Katholieke
Universiteit
Leuven

**Department of
Computer Science**

DOCUMENT PROCESSING

The complete architecture

Software Architecture (H09B5a and H07Z9a) – Part 2b

Jeroen Reinenbergh (r0460600)
Jonas Schouterden (r0260385)

Academic year 2014–2015

Contents

1 Introduction

The goal of this project was to develop an architecture for a system for document processing. This part of the project consisted of

2 Overview

2.1 Architectural decisions

Briefly discuss your architectural decisions for each non-functional requirement. Pay attention to the solutions that you employed (in your own terms or using tactics and/or patterns).

ReqX: requirement name Provide a brief discussion of the decisions related to *ReqX*.
Employed tactics and patterns: List all patterns and tactics used to achieve ReqX, if any.

2.2 Discussion

Use this section to discuss your architecture in retrospect. For example, what are the strong points of your architecture? What are the weak points? Is there anything you would have done otherwise with your current experience? Are there any remarks about the architecture that you would give to your customers? Etc.

3 Attribute-driven design documentation

3.1 Decomposition 1: eDocs (X1, Y3, UCa, UCb, UCc)

3.1.1 Module to decompose

In the first run, the eDocs System is decomposed as a whole

3.1.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *X1*: name
- *Y3*: name

The related functional drivers are:

- *UCa*: name
- *UCb*: name
- *UCc*: name

Rationale A short discussion of why these drivers were selected for this decomposition.

3.1.3 Architectural design

Topic Discussion of the solution selected for (a part of) one of the architectural drivers.

Alternatives considered

Alternatives for solution A discussion of the alternative solutions and why that were not selected.

3.1.4 Instantiation and allocation of functionality

Decomposition Main aspects of the resulting decomposition.

ModuleB Per introduced component a paragraph describing its responsibilities.

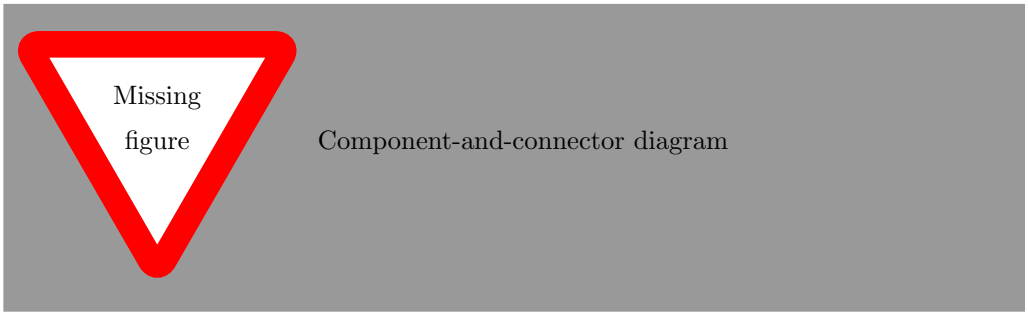


Figure 1: Component-and-connector diagram of this decomposition.

ModuleC Per introduced component a paragraph describing its responsibilities.

Behaviour If needed and explanation of the behaviour of certain aspects of the design so far.

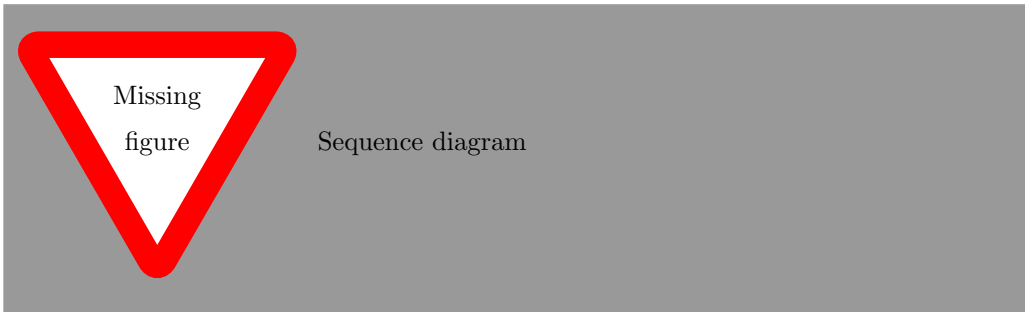


Figure 2: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

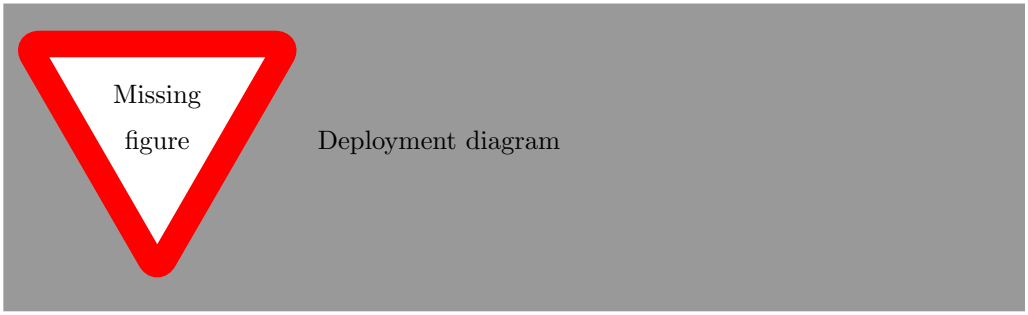


Figure 3: Deployment diagram of this decomposition.

3.1.5 Interfaces for child modules

PDSDB

- DocumentMgmt
  - `void storeDocument(DocumentId id, Document doc, MetaData md)`
    - Effect: The `PDSDB` will store the given document`doc` together with the provided metadata `md`.
    - Exceptions: None
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: Describe the effect of calling this operation.

