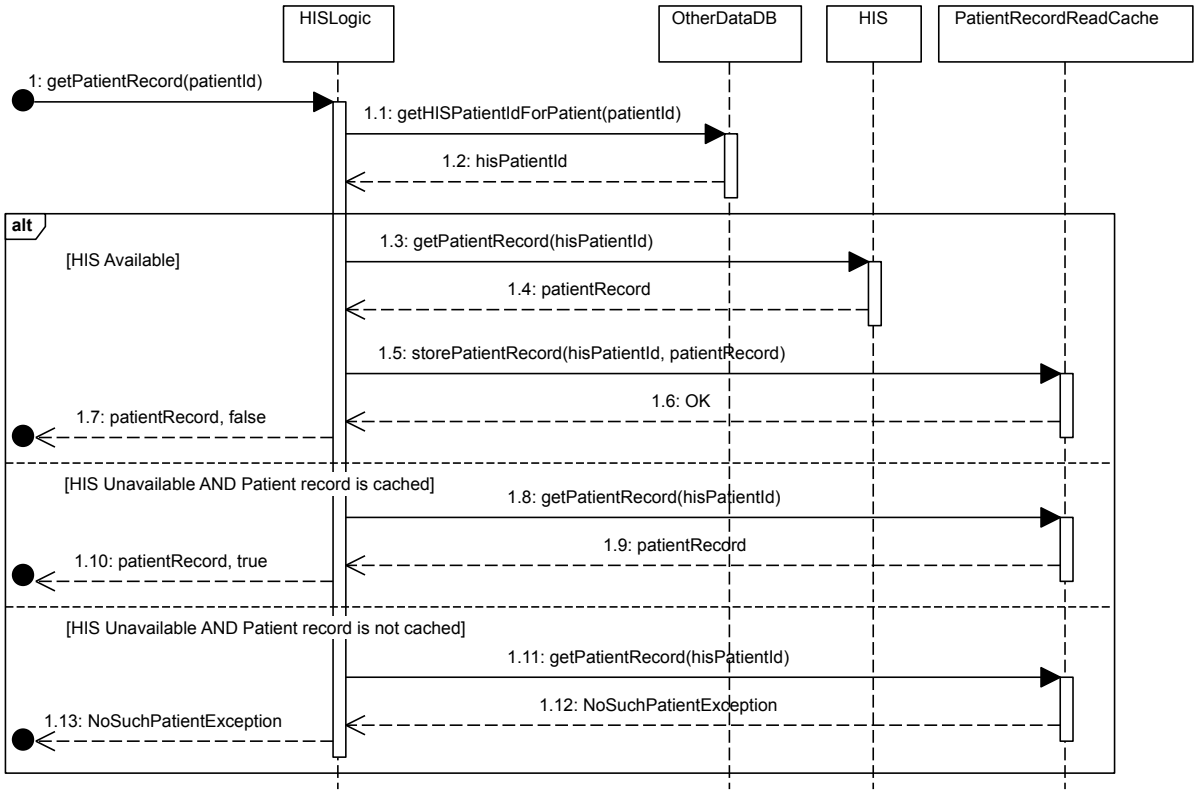

Figure 16: RetrievePatientRecord: Retrieve the patient record from the HIS.

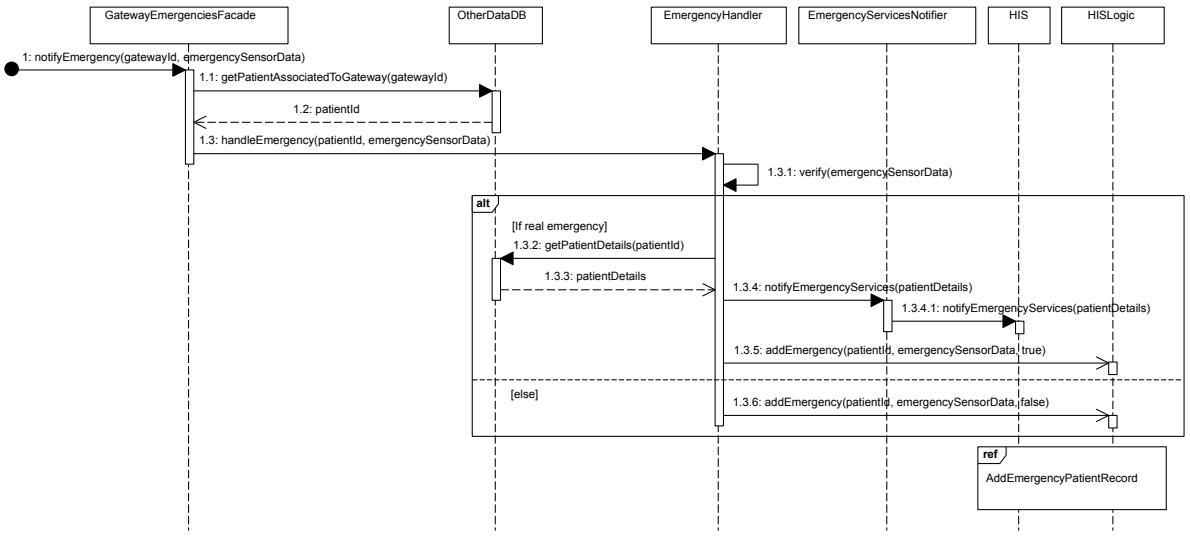

Figure 17: The internal behaviour when an emergency notification arrives at the PMS.

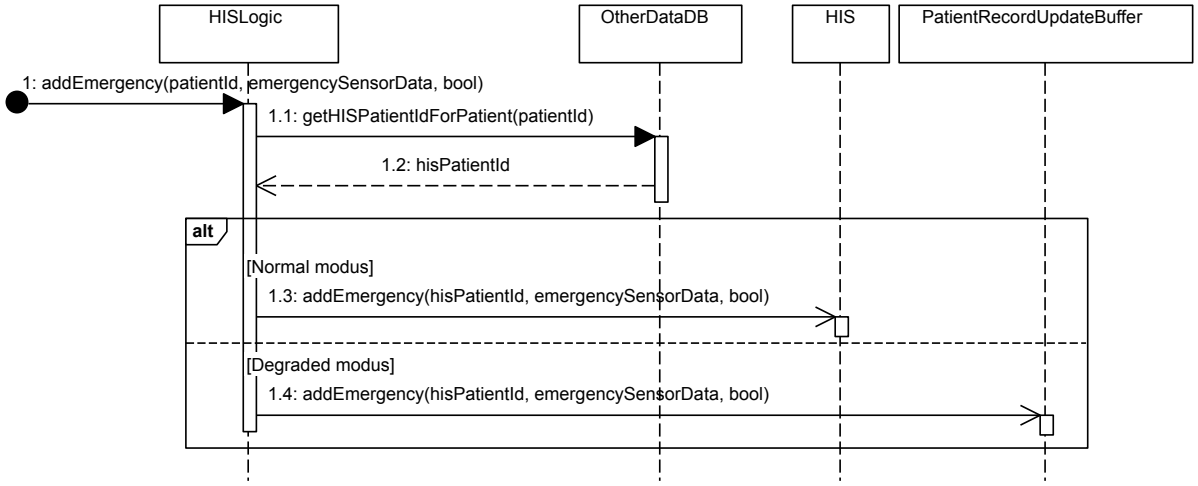

Figure 18: AddEmergencyPatientRecord: Add the notified emergency to the patient record at the HIS.

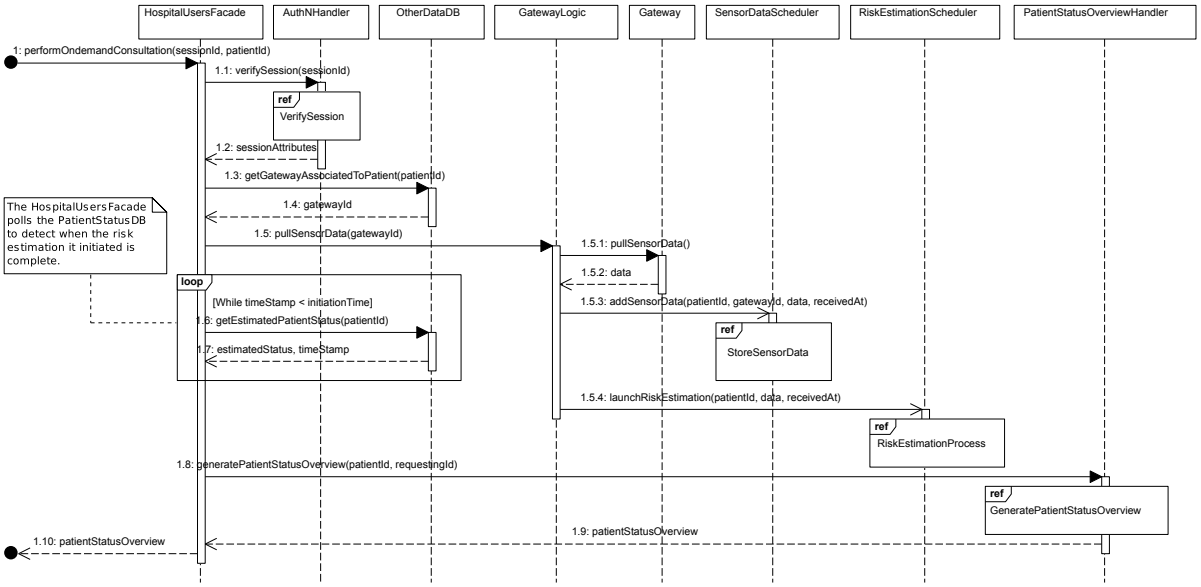

Figure 19: The internal behaviour when performing an on-demand patient consultation.

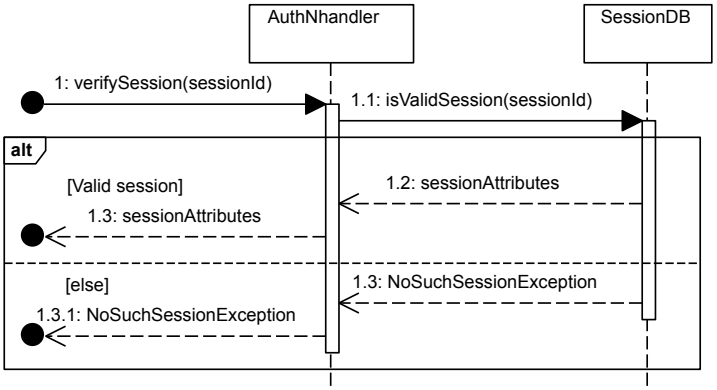

Figure 20: VerifySession: Verification whether a session is valid.

because it is requested by a cardiologist as part of an on-demand consultation. The role of each user is retrieved from the `OtherDataDB`.

