Appendix B.
Initial eDocs architecture

March 27, 2015

Contents

This document presents an initial architecture of the document processing system, in which decompositions have already been executed for *Av1*, *P1* and *Av2*. Section 1 starts by listing the key architectural decisions for these requirements. Sections 2, 3, 4 and 5 illustrate the architecture that results from these decisions. Finally, Section 6 lists the residual drivers after these initial decompositions.

1 Key decisions and rationale

1.1 *Av1* (Document generation failure)

Remark: Notice that *Av1* and *P1* both apply to the same sub-system. As a result, they were taken into account together and the architectural decisions to achieve them are closely related.

Architectural decisions. The elements of *Av1* are tackled in the following manner:

1. The requirement that the system should "*be able to autonomously detect*" failure of the document generation infrastructure, gives rise to the `DocumentGenerationManager` and `Generator` components in the client-server view (see Section 2). The `Generator` generates the actual documents and forwards them to `OtherFunctionality` to store and deliver them. The `DocumentGenerationManager` **monitors** the availability of the `Generator` using **ping/echo** (every 4 seconds).

2. In order to make sure that a document is generated for every entry in the raw data in the presence of failures, the system should also be able to **restart document generation jobs** that did not complete correctly because of a failure. Therefore, the `DocumentGenerationManager` also keeps track of the jobs assigned to and being processed by the `Generator` and the `Generator` reports back completed jobs to the `DocumentGenerationManager` so that it can restart jobs upon failure. In order to minimize the overhead of this coordination, the `DocumentGenerationManager` assigns jobs to the `Generator` in groups of more than one job that are part of the same batch and the `Generator` reports back on the completion of such a group (this is a performance tactic related to *P1*).

3. The responsibility of restarting jobs "*so that any deadlines are still met*" or "*on an earliest-deadline-first basis*" if certain deadlines cannot be met any more is also assigned to the `DocumentGenerationManager`, which **prioritizes jobs based on their deadlines** and **schedules** them for execution at least 2 hours before their deadline so that restarting them is unlikely to affect meeting their deadlines. Notice that this functionality relates to *P1* (see Section 1.2).

4. In order to accomplish that failure of the infrastructure for generating documents "*does not affect the availability of any type of persistent data*" and "*does not affect the availability of other functionality of the system*", document generation is **separated from any database and the other functionality** of the system. More precisely, the `DocumentGenerationManager` and `Generator` are only responsible for generating documents and do not store any persistent data apart from state internal to the document generation process. Moreover, these components (or any of their sub-components) are deployed on nodes that do not host database components (see the deployment view in Section 4).

5. As described above, the `Generator` reports completion to the `DocumentGenerationManager` after it has forwarded the generated document to `OtherFunctionality`. If the `Generator` fails after forwarding the document and before reporting completion to the `DocumentGenerationManager`, it can be that the `DocumentGenerationManager` restarts jobs for which a document has already been stored and delivered. Therefore, in order to make sure that no duplicates are stored or delivered ("*exactly one instance of the correct document should eventually be generated and sent for each job*"), the methods in the `FinalizeDocument` interface are made **idempotent**, i.e., these methods filter duplicates based on the `JobId`.

6. In order to achieve that the infrastructure for generating documents has a "*guaranteed minimal uptime*" of 99.9%, the `Generator` component is **replicated** on multiple nodes, which is visible in the deployment view (see Section 4). The replication factor will be determined later, based on the frequency of failures in practice and the scale on which the system will be deployed (cf. *P1*). The `DocumentGenerationManager` is responsible for **monitoring** the availability of all `Generator` instances.

7. The requirement that "*the system notifies the eDocs Operators*" is in fact a new functional requirement (a use case called "*Notify eDocs operators*", highly similar to *UC22*). This is delegated as residual driver *Av1a* to `OtherFunctionality`.

8. The requirement to "*notify the operators within 1 minute*", is the joint responsibility of the `DocumentGenerationManager` and `OtherFunctionality`: the `DocumentGenerationManager` is responsible for immediately contacting `OtherFunctionality` when noticing a failure, and `OtherFunctionality` is made responsible for forwarding this notification to an operator within 1 minute using residual driver *Av1a*.