    - Exceptions: None
- `void markReceived(DocumentId id)`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None

3.1.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

returnType This data element represents X.

ParamType This data element represents Y.

3.1.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

ModuleB

- *Z1*: name
- *UCd*: name

ModuleC

- *UCba*: name
  Description which part of the original use case is the responsibility of this component.

3.2 Decomposition 2: OtherFunctionality(P2, UC12, UC13, UC14, UC15)

3.2.1 Module to decompose

In this run we decompose `Otherfunctionality`.

3.2.2 Selected architectural drivers

The non-functional driver for this decomposition is:

- *P2*: Document lookups

The related functional drivers are:

- *UC12*: Consult personal document store
- *UC13*: Search documents in personal document store
- *UC14*: Consult document in personal document store
- *UC15*: Download document via unique link

Rationale P2 was chosen because it has one of the highest priorities among all remaining non-functional drivers and the domain on which its focus lies complements the previous decomposition perfectly.

3.2.3 Architectural design

Extended functionality of PDSLongTermDocumentManager for P2 Ik denk dat we de facade beter laten throttlen Jonas, want die heeft een overzicht van alle requests en weet of dit individuele of bulk (search) requests zijn. Zo zit je niet met users die een halve lijst terugkrijgen.

Link mapping aparte linkdatabase voor P, want "should not affect the performance of other functionality of the system"

Separate document and link mapping database for P2 To ensure that documents can be looked up via the personal document store or a notification in a timely fashion, we chose to store documents and link mappings (discussed below) in separate database components that are deployed on different machines. This decision prohibits either of those two to be a bottleneck in the document lookup process.

Sharding for document database for P2 Note that there must be a (sub)component monitoring the requests to the different shards, to cap the number of requests. We want to have the response time of active replication. But actively replicating the whole database might have too high a cost, so we choose sharding. This keeps the fast response time, but has less hardware required.

Pingen is nodig voor write bij sharding.

Document storage manager DocumentStorageManager nodig want twee opslagkanalen, de PDSDB en de DocumentDB

extra methode in PDSDBMngmt voor de opslag

DocumentStorageManager is toegevoegd om de nieuwe componenten te koppelen met de vorige decompositie. De component synchroniseert de DocumentDB met de PDSDB. Hiermee wordt bedoeld dat documenten bedoeld voor Registered Recipients in beide databases worden opgeslagen.

UserFacade for P2 we moeten de duplicatie van UserFacade bespreken voor P 2. De userfacade is toegevoegd om onderscheid te maken tussen Registered Recipients en Unregistered Recipients.

De userfacade markeert ook documenten als received. De PDSFacade en LinkManager DOEN DIT NIET. Dit zowel voor documenten die opgevraagd worden met behulp van een unieke link in een email (voor registered recipients EN voor unregistered recipients) als voor die via de PDS. De reden dat we dat op deze plaats doen, en niet bij PDSFacade of LinkManager, is omdat de userfacade de laatste compomnent is waar het document voorbij komt voor het bij de recipient terecht komt. Als we dit in een eerdere component zouden doen, is het mogelijk dat een component dichter bij de recipient faalt zonder dat de gebruiker het document ontvangt en zonder dat dit opgemerkt wordt.

Motivatie voor AuthN: je moet ingelogd zijn voor o.a. use case 12. Motivatie voor UniqueLinkMgmt: het opvragen van documenten met behulp van een unieke link. Motivatie voor PDSDBMgmt: use case 12: the registered recipient indicates that he or she wants to consult his or her personal document store.

PDSFacade door een aparte component moet de pdsdb zich niet bezighouden met het aggregeren van tussenresultaten bij het verwerken van queries. De PDSFacade wordt niet gerepliceerd, aangezien de PDSDBLongTermStorageManager ook niet gerepliceerd is, en op zich een bottleneck is. De facade is enkel voor de reads, bij het opslaan van documenten gaat de DocumentStorageManager rechtstreek naar de PDSDB. De PDSFacade verzamelt altijd eerst een overview van all the recipients documents DESCRIPTIONS. Subsequently, queries can be performed on this collection. This approach introduces no significant overhead. Bij het opvragen van een document door de userfacade, worden een userid en een documentid naar de PDSFacade doorgestuurd. de PDSFacade gaat eerst de metadata horende bij de docid opvragen, checkt of de userid in de metadata overeen komt met userid. Als dat is wordt vervolgens het document opgevraagd. Als dat niet zo is, wordt een exception gegooid–¿ bij te schrijven in de element catalog

LinkManager VOORLOPIG NIET NODIG IN DEZE DECOMPOSITIE

Motivatie voor LinkManager: Checks expiration date ALS DAT NODIG IS–¿ reason: links naar de pdsdb vervallen niet (zolang de gebruik geregistreerd is) LinkManager maps link to (document ID, place where the document is stored)-pairs –¿ REASON: the unique link has two possible sources: an e-mail to an unregistered recipient or an email to a registered recipient. For an unregistered recipient, the RecipientFacade must look with the documentid for the document in the documentDB. For a registered recipient, the RecipientFacade must look with the documentid for the document in the PDSDB. (Mogelijk een boolean ofzo)

Does NOT do mapping removal after x years –¿ there has to be a notification when the link has expired

Residual Av2b1 - DocumentStorageCache QAS AV2b: hier voldaan, want de documenten worden zowel in de documentdb als in de pdsdb opgeslagen, dus de documenten worden wel nog opgeslagen mocht de pdsdb uitvallen. Een clear message aan de recipient wordt gegeven met behulp van de NotifyRecipient interface aan otherfunctionality2. Residual Av2b2 : TIJDIGE notification AAN DE USER