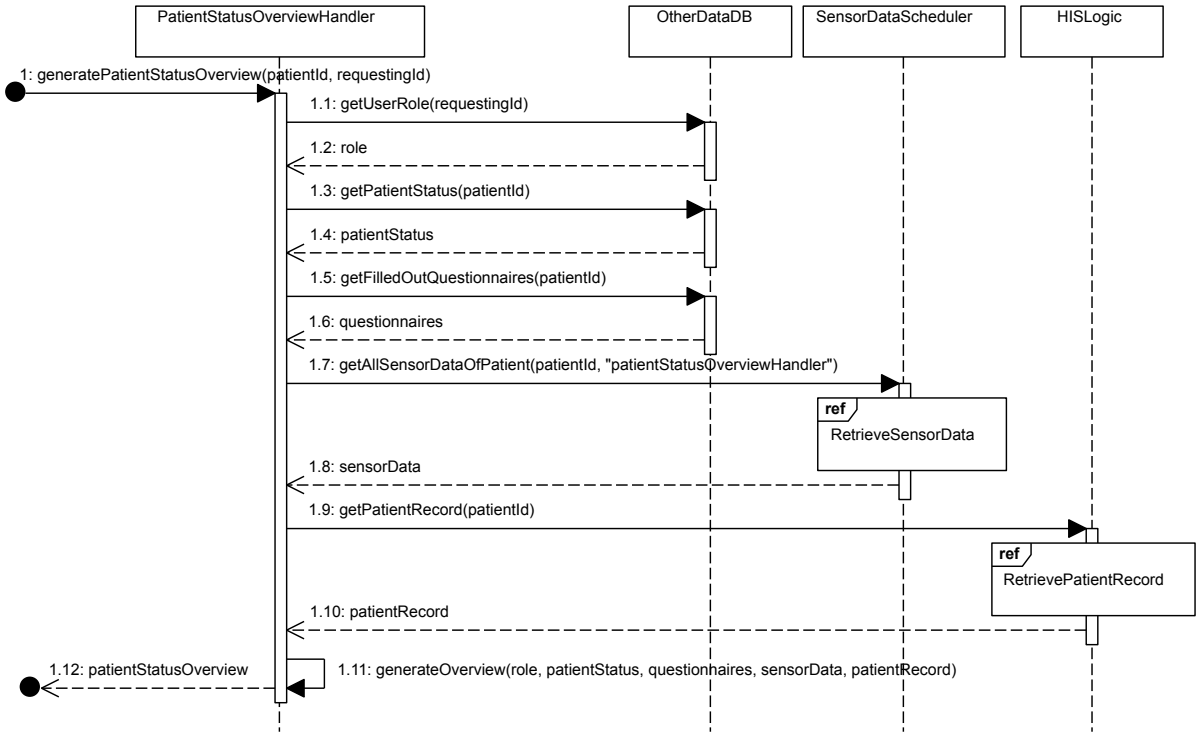

Figure 21: GeneratePatientStatusOverview: The generation of a patient status overview.

7.4 Consult patient status

Figure 22 shows the flow when a non-medical user (e.g. the patient himself) requests to consult a patient his status. First, the user must log in (detailed in Figure 23). As discussed earlier the session of the user is verified when a request from this user arrives. The patient status overview is generated by the `PatientOverviewHandler` as discussed above. Finally, the event is added to the log in `OtherDataDB`.

Figure 23 shows the login procedure enforced by the system. The user provides his or her credentials which are compared to those stored in the `OtherDataDB`. If the given credentials match the stored credentials a new session is opened and the corresponding session identifier is returned to the original caller.

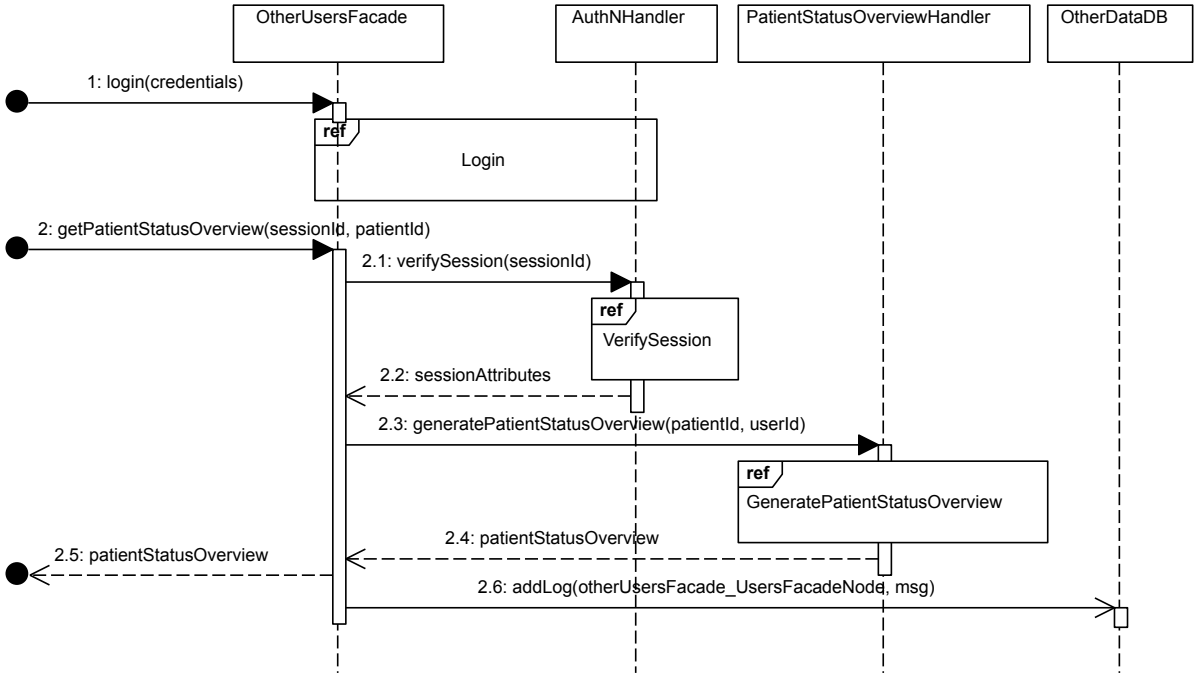

Figure 22: Consulting a patient's current status.

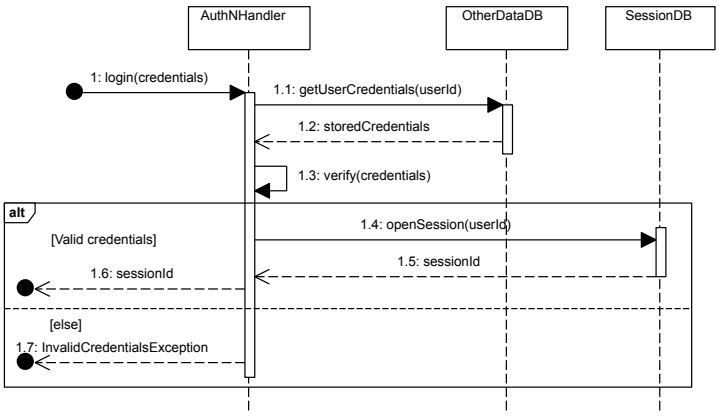

Figure 23: Login: The login behaviour of the PMS.

A Element catalog

In this section, we list all the components and the interfaces they provide. Per component, we clearly describe its responsibilities, declare its super-component (if any) and list its sub-components (if any). Per interface, we list its methods by means of its syntax. Per method, we list its effect (mostly in terms of subsequent actions) and possible exceptions. For the syntax of a method, we employ a Java-like notation. Since most data types are used by multiple methods, we give the details of these data types in a separate list in Section B and only refer to them in the interface specification.

Note that exceptions are only thrown by methods where the failing is important for the system and we do not throw exceptions for inconsistencies in the data model (e.g., no exception is thrown if a gateway is assigned to a non-existing user).

A.1 AuthNHandler

- **Description:** The `AuthNHandler` is responsible for authenticating end-users. The means of authentication (i.e., the type of credentials) depends on the type of user (e.g., patients will use a username and password, but physicians can use their hospital badges). The credentials of users are stored in the `OtherDataDB`.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- AuthN
  - `UserId getUserId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `AuthNHandler` will fetch and return the user identifier corresponding to the session identified by `sessionId` from the `SessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier.
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `AuthNHandler` will verify the credentials using the `OtherDataDB`. If correct, the `AuthNHandler` will create a new session using the `SessionDB`, store the id of the user as an attribute in this session (the id is present in the given credentials) and return the id of the new session.
    - Exceptions:
      - IncorrectCredentialsException: Thrown if the given credentials are invalid.
  - `SessionId login(UserId userId)`
    - Effect: The `AuthNHandler` will create a new session using the `SessionDB`, store the given user identifier `userId` as an attribute in this session and returns a new session identifier. This method is to be used by internal components in situations where no credentials have or can be verified, such as a new trustee confirming an invitation.
    - Exceptions: None
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `AuthNHandler` will remove the session with given id from the `SessionDB`. If no such session exists, nothing is changed and no exception is thrown.
    - Exceptions: None
- CheckSession
  - `Map<SessionAttributeKey, SessionAttributeValue> verifySession(SessionId sessionId) throws NoSuchSessionException`

    * Effect: The `AuthNHandler` will verify whether a session with the given id exists in the `SessionDB` and if so, will return all its associated attributes.
    * Exceptions:
        · NoSuchSessionException: Thrown if no session with given id exists.

A.2 ClinicalModelCache

- **Description:** The `ClinicalModelCache` is responsible for caching the list of clinical models that should be evaluated for a certain patient and their configurations. This cache is located close to the `RiskEstimationProcessor` and `RiskEstimationCombiner` in order to improve the latency of the whole risk estimation flow. The items in the `ClinicalModelCache` do not expire over time, but should be invalided explicitly if needed.
- **Super-component:** `RiskEstimator`
- **Sub-components:** None

Provided interfaces

- ClinicalModelCacheMgmt
    - `void invalidateCacheEntries(PatientId patientId)`
        * Effect: The `ClinicalModelCache` will invalidate (i.e., remove) all items in its cache for the patient identified by `patientId`. If the cache does not contain any items for this patient, nothing is changed. After invalidating the cached items for a certain patient, the next request for them will lead to fetching them from the `OtherDataDB` and storing them in the cache again.
        * Exceptions: None
- FetchClinicalModels
    - `Map<ClinicalModelId, Map<ClinicalModelParameterKey, ClinicalModelParameterValue>> getAssignedClinicalModelsAndConfigForPatient(PatientId patientId)`
        * Effect: The `ClinicalModelCache` will return the ids of the clinical models assigned to the patient identified by `patientId` and their configurations for this patient. The configurations are returned as a map of configuration parameters and their value. If the cache contains data for the patient with given id, the data from the cache is returned. If not, the `ClinicalModelCache` will fetch all clinical models assigned to the patient with given `patientId` and their configurations for this patient, store these in the cache and return them.
        * Exceptions: None

A.3 ComponentAvailabilityMonitor

- **Description:** The `ComponentAvailabilityMonitor` is responsible for checking the availability of other availability-sensitive components (e.g., the `GatewayFacade`) using the Ping interface. If one of these components has failed, the `ComponentAvailabilityMonitor` notifies the System Administrator using the `NotificationHandler`.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