9. In order to "*show the status of the document processing jobs affected by this problem correspondingly*", the `DocumentGenerationManager` stores the status of the jobs that were assigned to a failed `Generator` instance as "temporarily failed" via `OtherFunctionality` using the `SetStatus` interface. The responsibility of actually storing the status of an individual job is delegated to the `OtherFunctionality` component using residual driver *Av1b*.

1.2 *P1* (Document generation)

Architectural decisions. The elements of *P1* are tackled in the following way:

1. The requirement that the system is "*able to perform multiple document processing jobs in parallel*" is achieved by **replicating** the `Generator` as discussed for Av1. The `Generator` is deployed on multiple nodes and multiple times on each node for parallelism and employing each node optimally (see the deployment view in Section 4).

2. In order to "*dynamically adjust the number of parallel document processing jobs depending on the amount of documents that should be generated in the near future so that their deadlines are met*", the `DocumentGenerationManager` **plans** and decides which jobs should be processed in the near future **based on the deadlines of all jobs** that should still be processed and **starts up** new `Generator` instances if needed.

3. In order to achieve that "*the cost of the system remains as low as possible*", the `DocumentGenerationManager` also scales down the document generation sub-system by **shutting down** some of the parallel instances of the `Generator` if the projected near-future load allows it. The `Generator` instances that have received the signal to shut down will complete their assigned group of document generation jobs and report back completion to the `DocumentGenerationManager` before actually shutting down.

4. In the decomposition of the `DocumentGenerationManager` (see Section 3.1), the responsibilities of the `DocumentGenerationManager` are divided over multiple sub-components:

   (a) The `Scheduler` receives the new jobs initiated by a customer organization and adds them to a queue of all jobs that have not been processed yet. To lower the size of this queue, the `Scheduler` is only given the information it needs, i.e., the id of the batch, its deadline and the ids of the individual jobs. The raw data of each job and the meta-data of the batch is stored in `OtherFunctionality` and fetched by the `Completer` when needed.

      i. In order to be able to respond swiftly to critical jobs, `OtherFunctionality` provides new jobs synchronously to the `Scheduler`, and the `Scheduler` also plans in the new jobs synchronously.

   (b) The `GeneratorManager` is responsible for monitoring the `Generator` instances, starting up or shutting down such instances based on the number of required instances indicated by the `Scheduler`, keeping track of the jobs assigned to each `Generator` instance and restarting jobs if needed. At regular intervals (i.e., every 10 minutes), the `GeneratorManager` queries the `Scheduler` for the amount of documents that should be generated in the near future and adjusts the number of `Generator` instances accordingly. When a `Generator` instance requires a new group of jobs, the `GeneratorManager` asks the `Scheduler` for the ids of the group of jobs that belong to the same batch that should be generated next. The `GeneratorManager` then asks the `Completer` for the raw data and meta-data (e.g., type of document, the template to be used and the key to be used) for these jobs.

(c) The `Completer` is responsible for fetching the raw data and applicable meta-data for a group of job ids from `OtherFunctionality` so that the `GeneratorManager` can assign these jobs to a `Generator` instance.

(d) The `Completer` has to fetch the template and key every time a `Generator` instance requests new jobs, while these will be the same for all jobs belonging to the same batch. To avoid making the template and key storage systems the bottleneck for document generations, the `DocumentGenerationManager` caches templates and keys. The keys are cached based on the `CustomerId`, the template cache also takes into account the type of the document, and the date and time at which a batch was provided by the customer organization in order to account for template updates. This functionality is respectively assigned to a `KeyCache` and `TemplateCache` component.

5. In order to "*be able to adjust the capacity for processing documents in less than 5 hours*", the system is deployed on an infrastructure that allows to start up (virtual) machines at run-time within minutes and without human intervention. This infrastructure is dimensioned to cope with a realistic estimation of the highest possible load (e.g., the maximal load based of the recurring batches * 2).

1.3 *Av2* (Personal document storage failure)

Architectural decisions. The elements of *Av2* are tackled in the following manner:

1. In order to achieve that failure of the database of documents in the personal document stores "*does not affect the availability of other types of persistent data*", the documents in personal document stores are **separated from all other persistent data**. This gives rise to the `PDSDB` component (short for Personal Document Store Database) in the client-server view (see Section 2).