We hebben twee alternatieve manieren besproken om de drie uur aan documenten op te vangen die mogelijk verloren gaan bij het uitvallen van de PDSDB. De eerste maakt gebruikt van een cache. Die cache slaat de documentids op die de laatste drie uur gegenereerd zijn. Aangezien de documenten die in de PDSDB horen opgeslagen te worden ook in de documentDB zitten, kunnen dan de documenten van de laatste drie uur door de DocumentStorageManager uit de DocumentDB opgevraagd worden en opgeslagen worden in de opnieuw online gekomen PDSDB (de recipient krijgt dus pas opnieuw toegang tot de PDS na de downtime + overheveltijd, wat langer dan 3u kan zijn door het kopieren). Merk op dat in de docDB zowel de docID (DB key), het doc als de meta data (inclusief recipient ID) opgeslagen wordt, zodat deze info onmiddellijk beschikbaar is bij het overhevelen van docs naar de PDS (dataformaat in PDS en docDB is dus identiek). Een nadeel aan deze methode is wel dat wanneer de PDSDB langer dan drie uur offline is, er documenten in de DocumentDB opgeslagen zitten die niet in de PDSDB gestoken worden wanneer die terug online komt. Deze documenten zijn niet meet opvraagbaar -¿ out of scope Het alternatief is dat wanneer de PDSDB terug online komt, de DocumentStorageManager vergelijkt welke documenten er voor de Registered Recipients in de PDSDB en de DocumentDB opgeslagen zijn. Het voordeel hierbij is dat alle documenten opvraagbaar blijven wanneer de PDSDB terug online komt. Het nadeel is dat dit meer werk inhoudt voor de DocumentStorageManager. Aangezien enkel drie uur vereist zijn, gaan we voor het eerste alternatief. De DocumentStorageManager ziet het schrijven in de PDSDB als impliciete ping-berichten waarbij het weet dat de PDSDB nog online is. Wanneer een schrijfrequest faalt, zal er een timer gestart wordt in de cache. Wanneer de PDSDB terug online is, zal deze een eenmalige heartbeat naar de DocumentStorageManager gestuurd worden, waarna hij weet dat hij terug kan schrijven. Hierna worden de documenten (waarvan een id in de cache zit) geschreven naar de PDSDB door de DocumentStorageManager (die leest uit de DocumentDB en schrijft naar de PDSDB). Merk op: als alternatief hadden we gedacht aan GEEN actieve heartbeat vanuit de PDSDB bij het online komen, maar bij een schrijfpoging naar de PDSDB als impliciete ping wanneer er een document opgeslagen wordt. Het probleem hierbij is dat wanneer er al even geen documenten genereerd zijn maar er toch noch documentreferenties in de cache zitten en op dat moment de PDSDB online komt, een registered recipient die documenten niet kan opvragen.

Residual drivers - OtherFunctionality2 USE CASE 12: residual drivers: het opslaan van de recipient id's in de pdsdb –¿ hoort bij deliveryfunctionality. –¿ voorlopig verondersteld dat recipientID in meta data zit die samen met doc opgeslagen wordt in zowel docDB als PDS Ook het inloggen (authenticatie). USE CASE 13: residual drivers: we veronderstellen dat de DocumentStorageManager metadata bij elk document opslaat, like the name of the sender, the data range in which the document should be received and the document type –¿ hoort bij deliveryfunctionality. The metadata is saved in both the pdsdb en database components, because when a user registers, this metadata has to be copied form the one to the other. Dit is om gemakkelijk de documenten in de pdsdb te kunnen zoeken. USE CASE 14: residual driver: mark as received, puntje 3. –¿ beter te doen bij de deliveryfunctionality USE CASE 15: residual driver: tracking.

Alternatives considered

Alternatives for solution A discussion of the alternative solutions and why that were not selected.

3.2.4 Instantiation and allocation of functionality

Decomposition Main aspects of the resulting decomposition.

ModuleB Per introduced component a paragraph describing its responsibilities.

ModuleC Per introduced component a paragraph describing its responsibilities.

Behaviour If needed and explanation of the behaviour of certain aspects of the design so far.

Deployment Rationale of the allocation of components to physical nodes.

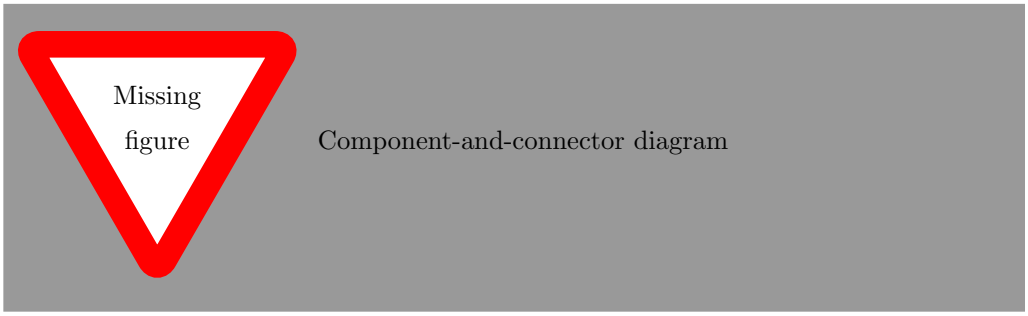


Figure 4: Component-and-connector diagram of this decomposition.