None

A.4 DeadlineChecker

- **Description:** The `DeadlineChecker` is responsible for checking the transmission deadlines of the different gateways and notifying the PMS system administrators in case a gateway fails to comply to its deadline.
- **Super-component:** `GatewayFacade`
- **Sub-components:** None

Provided interfaces

- NotifyUpdate
  - `void updateReceived(GatewayId gatewayId, TimeStamp nextDeadline)`
    - Effect: The `DeadlineChecker` will stop waiting for the previous deadline for the gateway identified by `gatewayId` and will start waiting for the next deadline for this gateway. If no update arrives before this deadline, The `DeadlineChecker` will notify the PMS system administrators.
    - Exceptions: None

A.5 EmergencyHandler

- **Description:** The `EmergencyHandler` is responsible for verifying an emergency notification sent by the `Gateway` by processing emergency clinical models. In case the emergency is verified, the `EmergencyHandler` notifies the emergency services through the `HISFacade`. The `EmergencyHandler` stores the event, the sensor data which triggered it and whether or not is was a real emergency in the patient's EHR, also using the `HISFacade`. As required by *P3: Emergency notifications*, the employed emergency clinical models are light-weight and do not rely on the patient's data (as opposed to the heavy-weight clinical models used in the `RiskEstimator`).
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- HandleEmergency
  - `void handleEmergency(PatientId patientId, SensorDataPackage emergencySensorData)`
    - Effect: The `EmergencyHandler` will verify the emergency using the given sensor data. If the emergency can be verified, the `EmergencyHandler` will notify the emergency services using the `HISFacade`. The event together with the triggering `emergencySensorData` and the result of the verification is stored in the patient's EHR using the `HISFacade`.
    - Exceptions: None

A.6 EmergencyServicesNotifier

- **Description:** The `EmergencyServicesNotifier` is responsible for notifying the emergency services. It is a separate sub-component in order to be able to deploy it separately such that it is not influenced by the load of the rest of the `HISFacade` and emergency notifications are always forwarded as quickly as possible.
- **Super-component:** `HISFacade`
- **Sub-components:** None

Provided interfaces

- NotifyEmergencyServices
  - `void notifyEmergencyServices(PatientDetails patientDetails, optional NotificationMessage msg)`
    - Effect: The `EmergencyServicesNotifier` will deliver the emergency notification (consisting of the details of the patient experiencing the emergency and an optional message) to the Emergency Services, which will handle the emergency situation.
    - Exceptions: None

A.7 Gateway

- **Description:** The `Gateway` is external to the PMS and represents the gateway device that communicates with the PMS.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- PullSensorData
  - `SensorDataPackage pullSensorData()`
    - Effect: The `Gateway` will request the latest measurements of all its connected sensors (e.g., the Wearable Unit), will package these and return them to the caller.
    - Exceptions: None
- DiagnosticTests
  - `DiagnosticTestResults performDiagnosticTests()`
    - Effect: The `Gateway` will perform local diagnostic tests and return their results.
    - Exceptions: None

A.8 GatewayEmergenciesFacade

- **Description:** The `GatewayEmergenciesFacade` is responsible for receiving and passing incoming emergency notifications. It is a separate sub-component in order to be able to deploy it separately such that it is not influenced by the load of the rest of the `GatewayFacade` and emergency notifications are always forwarded as quickly as possible.
- **Super-component:** `GatewayFacade`
- **Sub-components:** None

Provided interfaces

- IncomingEmergencies
  - `void notifyEmergency(GatewayId gatewayId, SensorDataPackage emergencySensorData)`
    - Effect: The `GatewayEmergenciesFacade` will look up the patient corresponding to the gateway identified by `gatewayId` and forward the emergency notification (i.e., the patient identifier and the given emergency sensor data) to the `EmergencyHandler`.
    - Exceptions: None

A.9 GatewayFacade

- **Description:** The `GatewayFacade` provides the main interface of the system to the `Gateway`.
- **Super-component:** None
- **Sub-components:** `DeadlineChecker`, `GatewayEmergenciesFacade` and `GatewayLogic`

Provided interfaces

- IncomingEmergencies
  - `void notifyEmergency(GatewayId gatewayId, SensorDataPackage emergencySensorData)`
    - Effect: The `GatewayFacade` will look up the patient corresponding to the gateway identified by `gatewayId` and forward the emergency notification (i.e., the patient identifier and the given emergency sensor data) to the `EmergencyHandler`.
    - Exceptions: None
- IncomingSensorData
  - `void sendSensorData(GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp nextDeadline)`
    - Effect: The `GatewayFacade` will look up the patient corresponding to the gateway identified by `gatewayId`, store the new sensor data labeled with the current time as time-stamp in the `SensorDataDB` and launch a new risk estimation at the `RiskEstimator` for the patient. In order to launch the new risk estimation, the `GatewayFacade` passes the patient identifier, the received sensor data and the time-stamp. The sensor data is passed so that the `RiskEstimator` does not have to fetch it any more. The time-stamp is passed so that the `RiskEstimator` does not fetch the new sensor data from the `SensorDataDB`. In addition, the `GatewayFacade` will store and keep track of the deadline specified by `nextDeadline` for this gateway.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `GatewayFacade` will respond to the ping request by sending an echo response. This is used to check the availability of the `GatewayFacade`.
    - Exceptions: None
- DiagnosticTests
  - `DiagnosticTestResults performDiagnosticTests(GatewayId gatewayId) throws NoSuchGatewayException`
    - Effect: The `GatewayFacade` will forward this request to the gateway with the specified id and return the results returned by that gateway.
    - Exceptions:
      - NoSuchGatewayException: Thrown when no gateway with the specified identifier exists.
- PullSensorData
  - `void pullSensorData(GatewayId gatewayId) throws NoSuchGatewayException`
    - Effect: The `GatewayFacade` will forward this request to the gateway identified by `gatewayId` and sends the received sensor data package (together with the identifier of the corresponding patient and a time-stamp specifying the arrival time) to the `RiskEstimator` to launch a risk estimation.
    - Exceptions:
      - NoSuchGatewayException: Thrown when no gateway with the specified identifier exists.

A.10 GatewayLogic

- **Description:** The `GatewayLogic` is responsible for communicating with the gateway devices, storing the new sensor data and launching risk level estimations. On each newly received sensor data package, the `GatewayLogic` notifies the `DeadlineChecker` and also passes the deadline of the next package as given by the gateway.
- **Super-component:** `GatewayFacade`
- **Sub-components:** None

Provided interfaces

- IncomingSensorData
  - `void sendSensorData(GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp nextDeadline)`
    - Effect: The `GatewayLogic` will look up the patient corresponding to the gateway identified by `gatewayId`, store the new sensor data labeled with the current time as time-stamp in the `SensorDataDB` and launch a new risk estimation at the `RiskEstimator` for the patient. In order to launch the new risk estimation, the `GatewayLogic` passes the patient identifier, the received sensor data and the time-stamp. The sensor data is passed so that the `RiskEstimator` does not have to fetch it any more. The time-stamp is passed so that the `RiskEstimator` does not fetch the new sensor data from the `SensorDataDB`. In addition, the `GatewayLogic` will forward the deadline specified by `nextDeadline` to the `DeadlineChecker`.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `GatewayLogic` will respond to the ping request by sending an echo response. This is used to check the availability of the `GatewayLogic`.
    - Exceptions: None
- DiagnosticTests
  - `DiagnosticTestResults performDiagnosticTests(GatewayId gatewayId) throws NoSuchGatewayException`
    - Effect: The `GatewayLogic` will forward this request to the gateway identified by `gatewayId` and return the results returned by that gateway.
    - Exceptions:
      - NoSuchGatewayException: Thrown when no gateway with the specified identifier exists.
- PullSensorData
  - `void pullSensorData(GatewayId gatewayId) throws NoSuchGatewayException`
    - Effect: The `GatewayLogic` will forward this request to the gateway identified by `gatewayId` and sends the received sensor data package (together with the identifier of the corresponding patient and a time-stamp specifying the arrival time) to the `RiskEstimator` to launch a risk estimation.
    - Exceptions:
      - NoSuchGatewayException: Thrown when no gateway with the specified identifier exists.

A.11 HIS

- **Description:** The `HIS` is external to the PMS and represents the hospital information system that communicates with the PMS.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- NotifyEmergencyServices
  - `void notifyEmergencyServices(PatientDetails patientDetails, optional NotificationMessage msg)`
    - Effect: The `HIS` will deliver the emergency notification (consisting of the details of the patient experiencing the emergency and an optional message) to the emergency services, which will handle the emergency situation.
    - Exceptions: None
- PatientRecordMgmt
  - `void updateRiskLevel(HISPatientId hisPatientId, PatientStatus status, optional SensorDataPackage sensorData)`
    - Effect: The `HIS` will update the risk level contained in `status`. If `sensorData` is given, it will be stored in the patient record of the patient identified by `hisPatientId` as well.
    - Exceptions: None
  - `List<HISPatientDescription> getAllPatientsDescriptions()`
    - Effect: The `HIS` will return the short description of all patients in this `HIS`. This description contains at least the patient's name and the `HIS` identifier.
    - Exceptions: None
  - `Tuple<PatientRecord> getPatientRecord(HISPatientId hisPatientId)`
    - Effect: The `HIS` will return the patient record of the patient identified by `hisPatientId`.
    - Exceptions: None
  - `void addEmergency(HISPatientId hisPatientId, SensorDataPackage emergencySensorData, Boolean realEmergency)`
    - Effect: The `HIS` adds the sensor data package `emergencySensorData` to the patient record of the patient identified by `hisPatientId`. The `realEmergency` parameter indicates whether this was a real emergency or not.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `HIS` will respond to the ping request by sending an echo response. This is used to check the availability of the `HIS` and the communication channel to it.
    - Exceptions: None

A.12 HISAvailabilityMonitor

- **Description:** The `HISAvailabilityMonitor` is responsible for checking the availability of the `HIS` (and the communication channel to it) according to *Av2*.
- **Super-component:** `HISFacade`
- **Sub-components:** None

Provided interfaces

None

A.13 HISFacade

- **Description:** The `HISFacade` provides the main interface of the system to the HIS.
- **Super-component:** None
- **Sub-components:** `EmergencyServicesNotifier`, `HISAvailabilityMonitor`, `HISLogic`, `PatientRecordReadCache` and `PatientRecordUpdateBuffer`

Provided interfaces

- NotifyEmergencyServices
  - `void notifyEmergencyServices(PatientDetails patientDetails, optional NotificationMessage msg)`
    - Effect: The `HISFacade` will deliver the given emergency notification (consisting of the details of the patient experiencing the emergency and an optional message) to the emergency services. The emergency services will then handle the emergency situation.
    - Exceptions: None
- PatientRecordMgmt
  - `void updateRiskLevel(PatientId patientId, PatientStatus status, optional SensorDataPackage sensorData) throws NoSuchPatientException`
    - Effect: The `HISFacade` will update the risk level of the patient record of the patient identified by `patientId` through the `HIS`. If `sensorData` is given, it will be stored in the patient record as well as the sensor data which triggered this update. If the patient record for the patient is cached in the `PatientRecordReadCache`, this cached version will be updated as well.
    - Exceptions:
      - NoSuchPatientException: Thrown if no patient with the given identifier exists.
  - `List<HISPatientDescription> getAllPatientsDescriptions()`
    - Effect: The `HISFacade` will fetch the short descriptions of all patients in the `HIS` and return them. This description contains at least the patient's name and the `HIS` identifier.
    - Exceptions: None
  - `Tuple<PatientRecord, Boolean> getPatientRecord(PatientId patientId) throws NoSuchPatientRecordException`
    - Effect: The `HISFacade` will fetch the EHR record of the patient with given id from the `HIS` and return it. If the `HIS` is not available, an older (cached) copy of the patient record is returned. In both cases, the `HISFacade` also returns whether the result is from the cache or not.
    - Exceptions:
      - NoSuchPatientRecordException: Thrown if no patient record for the patient with the given identifier exists.
  - `void addEmergency(PatientId id, SensorDataPackage emergencySensorData, Boolean realEmergency) throws NoSuchPatientException`
    - Effect: The `HISFacade` will update the patient record of the patient identified by `patientId` in the EHR through the `HIS`. The sensor data package `emergencySensorData` triggered an emergency notification to the PMS and will therefore be added to the patient record. The parameter `realEmergency` indicates whether the emergency was a real emergency (i.e., the PMS notified the emergency services after verification) or not.