2. In order to achieve a *"guaranteed minimal uptime"* of the documents in the personal document store and to ensure that a single failure "*does not lead to loss of documents*", these documents are **actively replicated**. This gives rise to the `PDSDBReplica` component that actually stores the documents and the `PDSReplication-Manager` component that manages the replication (i.e., writing to all `PDSDBReplica` instances, reading from one, and monitoring their availability). The multiple instances of the `PDSDBReplica` are **deployed on individual nodes**, which is visible in the deployment view (see Section 4).

3. In order to differentiate between documents that require different uptime percentages, these documents are stored in **different storage clusters**. These clusters each consist of a `PDSReplicationManager` and one or more `PDSDBReplica` instances so that the number of `PDSDBReplica` instances can be tailored to the uptimes required for each document type. The system employs two such clusters: one for documents that have been generated less then 30 days ago or have not yet been received by their recipient, and one for documents that have been received by their recipient and were generated over 30 days ago. The `PDSLong-TermDocumentManager` is introduced to manage both clusters, i.e., storing new documents in the former, reading documents from both clusters, and periodically (i.e., once a day) transferring old received documents from the one system to the other. Notice that in order to avoid that a failure can result into loss of documents, every document should have at least 1 back-up, meaning that both clusters should at least consist of 2 replicas. The deployment view in Section 4 illustrates this using respectively 3 and 2 replicas.

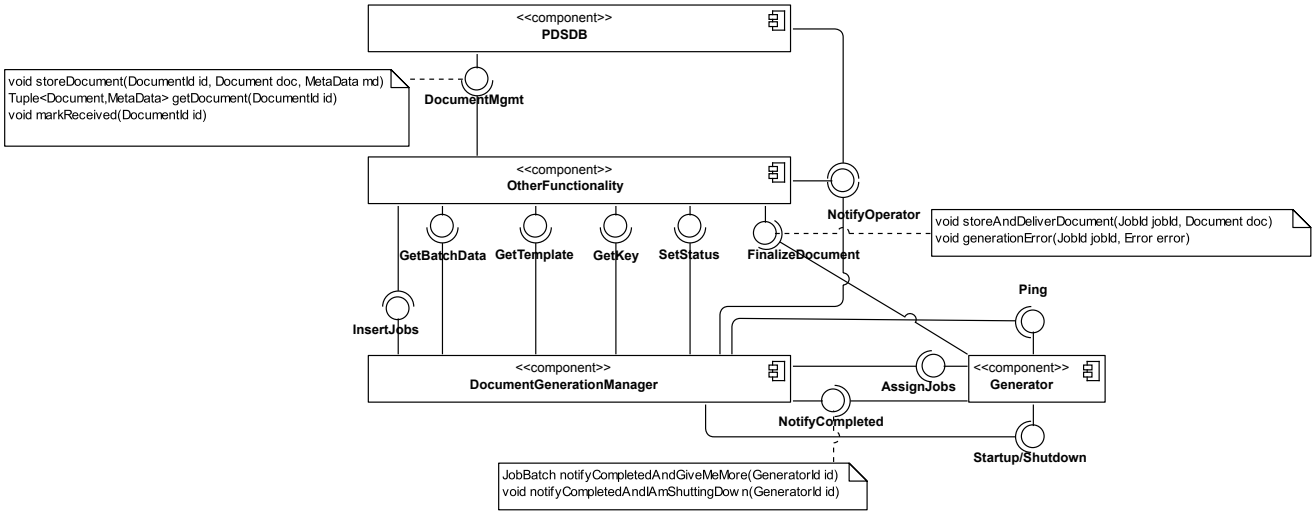
4. In order to be able to autonomously detect a failure of a `PDSDBReplica` instance, the `PDSReplication-Manager` **monitors** their availability using **ping/echo** (every 4 seconds). Queries and their responses are used as implicit ping/echos and an explicit ping is sent only when needed.

5. In order to notify the eDocs Operators of a failed `PDSDBReplica` instance, the `PDSReplicationManager` employs the `NotifyOperator` interface of `OtherFunctionality` introduced for *Av1*. This interface should be able to accept notifications from both the `PDSReplicationManager` and the `DocumentGenerationManager`.

6. Similarly to *Av1*, the requirement to "*notify the operators within 1 minute*" is the joint responsibility of the `PDSReplicationManager` and `OtherFunctionality`: the `PDSReplicationManager` is responsible for immediately contacting `OtherFunctionality` when noticing a failure, and `OtherFunctionality` is made responsible for forwarding this notification to an operator within 1 minute using residual driver *Av2a*.