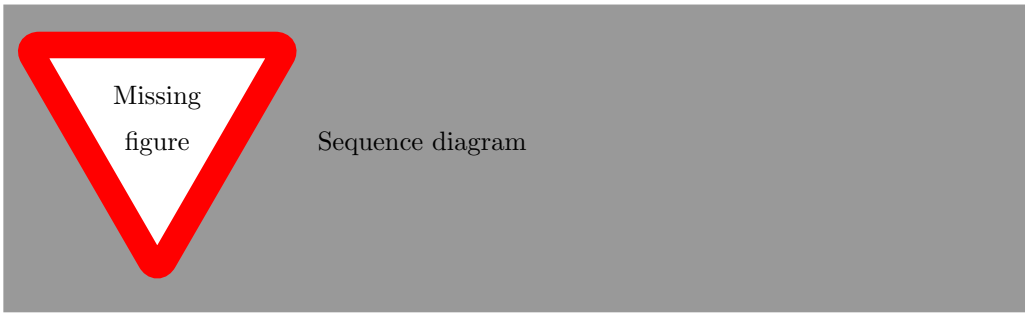


Figure 5: Sequence diagram illustrating a key behavioural aspect.

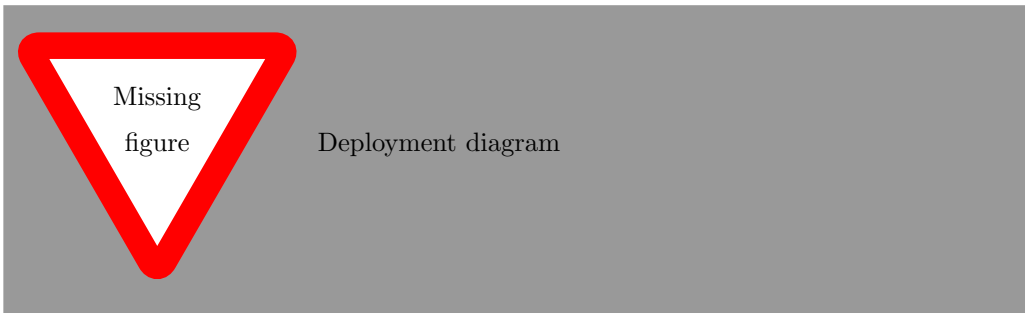


Figure 6: Deployment diagram of this decomposition.

3.2.5 Interfaces for child modules

ModuleB

- InterfaceA
  - `returnType operation1(ParamType param1)` throws TypeOfException
    - Effect: Describe the effect of calling this operation.
    - Exceptions:
      - TypeOfException: Describe when this exception is thrown.
  - `returnType operation2()`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None

3.2.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

returnType This data element represents X.

ParamType This data element represents Y.

3.2.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

ModuleB

- *Z1*: name
- *UCd*: name

ModuleC

- *UCba*: name
  Description which part of the original use case is the responsibility of this component.

3.3 Decomposition 3: ModuleA (X1, Y3, UCa, UCb, UCc)

3.3.1 Module to decompose

In this run we decompose `ModuleA`.

3.3.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *X1*: name
- *Y3*: name

The related functional drivers are:

- *UCa*: name
- *UCb*: name
- *UCc*: name

Rationale A short discussion of why these drivers were selected for this decomposition.

3.3.3 Architectural design

Topic Discussion of the solution selected for (a part of) one of the architectural drivers.

Alternatives considered

Alternatives for solution A discussion of the alternative solutions and why that were not selected.

3.3.4 Instantiation and allocation of functionality

Decomposition Main aspects of the resulting decomposition.

ModuleB Per introduced component a paragraph describing its responsibilities.

ModuleC Per introduced component a paragraph describing its responsibilities.

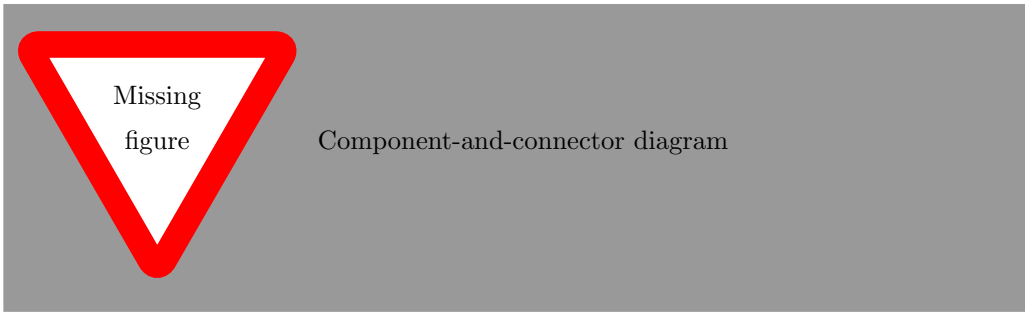


Figure 7: Component-and-connector diagram of this decomposition.

Behaviour If needed and explanation of the behaviour of certain aspects of the design so far.