    - Exceptions:
        - NoSuchPatientException: Thrown if no patient with the given identifier exists.

- Ping
  - `Echo ping()`
    - Effect: The `HISFacade` will respond to the ping request by sending an echo response. This is used to check the availability of the `HISFacade`.
    - Exceptions: None

A.14 HISLogic

- **Description:** The `HISLogic` is responsible for interacting with the `HIS` for the patient record.
- **Super-component:** `HISFacade`
- **Sub-components:** None

Provided interfaces

- PatientRecordMgmt
  - `void updateRiskLevel(PatientId patientId, PatientStatus status, optional SensorDataPackage sensorData) throws NoSuchPatientException`
    - Effect: The `HISLogic` will update the risk level of the patient record of the patient identified by `patientId` through the `HIS`. If `sensorData` is given, it will be stored in the patient record as well as the sensor data which triggered this update. If the patient record for the patient is cached in the `PatientRecordReadCache`, this cached version will be updated as well.
    - Exceptions:
        - NoSuchPatientException: Thrown if no patient with the given identifier exists.
  - `List<HISPatientDescription> getAllPatientsDescriptions()`
    - Effect: The `HISLogic` will fetch the short descriptions of all patients in the `HIS` and return them. This description contains at least the patient's name and the `HIS` identifier.
    - Exceptions: None
  - `Tuple<PatientRecord, Boolean> getPatientRecord(PatientId patientId) throws NoSuchPatientRecordException`
    - Effect: The `HISLogic` will fetch the EHR record of the patient with given id from the `HIS` and return it. If the `HIS` is not available, an older (cached) copy of the patient record is returned if possible. If the cache does not contain the patient record of the patient with given id, an exception is thrown. In both successful cases, the `HISFacade` also returns whether the result is from the cache or not.
    - Exceptions:
        - NoSuchPatientRecordException: Thrown if no patient record for the patient with the given identifier exists in the HIS or if the HIS is not available and no such patient record exists in the cache.
  - `void addEmergency(PatientId id, SensorDataPackage emergencySensorData, Boolean realEmergency) throws NoSuchPatientException`
    - Effect: The `HISLogic` will update the patient record of the patient identified by `patientId` in the EHR through the `HIS`. The sensor data package `emergencySensorData` triggered an emergency notification to the PMS and will therefore be added to the patient record. The parameter `realEmergency` indicates whether the emergency was a real emergency (i.e., the PMS notified the emergency services after verification) or not.
    - Exceptions:

· NoSuchPatientException: Thrown if no patient with the given identifier exists.

- Ping
  - `Echo ping()`
    - Effect: The `HISLogic` will respond to the ping request by sending an echo response. This is used to check the availability of the `HISLogic`.
    - Exceptions: None
- SetDegradedModus
  - `void setDegradedModus()`
    - Effect: The `HISLogic` will switch to degraded modus. This affects using the cache or directly using this `HIS` for handling read and write requests.
    - Exceptions: None
  - `void setNondegradedModus()`
    - Effect: The `HISLogic` will switch to non-degraded (i.e., normal) modus. This affects using the cache or directly using this `HIS` for handling read and write requests.
    - Exceptions: None

A.15 HospitalUsersClient

- **Description:** The `HospitalUsersClient` is external to the PMS and represents the client device of the user employed at the hospital that communicates with the PMS.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- DeliverNotification
  - `void deliverNotification(NotificationMessage msg)`
    - Effect: Deliver the notification message `msg` to the user.
    - Exceptions: None

A.16 HospitalUsersFacade

- **Description:** The `HospitalUsersFacade` provides the main interface of the system to the human users employed at the hospital, e.g., the cardiologists and the technical nurse.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- AppointTrustee
  - `void createNewTrustee(SessionId sessionId, TrusteeContactDetails details, Boolean isBuddy, PatientId patientId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier and then create a new trustee account in the `OtherDataDB` using the given contact information and whether the new trustee also is a buddy. Furthermore, this trustee is associated to the patient identified by `patientId`. The contact information at least contains the new trustee's e-mail address. The `HospitalUsersFacade` will then contact the new trustee on this e-mail address with detailed information on how to crate an account in the PMS,

including an identifier of the trustee invitation. The new trustee can use this identifier to confirm or decline the invitation. Finally, the `HospitalUsersFacade` will log this event in the `OtherDataDB`.

  - Exceptions:
    - NotAuthenticatedException: Thrown if the given session identifier is invalid.

- AuthN
  - `SessionId login(Credentials credentials) throws IncorrectCredentialsException`
    - Effect: The `HospitalUsersFacade` will forward the given `credentials` to the `AuthNHandler` which will verify them and return a new session identifier if correct. This session identifier can be used in future requests to the `HospitalUsersFacade`.
    - Exceptions:
      - IncorrectCredentialsException: Thrown if the `AuthNHandler` indicated that the given credentials were incorrect.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `HospitalUsersFacade` will remove the session corresponding to `sessionId` using the `AuthNHandler`. As a result, this session cannot be used anymore to access the system without logging in again. If the no session corresponds to `sessionId` does not exists, nothing is changed, but no exception is thrown.
    - Exceptions: None

- ClinicalModels
  - `List<ClinicalModelDescription> getAvailableClinicalModels(SessionId sessionId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then fetch all available clinical models from the `OtherDataDB` and will return their descriptions.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `List<ClinicalModelDescription> getAssignedClinicalModelsForPatient(SessionId sessionId, PatientId patientId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then fetch all clinical models assigned to the patient identified by `patientId` from the `OtherDataDB` and will return their descriptions.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void assignClinicalModel(SessionId sessionId, PatientId patientId, ClinicalModelId clinicalModelId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then store the assignment of the clinical model identified by `clinicalModelId` to the patient identified by `patientId` in the `OtherDataDB` and will instantiate the default configuration of this clinical model for this patient. In addition, the `HospitalUsersFacade` will invalidate stale entries of the clinical model cache for this patient in the `RiskEstimator`.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void deassignClinicalModel(SessionId sessionId, PatientId patientId, ClinicalModelId clinicalModelId) throws NotAuthenticatedException`

    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then remove the assignment of the clinical model identified by `clinicalModelId` to the patient identified by `patientId`. The personalized configuration of this clinical model for this patient is also removed. If this clinical model was not assigned to this patient, nothing is changed, but no exception is thrown either. In addition, the `HospitalUsersFacade` will invalidate stale entries of the clinical model cache for this patient in the `RiskEstimator`.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `Map<ClinicalModelParameterKey, ClinicalModelParameterValue> getClinicalModelConfiguration(SessionId sessionId, PatientId patientId, ClinicalModelId clinicalModelId) throws NotAuthenticatedException, NoSuchConfigurationException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then fetch the current configuration of the clinical model corresponding to `clinicalmodelId` for the patient identified by `patientId` and will return it as a map of configuration parameters and their value.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void updateClinicalModelConfiguration(SessionId sessionId, PatientId patientId, ClinicalModelId clinicalModelId, ClinicalModelParameterKey key, ClinicalModelParameterValue value) throws NotAuthenticatedException, NoSuchClinicalModelParameterException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then store the new value of the given configuration parameter in the `OtherDataDB` for the clinical model with corresponding to `clinicalModelId` for the patient identified by `patientId`. In addition, the `HospitalUsersFacade` will invalidate stale cache entries of this clinical model for this patient in the `RiskEstimator`.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
      - NoSuchClinicalModelParameterException: Thrown when the given clinical model parameter is invalid.

- ConsultPatientStatus
  - `PatientStatusOverview getPatientStatusOverview(SessionId sessionId, PatientId patientId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then request the `PatientStatusOverviewHandler` to generate a status overview for the patient identified by `patientId`. This status overview will be customized depending on the role of the caller, which is determined based on the caller's identifier found in the `sessionId`. The generated status overview is then returned to the caller.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.

- ManageQuestionnaires
  - `void createQuestionnaire(SessionId sessionId, List<UserId> targetUsers, List<Question> questions) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then store the new questionnaire in the `OtherDataDB`. More precisely, the `HospitalUsersFacade` stores the questions to be filled out, the user who created the questionnaire, the time of creation and the target users who should fill out the questionnaire.

    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `List<QuestionnaireDescription> getCreatedQuestionnaires(SessionId sessionId, UserId userId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then fetch all questionnaires created by the user identified by `userId` from the `OtherDataDB` and will return them to the caller.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `Questionnaire getQuestionnaire(SessionId sessionId, QuestionnaireId questionnaireId) throws NotAuthenticatedException, NoSuchQuestionnaireException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will fetch the questionnaire corresponding to `questionnaireId` from the `OtherDataDB` and will return it to the caller.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - NoSucQuestionnaireException: Thrown if no questionnaire exists with the given identifier.

- OndemandConsultation
  - `PatientStatusOverview performOndemandConsultation(SessionId sessionId, PatientId patientId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will request the `GatewayFacade` to get the latest sensor data from the gateway associated with the patient identified by `patientId`. Then the `HospitalUsersFacade` will poll the `OtherDataDB` to check whether the risk estimation triggered by pulling the sensor data has completed (i.e., by checking the time-stamp of the last estimated risk level has changed). On completion of the risk estimation the `HospitalUsersFacade` will request the `PatientStatusOverviewHandler` to create a status overview for the patient and then return this overview to the original caller.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.

- PatientRegistration
  - `List<HISPatientDescription> getAllHISPatients(SessionId sessionId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then fetch the descriptions of all patients in the `HIS` and will return them. This description at least contains the patient's name and `HIS` identifier.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `PatientId createNewPatient(SessionId sessionId, HISPatientId hisPatientId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then create a new patient account in the `OtherDataDB` based on the description of the patient retrieved from the `HIS`. More precisely, the new PMS patient account will contain at least the patient's name and his or her `HISPatientId`. No credentials or devices are assigned to this new patient account yet. Finally, the `HospitalUsersFacade` will return the unique identifier for this new patient account in the PMS.