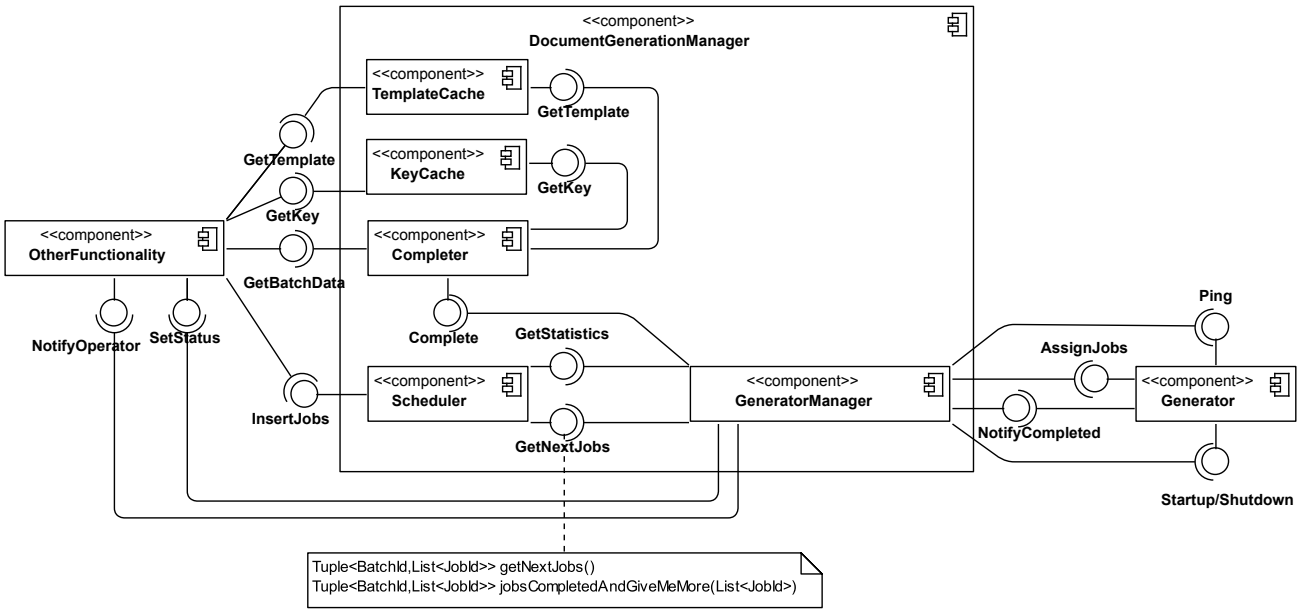
7. The responsibilities to "*temporarily store documents that should be delivered via the personal document store*" for at least 3 hours in case the `PDSDB` is unavailable and provide a clear message to users in order to "*fail gracefully*" are delegated to `OtherFunctionality` using residual driver *Av2b*.