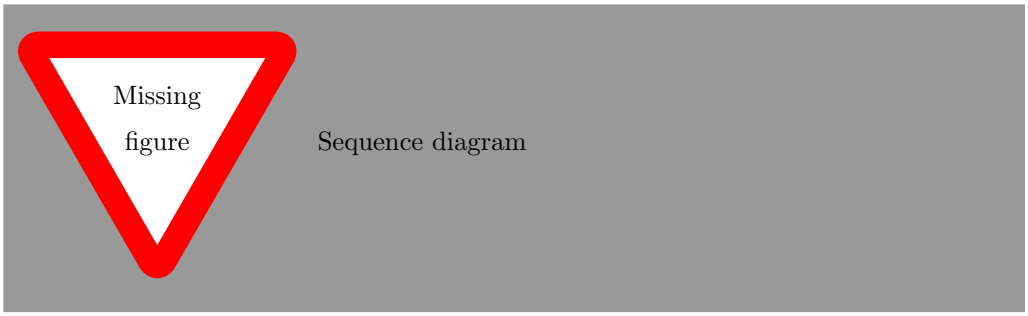


Figure 8: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

3.3.5 Interfaces for child modules

ModuleB

- InterfaceA
  - `returnType operation1(ParamType param1)` throws TypeOfException
    - Effect: Describe the effect of calling this operation.
    - Exceptions:
      - TypeOfException: Describe when this exception is thrown.
  - `returnType operation2()`

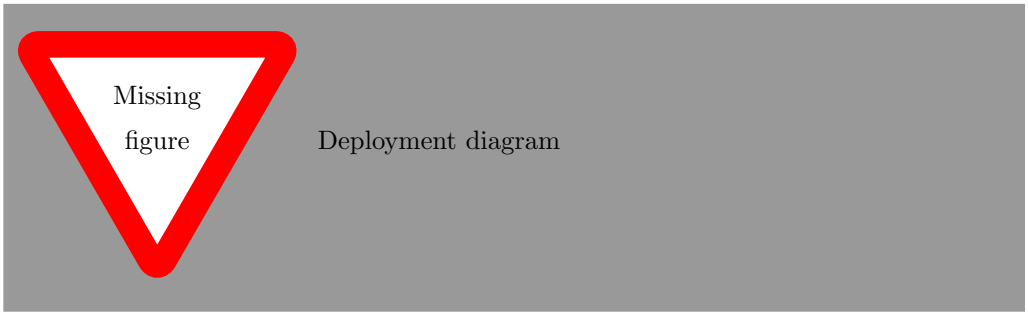


Figure 9: Deployment diagram of this decomposition.

* Effect: Describe the effect of calling this operation.
* Exceptions: None

3.3.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

returnType This data element represents X.

ParamType This data element represents Y.

3.3.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

ModuleB

- *Z1*: name
- *UCd*: name

ModuleC

- *UCba*: name
  Description which part of the original use case is the responsibility of this component.

4 Client-server view (UML Component diagram)

The context diagram of the client-server view. Discuss which components communicate with external components and what these external components represent.

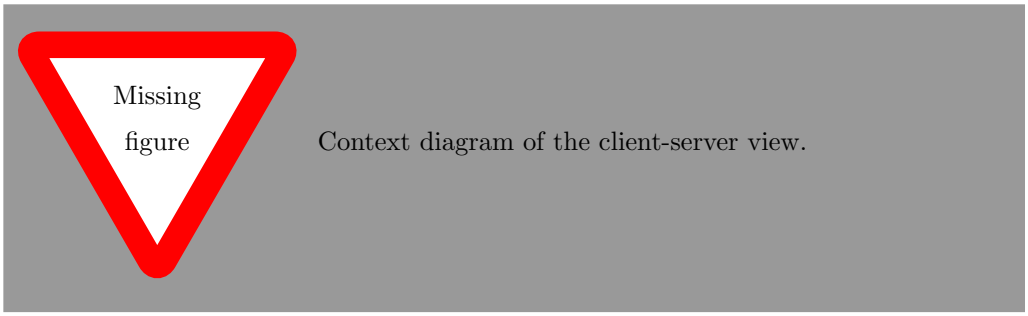


Figure 10: Context diagram for the client-server view.

The primary diagram and accompanying explanation.

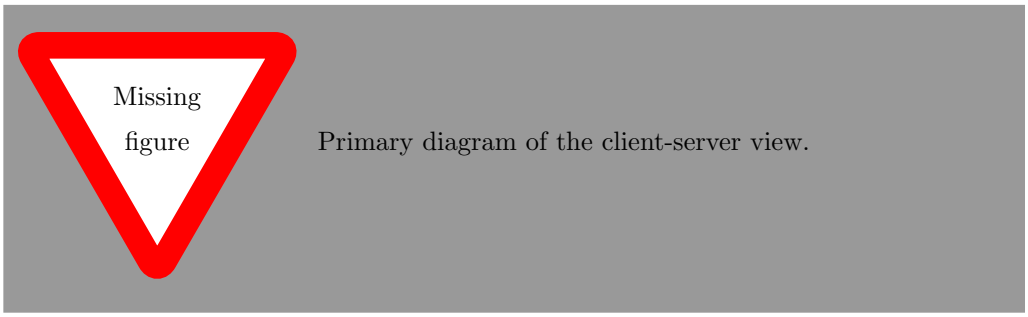


Figure 11: Primary diagram of the client-server view.

4.1 Main architectural decisions

Discuss your architectural decisions for the most important requirements in more detail using the components of the client-server view. Pay attention to the solutions that you employed and the alternatives that you considered. The explanation here must be self-contained and complete. Imagine you had to describe how the architecture supports the core functionality to someone that is looking at the client-server view only. Hide unnecessary details (these should be shown in the decomposition view).

4.1.1 ReqX: requirement name

Describe the design choices related to *ReqX* together with the rationale of why these choices where made.

Alternatives considered

Alternative(s) for choice 1 Explain what alternative(s) you considered for this design choice and why they where not selected.

5 Decomposition view (UML Component diagram)

Discuss the decompositions of the components of the client-server view which you have further decomposed.