    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `List<GatewayId> getAvailableGateways(SessionId sessionId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then fetch and return the identifiers of all gateways currenlty not associated with a patient.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `GatewayInitializationResults associateGatewayToPatient(SessionId sessionId, GatewayId gatewayId, PatientId patientId) throws NotAuthenticatedException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then store the association of the gateway corresponding to `gatewayId` to the patient identified by `patientId` in the `OtherDataDB`. The `HospitalUsersFacade` will then initialize the gateway by performing diagnostic tests, requesting first sensor data and returning the results to the caller.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void setPatientCredentials(SessionId sessionId, PatientId patientId, Credentials credentials) throws NotAuthenticatedException, NoSuchPatientException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then store the given `credentials` for the patient identified by `patientId` in the `OtherDataDB`. From now on, this patient can use these credentials to log in to the `OtherUsersFacade`.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - NoSuchPatientException: Thrown if no patient with the given identifier exists.

- UpdateRiskLevel
  - `void updateRiskLevel(SessionId sessionId, PatientId patientId, PatientStatus status) throws NotAuthenticatedException, NoSuchPatientException`
    - Effect: The `HospitalUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `HospitalUsersFacade` will then store the new risk level for the patient identified by `patientId` in both the `OtherDataDB` and the EHR.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - NoSuchPatientException: Thrown if no patient exists with the given identifer.

A.17 NotificationHandler

- **Description:** The `NotificationHandler` is responsible for sending notifications to the appropriate parties, e.g., PMS system administrators, patients or trustees. For notifications regarding a certain patient, the `NotificationHandler` employs the subscriptions in the `OtherDataDB`.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- NotifySysAdmin
  - `void notifySysAdmin(NotificationMessage msg)`

    - Effect: The `NotificationHandler` will send the given notification message `msg` to the PMS system administrators.
    - Exceptions: None

- NotifySubscribers
  - `void notifySubscribersOfPatient(PatientId patientId, NotificationMessage msg)`
    - Effect: The `NotificationHandler` will fetch the list of subscribers for the patient identified by `patientId` and their contact addresses from the `OtherDataDB` and send the given message to all subscribers.
    - Exceptions: None

A.18 OtherDataDB

- **Description:** The `OtherDataDB` is responsible for storing the all data required by the PMS except for sensor data. For all users this includes account information such as name, credentials and contact information. For patients this furthermore contains their current status (i.e., current risk level and last estimated risk level), associated devices and trustees, clinical model configurations. Furthermore, this contains the clinical models available in the PMS, subscriptions of users to notifications concerning other users, questionnaires, any pending invitations to new trustees and the log of events which occurred in the PMS.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- OtherDataMgmt
  - `void addLog(LogSource source, LogMessage msg)`
    - Effect: The `OtherDataDB` will store the given log message `msg` associated with a log source `source` and a generated time-stamp.
    - Exceptions: None
  - `void assignClinicalModel(PatientId patientId, ClinicalModelId clinicalModelId)`
    - Effect: The `OtherDataDB` will store the assignment of the clinical model identified by `clinicalModelId` to the patient idendified by `patientId` and will instantiate the default configuration of this clinical model for this patient.
    - Exceptions: None
  - `void associateGatewayToPatient(GatewayId gatewayId, PatientId patientId)`
    - Effect: The `OtherDataDB` will store the association of the gateway identified by `gatewayId` to the patient identified by `patientId`.
    - Exceptions: None
  - `InvitationId createNewInvitationId(UserId trusteeId)`
    - Effect: The `OtherDataDB` will generate a new unique invitation identifier, store this, associate the trustee identified by `trusteeId` to it and return the generated identifier to the caller..
    - Exceptions: None
  - `PatientId createNewPatientAccount(HISPatientDescription hisPatientDescription)`
    - Effect: The `OtherDataDB` will create a new patient account in the PMS based on the description of the patient from the `HIS`. More precisely, the new PMS patient account will contain at least the patient's name and his or her `HISPatientId`. No credentials or devices are assigned to this new patient account yet. Finally, the `OtherDataDB` will return the unique identifier in the PMS for the new patient account.

    * Exceptions: None

- `UserId createNewTrusteeAccount(TrusteeContactDetails contactDetails, Boolean isBuddy, PatientId patientId)`
    * Effect: The `OtherDataDB` will create a new trustee account using the given contact information `contactDetails` and whether this trustee also is a buddy and associate this trustee to the patient identified by `patientId` and return the identifier of the new account. The given contact information contains at least the new trustee's e-mail address.
    * Exceptions: None
- `void deassignClinicalModel(PatientId patientId, ClinicalModelId clinicalModelId)`
    * Effect: The `OtherDataDB` will remove the assignment of the clinical identified by `clinicalModelId` to the patient identified by `patientId`. The personalized configuration of this clinical model for this patient is also removed. If this clinical model was not assigned to this patient, nothing is changed, but no exception is thrown either.
    * Exceptions: None
- `List<ClinicalModelDescription> getAvailableClinicalModels()`
    * Effect: The `OtherDataDB` will fetch and return the description of all clinical models in the database.
    * Exceptions: None
- `Map<ClinicalModelId, Map<ClinicalModelParameterKey, ClinicalModelParameterValue>> getAssignedClinicalModelsAndConfigForPatient(PatientId patientId)`
    * Effect: The `OtherDataDB` will fetch and return the ids of the clinical models assigned to the patient identified by `patientId` and their configurations for this patient. The configurations are returned as a map of configuration parameters and their value.
    * Exceptions: None
- `List<ClinicalModelDescription> getAssignedClinicalModelsForPatient(PatientId patientId)`
    * Effect: The `OtherDataDB` will fetch and return the description of all clinical models assigned to the patient identified by `patientId`. If no patient with given id exists, an empty list is returned.
    * Exceptions: None
- `Map<ClinicalModelParameterKey, ClinicalModelParameterValue> getClinicalModelConfiguration(PatientId patientId, ClinicalModelId clinicalModelId) throws NoSuchConfigurationException`
    * Effect: The `OtherDataDB` will fetch the current configuration of the clinical model identified by `clinicalModelId` for the patient identified by `patientId` and will return it as a map of configuration parameters and their value.
    * Exceptions:
        · NoSuchConfigurationException: Thrown if no configurations for the given patient id and clinical model id exists.
- `Destination getContactAddress(UserId userId) throws NoSuchUserException`
    * Effect: The `OtherDataDB` will fetch and return the contact address for user identified by `userid`.
    * Exceptions:
        · NoSuchUserException: Thrown if no user with the given identifier exists.
- `List<QuestionnaireDescription> getCreatedQuestionnaires(UserId userId)`
    * Effect: The `OtherDataDB` will fetch all questionnaires created by the user with given `UserId` and return them.
    * Exceptions: None
- `Tuple<PatientStatus, TimeStamp> getEstimatedPatientStatus(PatientId patientId) throws NoSuchPatientException`

    * Effect: The `OtherDataDB` will fetch and return the last estimated status together with the time of estimation of the patient identified by `PatientId`.
    * Exceptions:
        · NoSuchPatientException: Thrown if no patient with the given identifier exists.

- `List<Questionnaire> getFilledOutQuestionnaires(PatientId patientId)`
    * Effect: The `OtherDataDB` will fetch and return all questionnaires filled out for the patient identified by `patientId`.
    * Exceptions: None

- `GatewayId getGatwayAssociatedToPatient(PatientId patientId) throws NoSuchPatientException`
    * Effect: The `OtherDataDB` will return the gateway associated to the patient with identified by `patientId`.
    * Exceptions:
        · NoSuchPatientException: Thrown if no patient with the given identifier exists.

- `HISPatientId getHISPatientIdForPatient(PatientId id) throws NoSuchPatientException`
    * Effect: The `OtherDataDB` will return the identifier used by the `HIS` for the patient identified in the PMS by `id`.
    * Exceptions:
        · NoSuchPatientxception: Thrown if no patient with the given identifier exists.

- `PatientId getPatientAssociatedToGateway(GatewayId gatewayId) throws NoSuchGatewayException`
    * Effect: The `OtherDataDB` will return the patient associated to the gateway identified by `gatewayId`.
    * Exceptions:
        · NoSuchGatewayException: Thrown if no gateway with the given identifier exists.

- `PatientId getPatientAssociatedToTrustee(UserId trusteeId) throws NoSuchUserException`
    * Effect: The `OtherDataDB` will fetch and return the identifier of the patient associated to the trustee identified by `trusteeId`.
    * Exceptions:
        · NoSuchUserException: Thrown if no trustee with the given identifier exists.

- `PatientDetails getPatientDetails(PatientId id) throws NoSuchPatientException`
    * Effect: The `OtherDataDB` will fetch and return the patient details for the patient identified by `id`.
    * Exceptions:
        · NoSuchPatientException: Thrown if no patient with the given identifier exists.

- `PatientStatus getPatientStatus(PatientId patientId) throws NoSuchPatientException`
    * Effect: The `OtherDataDB` will fetch and return the status of the patient identified by `patientId`.
    * Exceptions:
        · NoSuchPatientException: Thrown if no patient with the given identifier exists.

- `Questionnaire getQuestionnaire(QuestionnaireId questionnaireId) throws NoSuchQuestionnaireException`
    * Effect: The `OtherDataDB` will fetch and return the questionnaire identified by `questionnaireId`.
    * Exceptions:
        · NoSuchQuestionnarieException: Thrown if no quesitonnaire with the given identifier exists.