2 Client-server view (UML Component diagram)

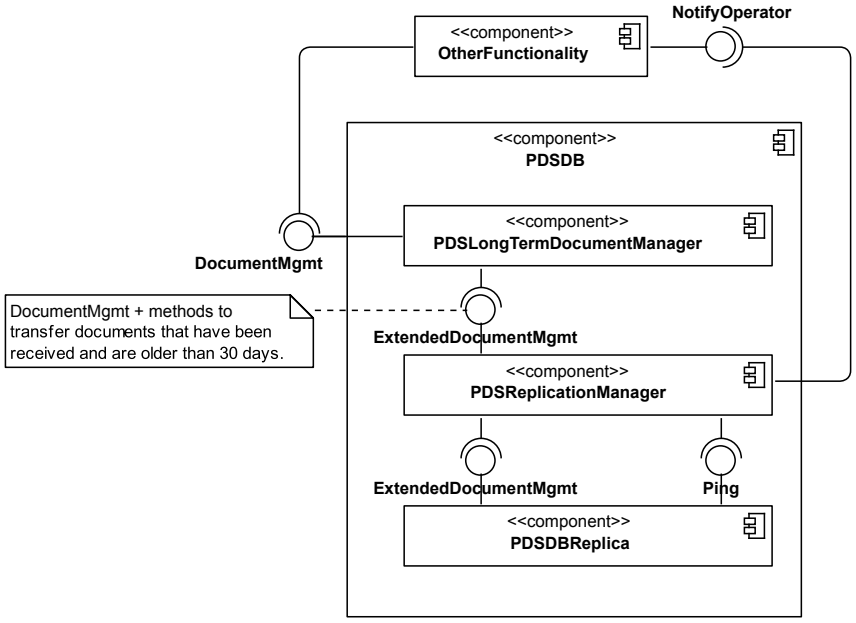


3 Decomposition view (UML Component diagram)

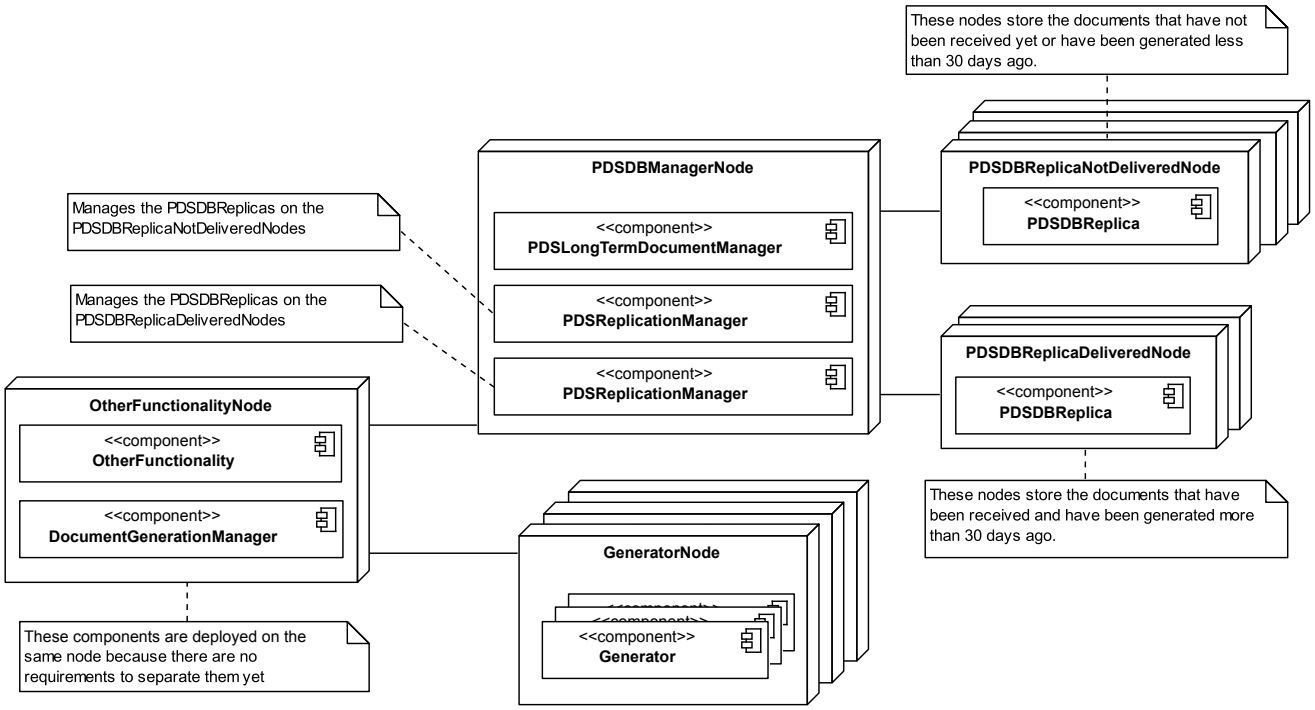
3.1 `DocumentGenerationManager` component