5.1 ComponentX

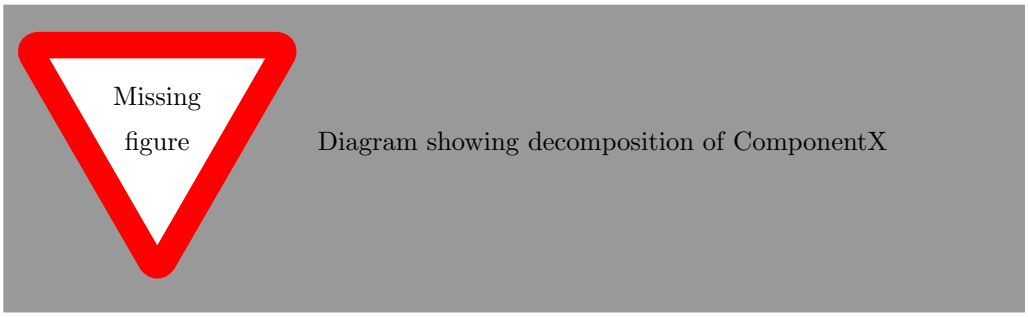
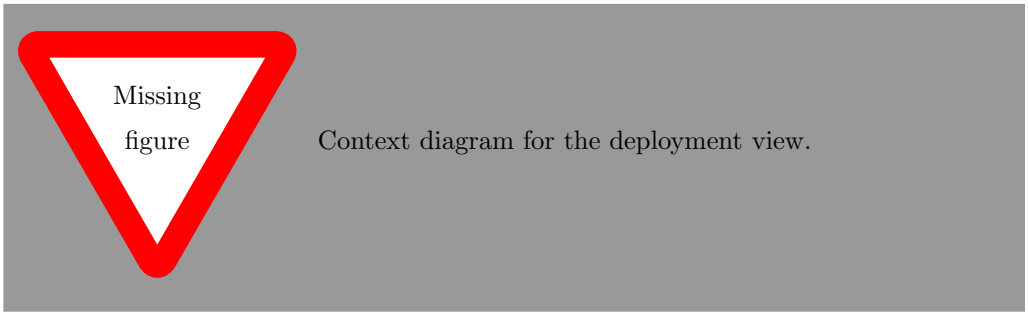


Figure 12: Decomposition of `ComponentX`

Describe the decomposition of `ComponentX` and how this relates to the requirements.

6 Deployment view (UML Deployment diagram)

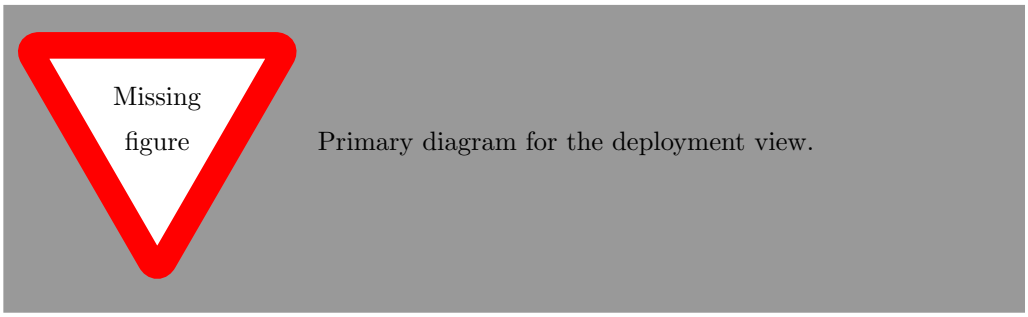
Describe the context diagram for the deployment view. For example, which protocols are used for communication with external systems and why?



Context diagram for the deployment view.

Figure 13: Context diagram for the deployment view.

The primary deployment diagram itself and accompanying explanation. Pay attention to the parts of the deployment diagram which are crucial for achieving certain non-functional requirements. Also discuss any alternative deployments that you considered.



Primary diagram for the deployment view.

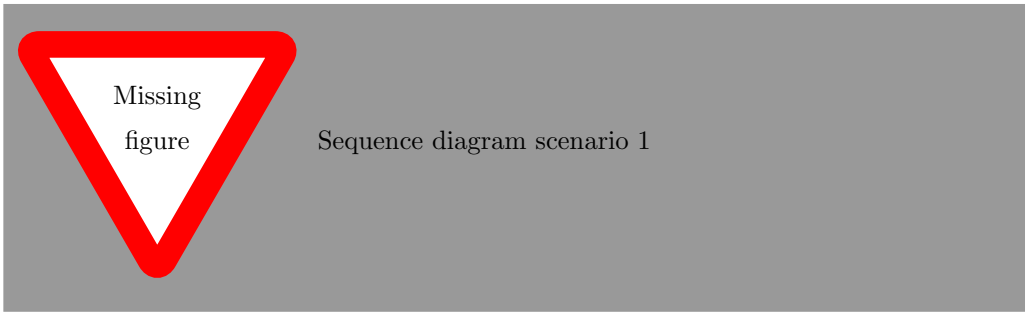
Figure 14: Primary diagram for the deployment view.

7 Scenarios

Illustrate how your architecture fulfills the most important data flows. As a rule of thumb, focus on the scenario of the domain description. Describe the scenario in terms of architectural components using UML Sequence diagrams and further explain the most important interactions in text. Illustrating the scenarios serves as a quick validation of the completeness of your architecture. If you notice at this point that for some reason, certain functionality or qualities are not addressed sufficiently in your architecture, it suffices to document this, together with a rationale of why this is the case according to you. You do not have to further refine you architecture at this point.

7.1 Scenario 1

Shortly describe the scenario shown in this subsection. Show the complete scenario using one or more sequence diagrams.



Sequence diagram scenario 1

Figure 15: The system behavior for the first scenario.