- `List<UserId> getSubscribers(PatientId patientId)`

    - Effect: The `OtherDataDB` will fetch and return the list of identifiers of all users subscribed to notifications of the patient identified by `patientId`.
    - Exceptions: None
- `Credentials getUserCredentials(UserId userId) throws NoSuchUserException`
    - Effect: The `OtherDataDB` will return the credentials belonging to the user identified by the given `userId`.
    - Exceptions:
        - NoSuchUserException: Thrown if no user with the given identifier exists.
- `UserId getTrusteeOfInvitation(InvitationId invitationId) throws NoSuchInvitationException`
    - Effect: The `OtherDataDB` will return the identifier of the trustee associated to the given invitation.
    - Exceptions:
        - NoSuchInvitationException: Thrown if no invitation with the given identifier exists.
- `Role getUserRole(UserId userId) throws NoSuchUserException`
    - Effect: The `OtherDataDB` will return the role of the user identified by `userId`. Examples of such a role are "cardiologist", "patient", "gp" and "trustee". etc.
    - Exceptions:
        - NoSuchUserException: Thrown if no user with the given identifier exists.
- `Boolean invitationIdExists(InvitationId invitationId)`
    - Effect: The `OtherDataDB` will return whether an invitation corresponding to `invitationId` exists.
    - Exceptions: None
- `void removeInvitation(InvitationId invitationId)`
    - Effect: The `OtherDataDB` will remove the trustee invitation with given id. If no such invitation exits, nothing is changed and no exception is thrown.
    - Exceptions: None
- `void setContactAddress(UserId userId, Destination destination)`
    - Effect: The `OtherDataDB` will store the contact address for the given user.
    - Exceptions: None
- `void setEstimatedPatientStatus(PatientId patientId, PatientStatus estimatedStatus, TimeStamp estimationTime) throws NoSuchPatientException`
    - Effect: The `OtherDataDB` will update the patient status estimation of the patient identified by `patientId` to the given value `estimatedStatus` and update the time of estimation to `estimationTime`. The time of estimation indicates when the last estimation for this patient was completed.
    - Exceptions:
        - NoSuchPatientException: Thrown if no patient with the given identifier exists.
- `void setPatientStatus(PatientId patientId, PatientStatus status)`
    - Effect: The `OtherDataDB` will store the given status for the patient identified by `PatientId`.
    - Exceptions: None
- `void storeQuestionnaire(List<UserId> targetUsers, List<Question> questions, UserId createdBy)`
    - Effect: The `OtherDataDB` will store the new questionnaire, i.e., the `questions` to be filled out, the user who created the questionnaire (`createdBy`) and the target users who should fill out the questionnaire (`targetUsers`). Moreover, the `OtherDataDB` will also store the time of creation.
    - Exceptions: None

- `void storeSubscription(PatientId patientId, UserId subscriberId)`
  - Effect: The `OtherDataDB` will store the subscription of the user identified by `subscriberId` to the patient identified by `patientId`.
  - Exceptions: None
- `void updateClinicalModelConfiguration(PatientId patientId, ClinicalModelId clinicalModelId, ClinicalModelParameterKey key, ClinicalModelParameterValue value) throws NoSuchConfigurationException`
  - Effect: The `OtherDataDB` will store the new value of the given configuration parameter for the clinical model identified by `clinicalModelId` for the patient identified by `patientId`.
  - Exceptions:
    - NoSuchConfigurationException: Thrown if no configurations for the given patient and clinical model exists.
- `Boolean userExists(UserId userId)`
  - Effect: The `OtherDataDB` will verify and return whether a user with given `userId` exists in its database.
  - Exceptions: None

A.19 OtherUsersUsersClient

- **Description:** The `OtherUsersUsersClient` is external to the PMS and represents the client device of human users that communicate with the PMS, apart from the users employed at the hospital.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- DeliverNotification
  - `void deliverNotification(NotificationMessage msg)`
    - Effect: Deliver the notification message `msg` to the user.
    - Exceptions: None

A.20 OtherUsersFacade

- **Description:** The `OtherUsersFacade` provides the main interface of the system to the human users apart from the users employed at the hospital. More specifically, the `OtherUsersFacade` is used by the patients, the trustees, the telemedicine operators, the home caretakers and the GPs.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- Confirm/Decline
  - `SessionId confirm(InvitationId invitationId) throws NoSuchInvitationException`
    - Effect: The `OtherUsersFacade` will verify the given invitation identifier `invitationId` using the `OtherDataDB` and fetch the identifier of the associated trustee and remove the invitation from the `OtherDataDB`. The `OtherUsersFacade` then logs this event in the `OtherDataDB`. The patient associated to the trustee is send a confirmation message. Finally,the trustee is logged in using the `AuthNHandler` and a new session identifier is returned to the caller.

    - Exceptions:
        - NoSuchInvitationException: Thrown if no invitation exists with the given identifier.
  - `void completeNewTrusteeAccount(SessionId sessionId, Credentials credentials) throws NotAuthenticatedException`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `OtherUsersFacade` will then complete the account of the new trustee with the given information, i.e., add the credentials chosen by the trustee.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier `sessionId` is invalid.
  - `void decline(InvitationId invitationId) throws NoSuchInvitationException`
    - Effect: The `OtherUsersFacade` will remove the invitation from the `OtherDataDB` and notify the patient of the trustee's decline.
    - Exceptions:
        - NoSuchInvitationException: Thrown if no invitation exsts with the given identifier.

- AppointTrustee
  - `void createNewTrusteeAsPatient(SessionId sessionId, TrusteeContactDetails contact, Boolean isBuddy) throws NotAuthenticatedException`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `OtherUsersFacade` will then create a new trustee account in the `OtherDataDB` using the given contact information and associate this trustee to the patient identified by the patient identifier in the given `SessionId`. The contact information contains at least the new trustee's e-mail address. The `OtherUsersFacade` will then contact the new trustee on this e-mail address with detailed information on how to create an account in the PMS, including an identifier of the trustee invitation. The new trustee can use this identifier to confirm or decline the invitation. Finally, the `OtherUsersFacade` will log this event in `OtherDataDB`
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void createNewTrustee(SessionId sessionId, TrusteeContactDetails contact, Boolean isBuddy, PatientId patientId) throws NotAuthenticatedException`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `OtherUsersFacade` will then create a new trustee account in the `OtherDataDB` using the given contact information and associate this trustee to the patient identified by `patientId`. The contact information contains at least the new trustee's e-mail address. The `OtherUsersFacade` will then contact the new trustee on this e-mail address with detailed information on how to create an account in the PMS, including an identifier of the trustee invitation. The new trustee can use this identifier to confirm or decline the invitation. Finally, the `OtherUsersFacade` will log this event in `OtherDataDB`
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.

- FillOutQuestionnaires
  - `List<QuestionnaireId> getQuestionnairesToFillOut(SessionId sessionId, UserId userId) throws NotAuthenticatedException`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier using the `AuthNHandler`. The `OtherUsersFacade` will then fetch all questionnaires to be filled out by the user identified by `userId` from the `OtherDataDB` and return a list containing their identifiers.

    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
- `List<QuestionnaireId> getQuestionnairesToFillOutAsPatient(SessionId sessionId) throws NotAuthenticatedException`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier using the `AuthNHandler` and retrieve the identifier of the patient of the session. The `OtherUsersFacade` will then fetch all questionnaires to be filled out by the patient identified by the retrieved identifier from the `OtherDataDB` and will return a list containing their identifiers.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
- `Questionnaire getQuestionnaire(SessionId sessionId, QuestionnaireId questionnaireId) throws NotAuthenticatedException, NoSuchQuestionnaireException`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `OtherUsersFacade` will then fetch the questionnaire corresponding to `questionnaireId` from the `OtherDataDB` and will return it to the caller.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
- `void fillOutQuestionnaire(SessionId sessionId, QuestionnaireId questionnaireId, List<Question, Answer> answers) throws NotAuthenticatedException, NoSuchQuestionnaire-Exception`
    - Effect: The `OtherUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `OtherUsersFacade` will then check the provided answers (e.g., whether all questions in the questionnaire are answered). If no problems are detected, the `OtherUsersFacade` will store the answers in the `OtherDataDB` and log this event in the `OtherDataDB`. Finally the `OtherUsersFacade` will retrieve the identifier of the target patient of this questionnaire from the `OtherDataDB` and schedule a new risk estimation for this patient.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - NoSuchQuestionnaireException: Thrown if no questionnaire exists with the given identifier.

- AuthN
    - `SessionId login(Credentials credentials) throws IncorrectCredentialsException`
        - Effect: The `OtherUsersFacade` will forward the given credentials to the `AuthNHandler` which will verify the credentials and return a new session identifier if correct. This session identifier can be used in future requests to the `OtherUsersFacade`.
        - Exceptions:
            - IncorrectCredentialsException: Thrown if the `AuthNHandler` indicated that the given credentials were incorrect.
    - `Boolean logout(SessionId sessionId)`
        - Effect: The `OtherUsersFacade` will remove the session corresponding to `sessionId` using the `AuthNHandler`. As a result, this session identifier cannot be used anymore to access the system without logging in again. If the given session identifier does not exists, nothing is changed, but no exception is thrown.
        - Exceptions: None

- ConsultPatientStatus
    - `PatientStatusOverview getPatientStatusOverview(SessionId sessionId, PatientId patientId) throws NotAuthenticatedException, NoSuchPatientException`

    - Effect: The `OtherUsersFacade` will first verify the given session identifier `sessionId` using the `AuthNHandler`. The `OtherUsersFacade` will then fetch the identifier of the user of the current session from the `AuthNHandler`. Finally the `OtherUsersFacade` will ask the `PatientStatusOverviewHandler` to generate a status overview of the patient identified by `patientId` for the user identified by the fetched identifer and will return this status overview to the caller.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - NoSuchPatientException: Throw if no patients exists with the given identifier.

A.21 PatientRecordReadCache

- **Description:** The `PatientRecordReadCache` provides a cache of patient records from the `HIS` for use in degraded modus (i.e., when the `HIS` cannot be reached).
- **Super-component:** `HISFacade`
- **Sub-components:** None

Provided interfaces

- ReadCacheMgmt
  - `void storeHISPatientDescriptions(List<HISPatientDescription> descriptions)`
    - Effect: The `PatientRecordReadCache` will store the list of `descriptions` of all patients in this `HIS`. Since the given list is regarded to be the complete list of descriptions of all patients, all previous descriptions be removed first.
    - Exceptions: None
  - `List<HISPatientDescription> getAllHISPatientDescriptions()`
    - Effect: The `PatientRecordReadCache` will return all descriptions of patients in the `HIS` currently stored in its cache. This description contains at least the patient's name and the `HIS` identifier.
    - Exceptions: None
  - `void storePatientRecord(HISPatientId hisPatientId, PatientRecord patientRecord)`
    - Effect: The `PatientRecordReadCache` will store the EHR record of the patient identified by the `HIS` by `hisPatientId` in its cache. If the `PatientRecordReadCache` already contains an entry for this patient, this entry is replaced.
    - Exceptions: None
  - `PatientRecord getPatientRecord(HISPatientId hisPatientId) throws NoSuchPatient-RecordException`
    - Effect: The `PatientRecordReadCache` will return the EHR record of the patient identified by the `HIS` by `hisPatientId` from its cache.
    - Exceptions:
        - NoSuchPatientException: Thrown if the cache does not contain an EHR for the patient with given identifier.