3.2 PDSDB component

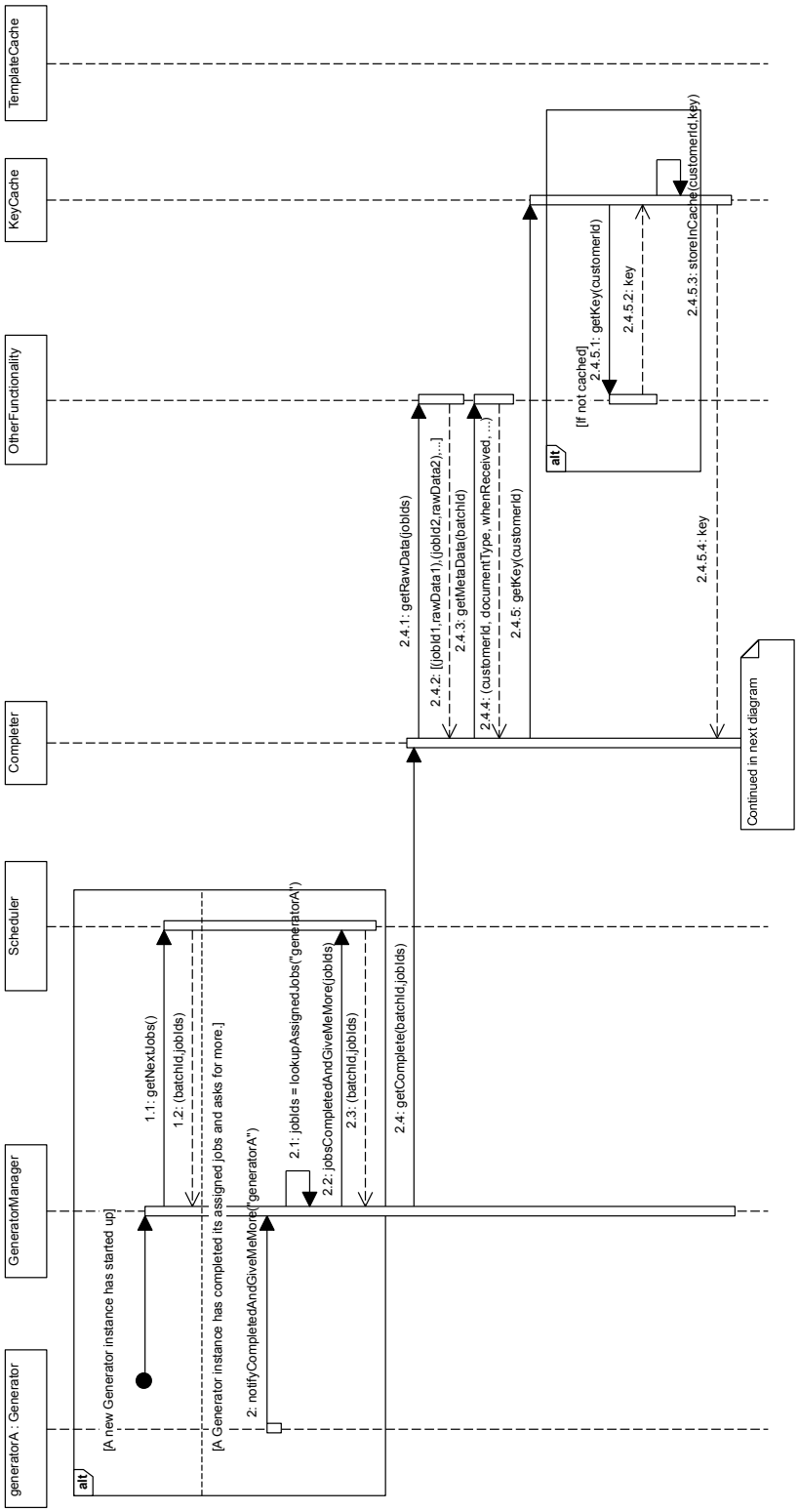


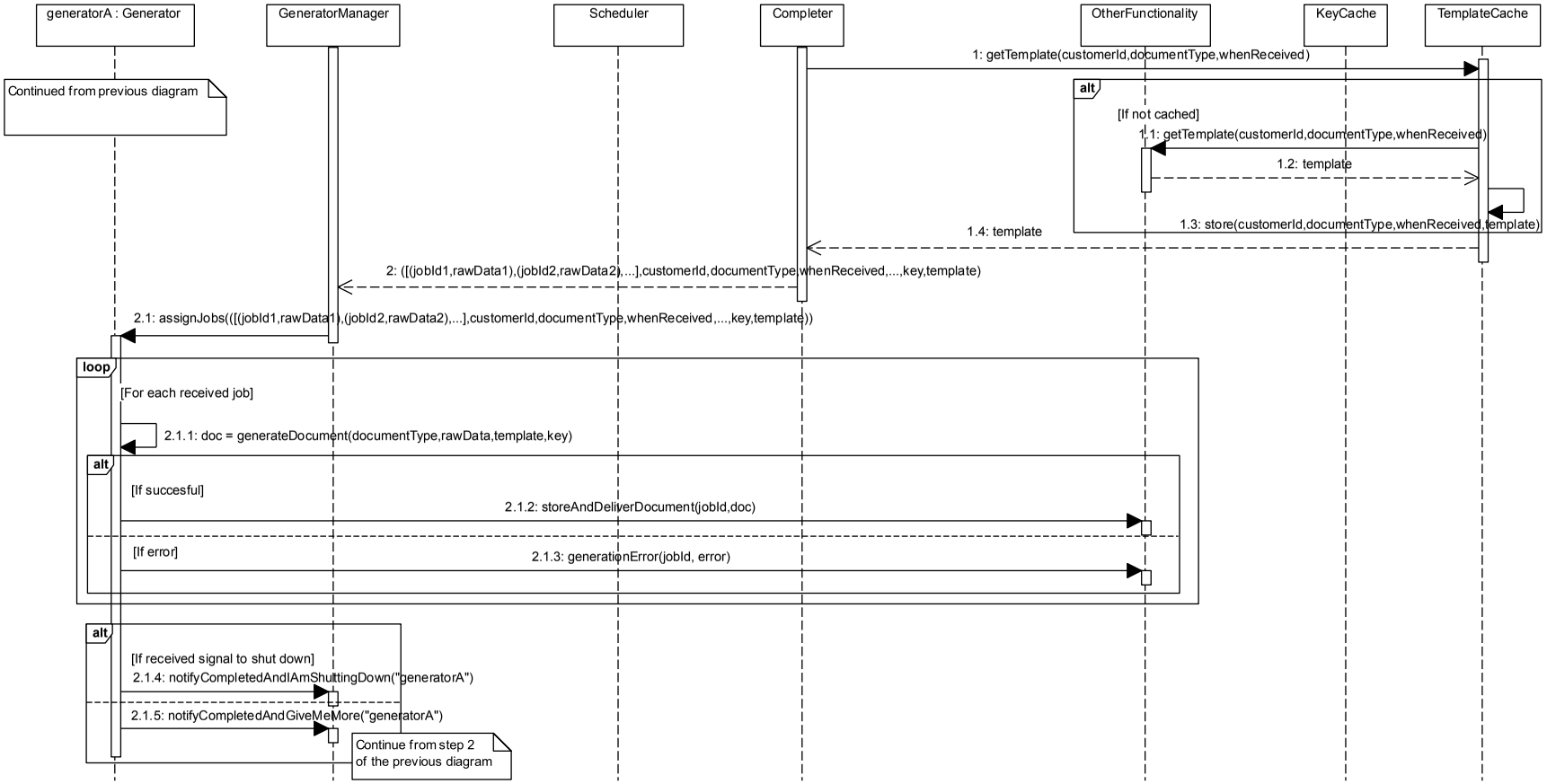