A Element catalog

List all components and describe their responsibilities and provided interfaces. Per interface, list all methods using a Java-like syntax and describe their effect and exceptions if any. List all elements and interfaces alphabetically for ease of navigation.

A.1 Completer

- **Description:** The `Completer` is responsible for fetching the raw data an applicable meta-data for a group of `JobIds` when a `Generator` instance requires a new group of jobs.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- Complete
  - `CompletePartialBatchData getComplete(BatchId batchId, List<JobId> jobIds )`
    - Effect: The `Completer` fetches data needed by a `Generator` for generation of the documents corrseponding to the`JobIds` belonging to the same batch, which is identified by `BatchId`.
    - Exceptions: None

A.2 DocumentGenerationManager

- **Description:** The `DocumentGenerationManager` monitors the availability of the `Generator` components using the Ping interface. The `DocumentGenerationManager` keeps track of the jobs assigned to and being processed by the `Generators`. To minimize the overhead of the job coordination, the `DocumentGenerationManager` assigns jobs to the `Generators` in groups of more than one job that are part of the same batch. If a `Generator` fails to complete its jobs, the `DocumentGenerationManager` can restart these failed jobs.
  It prioritizes jobs based on thei deadlines ansd schedules them according to *P1*.
- **Super-component:** None
- **Sub-components:** `Completer`, `GenerationManager`, `KeyCache`, `Scheduler`, `TemplateCache`

Provided interfaces

- InsertJobs
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - Exceptions: None
- NotifyCompleted
  - `void notifyCompletedAndGiveMeMore(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None
  - `void notifyCompletedAndIAmShuttingDown(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None

A.3 Generator

- **Description:** A `Generator` generates the documents and forwards them to `OtherFunctionality` to store and deliver them. Its availability is monitored by the `DocumenGenerationManager` with the Ping interface. Document processing jobs are assigned to
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- AssignJobs
  - `void assignJobs(CompletePartialBatchData batchData)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - Exceptions: None
- Startup/ShutDown
  - `void startUp(GeneratorId generatorId)`
    - Effect: Starts up the `Generator` instance an gives it the given `GeneratorId`.
    - Exceptions: None
  - `void shutDown()`
    - Effect: The `Generator` completes its assigned group of document generation jobs and report back completion to the `DocumentGenerationManager`, after which is shuts down.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `Generator` will respond to the ping request by sending an echo response. This is used by the `GeneratorManager` to check whether the `Generator` is available.
    - Exceptions: None

A.4 GeneratorManager

- **Description:** The `GenerationManager` is responsible for monitoring the `Generator` instances. It starts up or shuts down these instances based on the number of required instances indicated by the `Scheduler`.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- NotifyCompleted
  - `void notifyCompletedAndGiveMeMore(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None

A.5 KeyCache

- **Description:** The `KeyCache` caches the keys which are most recently used for document generation. The `Completer` has to fetch a key every time a `Generator` instance requests new jobs, while the key will be the same for all jobs belonging to the same batch. The `KeyCache` avoids that the key storage system becomes a bottleneck for document generations. The keys are cached based on the `CustomerId` of a Customer Organization.
- **Super-component:** `DocumentGenerationManager`.
- **Sub-components:** None

Provided interfaces

- GetKey
  - `Key getKey(CustomerId customerId)`
    - Effect: The `KeyCache` looks into its cache for the `Key` belonging to the customer organisation with id `customerId`. If the `Key` is in its cache, it returns it. If the `Key` is not in its cache, it asks `OtherFunctionality` for the `Key` and stores it in its cache, after which it returns that `Key`.
    - Exceptions: None

A.6 PDSDB

- **Description:** The `PDSDB` component is responsible for storing the database of documents in the personal document stores. That database is separated from all other persistent data so that its failure *"does not affect the availability of other types of persistent data"*, as required by *Av2*.
- **Super-component:** None
- **Sub-components:** `PDSDBReplica`, `PDSLongTermDocumentManager`, `PDSReplicationManager`

Provided interfaces

- DocumentMgmt
  - `void storeDocument(DocumentId id, Document doc, MetaData md)`
    - Effect: The `PDSDB` will store the given document`doc` together with the provided metadata `md`.
    - Exceptions: None
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None
  - `void markReceived(DocumentId id)`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None

A.7 PDSDBReplica

- **Description:** Responsibilities of the component.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `List<Document> getDocumentsSince(TimeStamp whenFailed)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void storeDocuments(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `PDSDBReplica` will respond to the ping request by sending an echo response. This is used by the `PDSReplicationManager` to check whether the `PDSDBReplica` is available.
    - Exceptions: None

A.8 PDSLongTermDocumentManager

- **Description:** Responsibilities of the component.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- DocumentMgmt
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - Exceptions: None

A.9 PDSReplicationManager

- **Description:** Responsibilities of the component.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - Exceptions: None

A.10 OtherFunctionality

- **Description:** Responsibilities of the component.
- **Super-component:** None
- **Sub-components:** the direct sub-components, if any.