A.22 PatientRecordUpdateBuffer

- **Description:** The `PatientRecordUpdateBuffer` provides a buffer for temporarily storing patient record updates in degraded modus (i.e., when the `HIS` cannot be reached).
- **Super-component:** `HISFacade`
- **Sub-components:** None

Provided interfaces

- UpdateCacheMgmt
  - `void storeRiskLevelUpdate(HISPatientId patientId, PatientStatus status, optional SensorDataPackage sensorData)`
    - Effect: The `PatientRecordUpdateBuffer` will store the given risk level update and (optionally) the sensor data package `sensorData` that caused it.
    - Exceptions: None
  - `List<Tuple<HISPatientId, PatientStatus, SensorDataPackage>> flushAllRiskLevelUpdates()`
    - Effect: The `PatientRecordUpdateBuffer` will return and remove all risk level updates in its cache. Each risk level update consists of the identifier of the patient, the new risk level and the associated sensor data (if any).
    - Exceptions: None
  - `void storeEmergency(HISPatientId hisPatientId, SensorDataPackage emergencySensorData, Boolean realEmergency)`
    - Effect: The `PatientRecordUpdateBuffer` will store the given emergency and meta-data.
    - Exceptions: None
  - `List<Tuple<HISPatientId, SensorDataPackage, Boolean>> flushAllRiskLevelUpdates()`
    - Effect: The `PatientRecordUpdateBuffer` will return and remove all emergencies in its cache. Each emergency consists of the identifier of the patient, the the sensor data that triggered the original emergency notification and whether the emergency was verified or not.
    - Exceptions: None

A.23 PatientStatusOverviewHandler

- **Description:** The `PatientStatusOverviewHandler` is responsible generating an overview of a patient status for human users. The content of this overview can differ depending on the role of the requesting user (e.g., patient or cardiologist).
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- GeneratePatientStatusOverview
  - `PatientStatusOverview generatePatientStatusOverview(PatientId patientId, UserId requestingUserId)`
    - Effect: The `PatientStatusOverviewHandler` will generate a status overview of the patient identified by `patientId` for the requesting user identified by `requestingUserId`. The contents of the status overview will differ depending on the role of the requesting user. Therefore, the `PatientStatusOverviewHandler` will first fetch the role of the requesting user from the `OtherDataDB`, will then fetch the required data from the `OtherDataDB`, the `SensorDataDB`, the `OtherDataDB`, etc. and will finally generate and return the status overview.
    - Exceptions: None

A.24 PrimaryDB

- **Description:** The `PrimaryDB` is responsible for actually storing the sensor data in the `SensorDataDB`. It receives read and write requests from the `ReplicationManager` and regularly sends all new sensor data to the `StandbyDB` to synchronize its state.
- **Super-component:** `SensorDataDB`
- **Sub-components:** None

Provided interfaces

- SensorDataMgmt:
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: The `PrimaryDB` will store the given sensor data and meta-data.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatient(PatientId patientId)`
    - Effect: The `PrimaryDB` will fetch and return all sensor data belonging to the patient identified by `patientId`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientAfter(PatientId patientId, TimeStamp startTime)`
    - Effect: The `PrimaryDB` will fetch and return all sensor data belonging to the patient identified by `PatientId` which was received after the specified time `startTime`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientAfterAndBefore(PatientId patientId, TimeStamp startTime, TimeStamp stopTime)`
    - Effect: The `PrimaryDB` will fetch and return all sensor data belonging to the patient identified by `patientId` and that entered the system between the given `startTime` and `stopTime`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId patientId, TimeStamp stopTime)`
    - Effect: The `PrimaryDB` will fetch and return all sensor data belonging to the patient identified by `patientId` which was received before the specified time `stopTime`
    - Exceptions: None
  - `Tuple<TimeStamp, SensorDataPackage> getLatestSensorDataOfPatient(PatientId patientId)`
    - Effect: The `PrimaryDB` will fetch and return the latest sensor data belonging to the patient with given identified by `patientId`.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `PrimaryDB` will respond to the ping request by sending an echo response. This is used by the `ReplicationManager` to check whether the `PrimaryDB` is available.
    - 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

- 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: The `PrimaryDB` will configure itself for operation as primary replica of the sensor data. This means it will respond to all read and write requests and periodically push new sensor data to the `StandbyDB`
    - Exceptions: None
  - `void assignStandbyRole()`
    - Effect: The `PrimaryDB` will configure itself for operation as standby replica of the sensor data. This means it will no longer responds to individual read and write requests and periodically stores all data the `PrimaryDB` pushes to it.
    - Exceptions: None

A.25 ReplicationManager

- **Description:** The `ReplicationManager` is responsible for managing the `PrimaryDB` and `StandbyDB` replicas. The `ReplicationManager` passes read requests to the `PrimaryDB` and passes write requests to both the `PrimaryDB` and the `SensorDataCache`. The `ReplicationManager` also checks the availability of the `PrimaryDB` by checking the results of all requests sent to it and pinging it if necessary, so as to check them at least every four seconds. In case of failure, the `ReplicationManager` notifies a PMS system administrator, changes the `StandbyDB` into the `PrimaryDB`, flushes the `SensorDataCache` to the new `PrimaryDB` in order to synchronize it and starts using it for reads and writes.
- **Super-component:** `SensorDataDB`
- **Sub-components:** None

Provided interfaces

- SensorDataMgmt:
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: The `ReplicationManager` will store the given sensor data package `sensorData` and meta-data in the `PrimaryDB` and the `SensorDataCache`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatient(PatientId patientId)`
    - Effect: The `ReplicationManager` will fetch and return all sensor data belonging to the patient identified by `patientId` from the `PrimaryDB`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientAfter(PatientId patientId, TimeStamp startTime)`
    - Effect: The `ReplicationManager` will fetch and return all sensor data belonging to the patient identified by `patientId` received after the given time `startTime` from the `PrimaryDB`.
    - Exceptions: None

- `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId patientId, TimeStamp stopTime)`
  - Effect: The `ReplicationManager` will fetch and return all sensor data belonging to the patient identified by `patientId` received before the given time `stopTime` from the `PrimaryDB`.
  - Exceptions: None
- `Tuple<TimeStamp, SensorDataPackage> getLatestSensorDataOfPatient(PatientId patientId)`
  - Effect: The `ReplicationManager` will fetch and return the latest sensor data packaged belinging to the patient identified by `patientId` the `PrimaryDB`.
  - Exceptions: None

A.26 RiskEstimationCombiner

- **Description:** The `RiskEstimationCombiner` is responsible for combining the results of the clinical model computation jobs belonging to a single risk estimation. More precisely, the `RiskEstimationScheduler` passes the set of scheduled jobs for a risk estimation to the `RiskEstimationCombiner` before scheduling them. The `RiskEstimationCombiner` then waits for all results to arrive, combines them, notifies the subscribers of that patient if the risk level is estimated to change and propagates the new sensor data and the results of the risk estimation to the patient record if needed. The estimated risk level together with its time of estimation is forwarded to the `OtherDataDB` for storage.
- **Super-component:** `RiskEstimator`
- **Sub-components:** None

Provided interfaces

- JobMgmt
  - `void addJobSet(RiskEstimationId 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
  - `void addJobSet(RiskEstimationId id, List<ClinicalModelJobId> jobIds, SensorDataPackage sensorData)`
    - 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. The sensor data is passed to the `RiskEstimationCombiner` in order to allow it to propagate it to the patient record if needed.
    - 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

A.27 RiskEstimationProcessor

- **Description:** The `RiskEstimationProcessor` is responsible for computing clinical models. The `RiskEstimationProcessor` fetches new clinical model computation jobs from the `RiskEstimationScheduler`), fetches the required data and passes the result to the `RiskEstimationCombiner`.
- **Super-component:** `RiskEstimator`
- **Sub-components:** None

Provided interfaces

None.

A.28 RiskEstimationScheduler

- **Description:** The `RiskEstimationScheduler` is responsible for taking in new and scheduling requests for risk estimations according to *P2*.
- **Super-component:** `RiskEstimator`
- **Sub-components:** None

Provided interfaces

- LaunchRiskEstimation
  - `void launchRiskEstimation(PatientId patientId, SensorDataPackage newSensorData, TimeStamp receivedAt)`
    - Effect: The `RiskEstimationScheduler` will fetch the clinical models and their configurations associated to the patient identified by `patientId` from the `OtherDataDB` using the `ClinicalModelCache`, notify the `RiskEstimationCombiner` of the different jobs that will be performed for a single risk estimation and schedule the individual jobs in its queue according to *P2*. For more details about the scheduling policy, see Section 4 and Section 5.3. The sensor data package `newSensorData` is passed because its arrival triggered the risk estimation.
    - Exceptions: None
  - `void launchRiskEstimation(PatientId patientId)`
    - Effect: The `RiskEstimationScheduler` will fetch the clinical models and their configurations associated to the patient identified by `patientId` from the `OtherDataDB` using the `ClinicalModelCache`, notify the `RiskEstimationCombiner` of the different jobs that will be performed for a single risk estimation and schedule the individual jobs in its queue according to *P2*. For more details about the scheduling policy, see Section 4 and Section 5.3.
    - 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

A.29 RiskEstimator

- **Description:** The `RiskEstimator` is responsible for estimating the risk level of a patient by evaluating clinical models. This evaluation is triggered every time new sensor data arrives, when the configuration of a clinical model is updated etc. As opposed to the emergency clinical models used in the `EmergencyHandler`, the clinical models used in the `RiskEstimator` are heavy-weight and can require large amounts of sensor data, plus other data of the patient such as his or her EHR. The `RiskEstimator` will store estimated risk levels in the `OtherDataDB`.
- **Super-component:** None
- **Sub-components:** `ClinicalModelCache`, `RiskEstimationCombiner`, `RiskEstimationProcessor` and `RiskEstimationScheduler`

Provided interfaces

- LaunchRiskEstimation
  - `void launchRiskEstimation(PatientId patientId, SensorDataPackage newSensorData, TimeStamp receivedAt)`
    - Effect: The `RiskEstimator` will fetch the clinical models and their configurations for the patient identified by `patientId`. A risk level is estimated based on the computation of these clinical models. If the result estimates a change in the patient's risk level notifications are send out to the patient's subscribers. For the computation of the clinical models, the `RiskEstimator` uses the given sensor data `newSensorData` (which is the new sensor data received by the `GatewayFacade`) and fetches other required data from the respective databases if needed. The given time-stamp `receivedAt` is used in order to avoid fetching the new sensor data from the database. This time-stamp is the time at which the `GatewayFacade` received the new sensor data.
    - Exceptions: None
  - `void launchRiskEstimation(PatientId patientId)`
    - Effect: The `RiskEstimator` will fetch the clinical models and their configurations for the patient identified by `patientId`. A risk level is estimated based on the computation of these clinical models. If the result estimates a change in the patient's risk level notifications are send out to the patient's subscribers.
    - Exceptions: None
- ClinicalModelCacheMgmt
  - `void invalidateClinicalModel(PatientId patientId)`
    - Effect: The `RiskEstimator` will invalidate (i.e., remove) all cached clinical models for the patient identified by `patientId`. If the cache does not contain any items for this patient, nothing is changed. After invalidating the cached items for a certain patient, the next request for them will lead to fetching them from the `OtherDataDB` and storing them in the cache again.
    - Exceptions: None

A.30 SensorDataCache

- **Description:** The `SensorDataCache` is responsible for temporarily storing new sensor data written to the `PrimaryDB`, in order to avoid losing the data written to the `PrimaryDB` after the last synchronization with the Standby. The `SensorDataCache` should be able to store sensor data for at least the synchronization period of the `StandbyDB` and regularly removes cached requests older than this period.
- **Super-component:** `SensorDataDB`
- **Sub-components:** None

Provided interfaces

- CacheMgmt
  - `void cacheNewSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: The `SensorDataCache` will store the given sensor data and associated meta-data.
    - Exceptions: None
  - `List<Tuple<PatientId,GatewayId,SensorDataPackage,TimeStamp>> flush()`
    - Effect: The `SensorDataCache` will return and remove all entries in the cache, sorted by their age, the eldest entry first.
    - Exceptions: None

A.31 SensorDataDB

- **Description:** The `SensorDataDB` is responsible for storing sensor data sent by the gateways. Internally, the `SensorDataDB` is replicated and prioritizes requests according to *P1*.
- **Super-component:** None
- **Sub-components:** `PrimaryDB`, `ReplicationManager`, `SensorDataCache`, `SensorDataScheduler` and `StandbyDB`.