4 Deployment view (UML Deployment diagram)



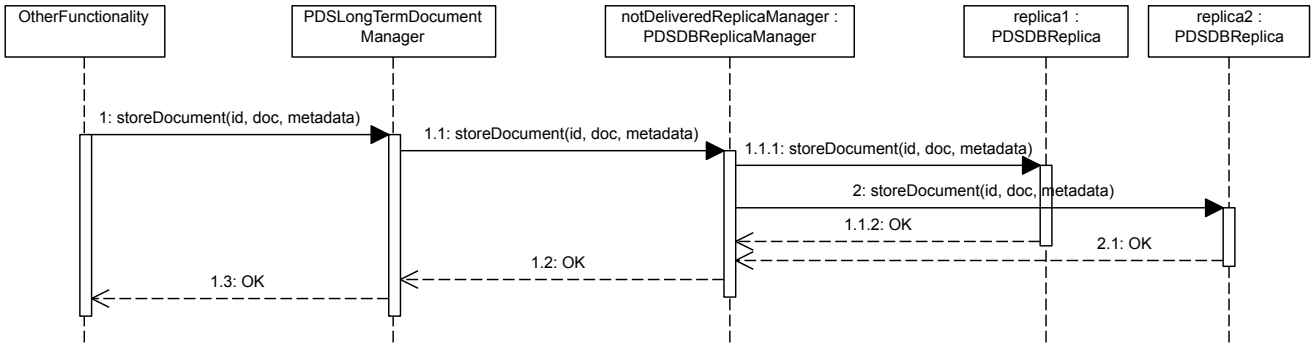
5 Process view (UML Sequence diagram)