Provided interfaces

- FinalizeDocument
  Note that the methods in this interface are made idempotent. The methods of this interface are called by `Generator` instances.
  - `void storeAndDeliverDocument(JobId jobid, Document doc)`
    - Effect: The `OtherFunctionality` will store the given document `document` and deliver it. This method is made idempotent. To filer duplicate method calls, it has the `JobId` of the document as an argument. This idempotence is to account for the case when case when a `Generator` fails after forwarding the document and before reporting completion to the `DocumentGenerationManager`. In this case, it can be that the `DocumentGenerationManager` restarts jobs for which a document has already been stored or delivered.
    - Exceptions: None
    - `void generationError(JobId jobid, Error error)`
      - Effect: Describe the effect of the operation
      - Exceptions: None
- GetBatchData
  - `Tuple<JobId, RawData> getRawData(List<JobId> jobIds)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
  - `BatchMetaData getMetaData(BatchId batchId)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- GetKey
  - `Key getKey(CustomerId customerId)`
    - Effect: The `OtherFunctionality` returns the key belonging to the Customer Organization identified by `customerId`.
    - Exceptions: None
- GetTemplate
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived`
    - Effect: The `OtherFunctionality` returns the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`.
    - Exceptions: None
- SetStatus
  - `void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)`
    - Effect: The `OtherFunctionality` stores the `JobStatus` as "temporarily failed" for each of the jobs identified by the given `JobIds`. Used by the `DocumentGenerationManager` for jobs that where assigned to a failed `Generator` instance.
    - Exceptions: None
- NotifyOperator

- `void notifyOperatorOfPDSDBReplicaFailure(PDSDBReplicaId replicaId, TimeStamp dateTime)`
  - Effect: The `OtherFunctionality` will send the given `PDSDBReplicaId` of the failed `PDSDBReplica` with the given time of failure `dateTime` to the eDocs operators.
  - Exceptions: None

A.11 Scheduler

- **Description:** 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.
  The `Scheduler` also indicates to the `GenerationManager` the number of required `Generator` instances through its GetStatistics interface.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetNextJobs
  - `Tuple<BatchId, List<JobId>> getNextJobs()`
    - Effect: The `Scheduler` returns the `JobIds` of the group of jobs that belong to the batch identified by `BatchId` that should be generated next. This method is called by the `GeneratorManager` when a `Generator` instance requires a new group of jobs.
    - Exceptions: None
    - `Tuple<BatchId, List<JobId>> jobsCompletedAndGiveMeMore(List<JobId>)`
      - Effect: The `Scheduler` gets notified that the document processing jobs belonging to the list of `JobIds` are completed. It returns the a list of `JobIds` belonging to a batch identified by `BatchId`. The returned list of `JobIds` identify document processing jobs which are not yet started.
      - Exceptions: None
- InsertJobs
  - `void insertJobs(BatchId batchId, List<JobId> jobIds )`
    - Effect: The `Scheduler` adds the jobs identified by their `JobId` to its 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. This method provides new jobs synchronously to the `Scheduler`, which it schedules synchronously. This means that when the method call returns, the given jobs are scheduled.
    - Exceptions: None
- GetStatistics
  - `int getNumberOfFutureJobs()`
    - Effect: The `Scheduler` returns the amount of documents that should be generated in the near future. The `GeneratorManager` queries this method at regular intervals and adjusts the number of `Generator` instances accordingly.
    - Exceptions: None

A.12 TemplateCache

- **Description:** The `TemplateCache` caches the templates which are most recently used for document generation. The `Completer` has to fetch a templateevery time a `Generator` instance requests new jobs, while the template will be the same for all jobs belonging to the same batch. The `TemplateCache` avoids that the template storage system becomes a bottleneck for document generations. The templates are cached based on the `CustomerId` of a Customer Organization, the type of the document and the date and time at which the batch was provided by the Customer Organization (in order to account for template updates).
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetTemplate
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived`
    - Effect: The `TemplateCache` looks into its cache for the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`. If the `Template` is in its cache, it returns it. If the `Template` is not in its cache, it asks `OtherFunctionality` for the `Template` and stores it in its cache, after which it returns that `Template`.
    - Exceptions: None

B Defined data types

List and describe all data types defined in your interface specifications. List them alphabetically for ease of navigation.

- `BatchId`: A piece of data uniquely identifying a batch of document processing jobs in the system.
- `BatchMetaData`: A data structure listing the metadata belonging to a batch of jobs. This includes the `CustomerID` of a Customer Organization, the `DocumentType` of the documents to be generated, the `TimeStamp` of when the batch was received, ...
- `CompletePartialBatchData`: A complex data structure listing all data a `Generator` needs to complete document generation jobs that are part of the same batch. It contains an array of `Tuple<JobId, RawData>`. The `JobIds` identify jobs that are all part of the same batch. The `RawData` belongs to these document processing jobs. Also listed in the `BatchMetaData` are the values of the `BatchMetaData`, `Key` and `Template` data types belonging to the batch. `CompletePartialBatchData` also contains a `BatchMetaData` entry, a `Key` and a `Template`. *Important to note:* a value of CompletePartialBatchData contains all information necessary to generate **some** jobs of belonging to same batch. It does not have to contain the information of all jobs belonging to same batch.
- `CustomerId`: A piece of data uniquely identifying a Customer Organization in the system.
- `Document`: Description of data type.
- `DocumentId`:
- `DocumentType`: A piece of data describing the type of a document. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `Echo`: The response to a ping message. This data element does not contain any meaningful data.
- `Error`: Description of data type.
- `GeneratordId`: A piece of data uniquely identifying a `Generator` in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `JobBatch`: Description of data type.

- `JobId`: A piece of data uniquely identifying a document processing job in the system.
- `Key`:
- `PDSDBReplicaId`:
- `RawData`: A data structure listing the raw data used in a document processing job.
- `TimeStamp`: The representation of a time (i.e. date and time of day) in the system.
- `Template`: A document used as a template for the generation of documents.
- `TemplateId`: A data structure uniquely identifying a template in the system. It lists three values. It contains `CustomerId` which identifies the Customer Organization who the template belongs to. It also contains a `DocumentType`, specifying for which kind of document it is a template for. The last piece of information it contains is a `TimeStamp` specifying when the system received the template.