Provided interfaces

- SensorDataMgmt:
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: The `SensorDataDB` will store the given sensor data `sensorData` together with the provided meta-data.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatient(PatientId patientId, SensorDataRequestSource source)`
    - Effect: The `SensorDataDB` will fetch and return all sensor data belonging to the patient identified by `patientId`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId patientId, TimeStamp stopTime, SensorDataRequestSource source)`
    - Effect: The `SensorDataDB` will fetch and return all sensor data belonging to the patient identified by `patientId` which was received before the given time `stopTime`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientAfterAndBefore(PatientId patientId, TimeStamp startTime, TimeStamp stopTime, SensorDataRequestSource source)`
    - Effect: The `SensorDataDB` will fetch and return all sensor data belonging to the patient identified by `patientId` which entered the system between `startTime` and `stopTime`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
  - `Tuple<TimeStamp, SensorDataPackage> getLatestSensorDataOfPatient(PatientId patientId, SensorDataRequestSource source)`
    - Effect: The `SensorDataDB` will fetch and return the latest sensor data of the patient identified by `patientId`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None

A.32 SensorDataScheduler

- **Description:** The `SensorDataScheduler` is responsible for scheduling read and write requests sent to the `SensorDataDB`. The employed scheduling policy depends on the modus of the system according to *P1* and *Av3*, i.e., normal modus, overload modus or degraded modus. For more details concerning the respective scheduling policies, see Section 4 and Section 5.4.
- **Super-component:** `SensorDataDB`
- **Sub-components:** None

Provided interfaces

- SensorDataMgmt:
  - `void addSensorData(PatientId patientId, GatewayId gatewayId, SensorDataPackage sensorData, TimeStamp receivedAt)`
    - Effect: The `SensorDataScheduler` will store the given sensor data `sensorData` together with the provided meta-data using the `ReplicationManager`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatient(PatientId patientId, SensorDataRequestSource source)`
    - Effect: The `SensorDataScheduler` will fetch and return all sensor data belonging to the patient identified by `patientId`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientBefore(PatientId patientId, TimeStamp stopTime, SensorDataRequestSource source)`
    - Effect: The `SensorDataScheduler` will fetch and return all sensor data belonging to the patient identified by `patientId` which was received before the given time `stopTime`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
  - `Map<TimeStamp, SensorDataPackage> getAllSensorDataOfPatientAfterAndBefore(PatientId patientId, TimeStamp startTime, TimeStamp stopTime, SensorDataRequestSource source)`
    - Effect: The `SensorDataScheduler` will fetch and return all sensor data belonging to the patient identified by `patientId` which entered the system between `startTime` and `stopTime`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
  - `Tuple<TimeStamp, SensorDataPackage> getLatestSensorDataOfPatient(PatientId patientId, SensorDataRequestSource source)`
    - Effect: The `SensorDataScheduler` will fetch and return the latest sensor data of the patient identified by `patientId`. The scheduling priority of this sensor data read request depends on the given `source`.
    - Exceptions: None
- SetDegradedModus
  - `void setDegradedModus()`
    - Effect: The `SensorDataScheduler` will switch to degraded modus for scheduling incoming read and write requests.
    - Exceptions: None
  - `void setNondegradedModus()`
    - Effect: The `SensorDataScheduler` will switch out of degraded modus scheduling incoming read and writes requests. Note that this does not mean that the `SensorDataScheduler` switches to normal modus, since it can still be in overload modus.
    - Exceptions: None

A.33 SessionDB

- **Description:** Stores the session identifiers for currently active sessions.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- SessionMgmt
  - `UserId getUserId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` will fetch and return the user identifier corresponding to the session identified by `sessionId`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier.
  - `SessionId openSession(UserId userId)`
    - Effect: Generates a new session identifier for the given `userId` and stores this as an active session.
    - Exceptions: None
  - `void closeSession(SessionId sessionId) throws NoSuchSessionException`
    - Effect: Closes the active session associated with the given `sessionId`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier.
  - `Map<SessionAttributeKey, SessionAttributeValue> isValidSession(SessionId sessionId) throws NoSuchSessionException`
    - Effect: Returns the session attributes if the given id corresponds to a valid session.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exitsts with the given identifier.

A.34 StandbyDB

- **Description:** The `StandbyDB` is responsible for storing a back-up of the data in the `PrimaryDB`. The `StandbyDB` does not respond to any read or write requests but periodically receives sensor data from the `PrimaryDB` to synchronize its state.
- **Super-component:** `SensorDataDB`
- **Sub-components:** None

Provided interfaces

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

A.35 SystemAdminDevice

- **Description:** The `SystemAdminDevice` is external to the PMS and represents the device of the PMS system administrators through the PMS contacts them.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- DeliverNotification
  - `void deliverNotification(NotificationMessage msg)`
    - Effect: Deliver the notification message `msg` to the system administrator..
    - Exceptions: None

B Defined data types

In this section, we list and define the data types used in the interfaces.

- `Answer`: The answer to a question in a questionnaire. This can, for example, be an answer in natural language or a number indicating the place on a scale.
- `ClinicalModelDescription`: The description of a clinical model. This description at least contains the id of the clinical model, a human-readable name and a human-readable textual description.
- `ClinicalModelId`: A piece of data uniquely identifying a certain clinical model in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `ClinicalModelJob`: A single job for the computation of a clinical model. This contains a `ClinicalModelJobId`, `ClinicalModelId` and `PatientId` respectively identifying the job itself, the corresponding clinical model and patient. If the corresponding risk estimation the job belongs to is triggered by the arrival of a sensor data packages this sensor data is also contained.
- `ClinicalModelJobId`: A piece of data uniquely identifying a certain clinical model job in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `ClinicalModelParameterKey`: The key of a parameter of the configuration of a clinical model. This architecture does not specify the exact format or type of this key, but a possible option is a flat string.
- `ClinicalModelParameterValue`: The value of a parameter of the configuration of a clinical model. This value can be of any primitive type.
- `ClinicalModelJobResult`: The result of the computation of a clinical model containing the estimated risk level for the patient.
- `Credentials`: The authentication credentials of a user not employed at the hospital. The credentials always contain the identifier of the user and a proof of his or her identity. As stated in the domain description, the type of this proof can differ for different types of users, e.g., a patient uses username and password while a telemedicine operator can use a smartcard.
- `Destination`: The destination of a notification message sent to a non-hospital user. This architecture does not specify the exact format of such a message and as such, the exact format of the destination is not specified either. Plausible options are a mobile telephone number for SMS or an e-mail address for e-mail.
- `DiagnosticTestResults`: The results of the diagnostic tests for a certain gateway.
- `Echo`: The response to a ping message. This data element does not contain any meaningful data.
- `GatewayId`: A piece of data uniquely identifying a gateway device in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `GatewayInitializationResults`: The results of the initialization of gateway when first linking this to a new patient. These results contain the results of the gateway's diagnostic tests and the first sensor data fetched from the gateway.
- `HISPatientId`: A piece of data uniquely identifying a patient in the `HIS`. The exact format of this identifier depends on the implementation of this `HIS`, but possibilities are a long integer, a string, a URL etc.
- `HISPatientDescription`: The short description of a patient in this `HIS`. This description contains at least the patient's name and the `HIS` identifier.

- `InvitationId`: A piece of data uniquely identifying a trustee invitation. A trustee invitation is sent to a new trustee when a user appoints a trustee for a patient.
- `LogMessage`: The textual message of a log.
- `LogSource`: The source of a log message which is incorporated in the complete log message. The source is a combination of the component which sent the message, the node on which this component was deployed and the location in the code (file name and line number, for example).
- `NotificationMessage`: A textual message which can be used to include extra information about the event of the notification.
- `PatientDetails`: A complex data structure listing the details of the patient, such as his or her first name, last name, age, address, current location etc.
- `PatientId`: A piece of data uniquely identifying a patient in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `PatientRecord`: The structured contents of the EHR record of a certain patient, such as medication history, recent treatments, allergies, etc.
- `PatientStatus`: The status of a patient, i.e., his or her risk level.
- `PatientStatusOverview`: The overview of the status of a patient that can be shown to a human user. This overview can differ depending on the type of user (e.g., physician vs patient). This architecture does not specify the exact format of such an overview, but a likely possibility is an HTML page.
- `Question`: The textual representation of a question in a questionnaire, plus the preferred data type of the answer or possible options if required.
- `Questionnaire`: The representation of a questionnaire, i.e., the list of questions in the questionnaire, and meta-data such as by whom it was created, when it was created, whether it was already filled out, when it was filled out etc. If the questionnaire was already filled out, this representation also contains the provided answers.
- `QuestionnaireDescription`: The short description of a questionnaire in the system. This description consists of the basic information of a questionnaire such as by whom it was created, when it was created, whether it was already filled out, when it was filled out etc.
- `QuestionnaireId`: A piece of data uniquely identifying a `Questionnaire` in the PMS. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `RiskEstimationId`: A piece of data uniquely identifying a risk estimation performed by the PMS. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `Role`: The role of a user in the system. Example of such a role are “cardiologist”, “patient”, “GP”, “trustee” etc.
- `SensorDataPackage`: A package of sensor data, i.e., a list of sensor data packages per sensor. Each sub-package lists the id of the sensor, the type of the sensor, the type of the measurements and the measurements itself. These measurements range from a single value (e.g., in case of the blood pressure) to a complex data structure (e.g., in case of an ECG).
- `SensorDataRequestSource`: The source of a request for sensor data sent to the `SensorDataDB`. This source is used to schedule the requests. For example, a request by an end-user facade will have lower priority with respect to a request sent by a risk estimator.

- `SessionAttributeKey`: The key of an attribute attached to a session. This architecture does not specify the exact format of this key, but a possible value is a flat string.
- `SessionAttributeValue`: The value of an attribute attached to a session. This value can be of any primitive type.
- `SessionId`: A piece of data uniquely identifying a user session in the PMS. This contains at least the user identifier and the time the session was initiated.
- `StandbyAddress`: The address at which the `StandbyDB` can reach the `PrimaryDB` for synchronization. This architecture does not specify the exact format of such an address, but likely possibilities are an IP address and port, or a URL.
- `TimeStamp`: The representation of a time (i.e., date and time of the day) in the system.
- `TrusteeContactDetails`: The contact information of a new trustee. This contact information at least contains the new trustee's e-mail address.
- `UserId`: A piece of data uniquely identifying a user in the `HIS`. For patients, this identifier equals their `PatientId`. The exact format of this identifier depends on the implementation of this `HIS`, but possibilities are a long integer, a string, a URL etc.