5.1 Document generation

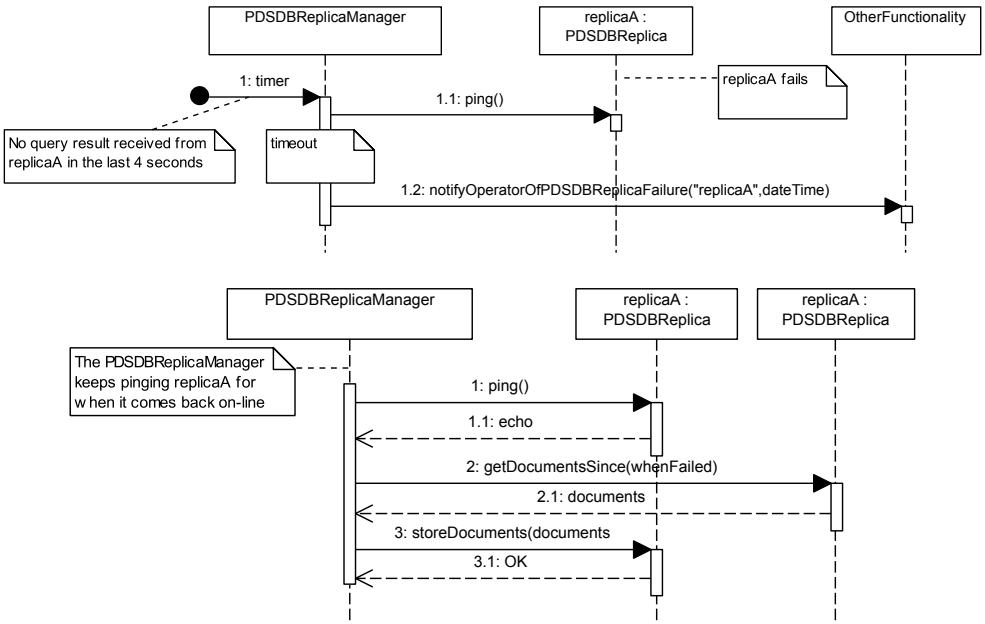



generatorA : Generator
GeneratorManager
Scheduler
Completer
OtherFunctionality
KeyCache
TemplateCache
1: getTemplate(customerId,documentType,whenReceived)
Continued from previous diagram
alt
[If not cached]
1.1: getTemplate(customerId,documentType,whenReceived)
1.2: template
1.3: store(customerId,documentType,whenReceived,template)
1.4: template
2: ([(jobId1,rawData1),(jobId2,rawData2),...],customerId,documentType,whenReceived,...,key,template)
2.1: assignJobs(([(jobId1,rawData1),(jobId2,rawData2),...],customerId,documentType,whenReceived,...,key,template))
loop
[For each received job]
2.1.1: doc = generateDocument(documentType,rawData,template,key)
alt
[If succesful]
2.1.2: storeAndDeliverDocument(jobId,doc)
[If error]
2.1.3: generationError(jobId, error)
alt
[If received signal to shut down]
2.1.4: notifyCompletedAndIAmShuttingDown("generatorA")
2.1.5: notifyCompletedAndGiveMeMore("generatorA")
Continue from step 2
of the previous diagram

5.2 Storing a document in the personal document store



5.3 Failure and recovery of a `PDSDBReplica` instance



6 Residual drivers

All drivers not explicitly mentioned in this report have not been addressed yet. Of the use cases, the following elements have been tackled:

- Of *UC3: Initiate document processing*, step 10 (starting new processing jobs) has been covered.
- Of *UC4: Generate payslip*, step 1 (generating the payslip) has been covered.
- Of *UC5: Generate invoice*, step 1 (generating the invoice) has been covered.
- Of *UC11: Deliver document via personal document store*, step 1 has been covered (adding the document to the personal document store of the Registered Recipient).

The following requirements were discussed, but are delegated to `OtherFuncationality`:

- New use case UC22': Notify eDocs operator
- Av1a: notifying the appropriate operator within 1 minute

- Av1b: storing the status of an individual job
- Av2a: notifying the appropriate operator within 1 minute
- Av2b: temporarily storing documents that should be delivered via the personal document store for at least 3 hours in case the PDSDB is unavailable and providing a clear message to users in order to fail gracefully