Katholieke
Universiteit
Leuven

**Department of
Computer Science**

DOCUMENT PROCESSING

ADD application

Software Architecture (H09B5a and H07Z9a) – Part 2a

Thijs Dieltjens (r0299068)
Wout Vekemans (r0298755)

Academic year 2014–2015

Contents

1 Introduction

This reports describes the first three iterations of an ADD decompostition of the eDocs system, using the given requirements. Every decision made, was driven by ADD principles. Since we only did three decompositions, the presented result is only a partial architecture.

In section 2 we provide the log of the three decompositions. In section 3, we present the partial architecture resulting from these decompositions.

2 Attribute-driven design documentation

2.1 Decomposition 1: eDocs System (Av1, P1, UC4, UC5)

2.1.1 Module to decompose

In this run we decompose the eDocs system as a whole.

2.1.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *P1*: Document generation
- *Av1*: Document generation failure

Two use cases in the eDocs system concern document generation.

- *UC4*: Generate payslip
- *UC5*: Generate invoice

Rationale P1 and Av1 are both high priority requirements. They both concern the generation of documents, which is the core of the eDocs system.

2.1.3 Architectural design

Scheduling for P1 P1 specifies that each document needs to be processed before its deadline expires. To achieve this, the batches of raw data need to be scheduled. For this we introduce the `DocumentProcessingScheduler`. Since there are different document priorities (silver, gold, diamond and critical) and start dates, an *Earliest Deadline First* scheduling algorithm is used. This algorithm is used to ensure that the raw data and the corresponding template are assigned to a `DocumentProcessor` in an optimal order.Whenever a `DocumentProcessor` is idle, the processor pulls the first job from the queue. This way, the `DocumentProcessingScheduler` does not need to worry about the assignment to a `DocumentProcessor`.

Concurrency for P1 P1 specifies that the system should be able to handle load variability in a way that every batch is processed. Therefore the processing needs to be parallel. The amount of parallelism can be adjusted dynamically. To address this requirement the `DocumentProcessingScheduler` checks the length of the queue on each new entry. When the length is above a certain threshold, the `DocumentProcessingScheduler` notifies the `Manager`. This manager wakes up a `DocumentProcessor` and adds it to the set of active processors. The activated processor then starts pulling jobs from the queue and generating documents. When the queue length is below a certain threshold, a notification is sent to the `Manager`. The manager then selects one of the active `DocumentProcessors`. When the selected processor has finished its current job, it is set to non-active. This way there is an efficient amount of active `DocumentProcessors` that still ensures timely document generation.

Transactions for Av1 To ensure that a document is only generated once, we introduce *transactions*. A processor requests a new job from the scheduler (start transaction). The scheduler then assigns a job to the processor. When the processor has finished the job, it reports to the scheduler and writes the documents to provisional non-volatile storage. The scheduler answers with either an abort or a commit message. Only when the processor has to commit, it sends the finished documents to the `SendDocFunctionality`. Otherwise the documents are discarded. If the processor is still active, it can request another job. By using transactions the ACID-properties hold.

Document Processor failure detection for Av1 Av1 states that internal failures should be detected within 5 seconds. Operators should be notified within 1 minute. To do so, the `DocumentProcessingScheduler` keeps a list of all documents that are currently being processed, together with the `DocumentProcessor` on which the process is executed and a timer. When a processor is done with a job and a timeout did not occur, it pulls another job from the queue. When there is a timeout, the scheduler reschedules the job and uses the `NotificationHandler` to notify an administrator of the broken processor. By using transactions this causes no problem if the processor did not fail and requests to store the generated document, since this transaction can never commit.

Broker between scheduler and processors for Av1
We insert an intermediary between the `DocumentProcessingScheduler` and the `DocumentProcessor`. In this way the processor can be completely ignorant of the identity, location and characteristics of the scheduler. If the scheduler becomes available, this does not affect the processors since the broker can forward the request it to another scheduler.

Scheduler failure detection and recovery for Av1
Ping/Echo between the `DocumentProcessingScheduler` and the `Broker` detects when the scheduler fails. Since the communication between `DocumentProcessor` and `DocumentProcessingScheduler` happens through the `Broker`, a replacement can be selected dynamically by the broker. The consistency between the schedulers is guaranteed by passive redundancy. Whenever failure is detected, the `Broker` will use a *warm spare* and set it as the active scheduler. The broker also notifies an `eDocsAdministrator`. Implicit *Ping/Echo* can be done by using the transaction messages. When there is no message to be sent within 4 seconds, a *Ping* is sent.

Manager failure detection for Av1
When the `Manager` fails, *Ping/Echo* between the `DocumentProcessingScheduler` and the `Manager` detects it. An administrator is notified and a replacement can be selected. The consistency between the `Managers` is guaranteed by passive redundancy. When failure is detected, the `eDocsAdministrator` will take a *warm spare* and set it as the active manager. The administrator also informs the `DocumentProcessingScheduler` of the changed manager.

Broker failure detection for Av1
Whenever the `Broker` fails, *Ping/Echo* between the `DocumentProcessingScheduler` and the `Broker` detects it. An administrator is notified and a replacement can be selected. When failure is detected, the `eDocsAdministrator` starts a new `Broker`. The scheduler is also notified of the new broker.

Data Delivery
The needed raw data and templates are supplied via `ProvideRawData` by the `OtherFunctionality`. If the batches are too large to efficiently process on a single `DocumentProcessor`, they are split into smaller packets. The number of documents in the packets is chosen such that each packet can be processed in a timely manner. The packets are then queued individually.

Alternatives considered

Alternatives for passive redundancy in the scheduler Since the `DocumentProcessingScheduler` has explicit state, it is not possible to use *cold spares*. Active redundancy could be an alternative, but it adds a lot of complexity. It could lead to faster swapping between backup schedulers, but the *warm spares* are already fast enough for the availability requirements.

Alternative for broker It is possible to connect the scheduler directly to all the processing units. If the scheduler would fail, this would require all processors to be notified of the new scheduler. By using a broker, this is reduced to just notifying the broker of the new scheduler. Downside to this broker, is the extra level of indirection and added complexity for implementing the broker.

Alternative for timeouts To check whether or not a `DocumentProcessor` failed, the manager could use *Ping/Echo* like the rest of the system. This would cause large amounts of ping messages to be sent by the manager, while most of them are useless. Furthermore timers are already in use for the transactions so not much additional implementation is required.

2.1.4 Instantiation and allocation of functionality

Decomposition The result of the first decomposition is shown in figure 1. Notice that the `SchedulerBackup` has exactly the same functionality and interfaces as the `DocumentProcessingScheduler`. To keep the diagram clean, these connectors are not explicitly drawn. Through this diagram, it is clear that the chosen drivers are addressed. The `OtherFunctionality` provides the raw data, which are then scheduled and processed. After processing, the documents are forwarded to the `OtherFunctionality`. There they are sent to the right recipient, through the preferred channel. The scheduler communicates with a `Manager` to make sure the amount of active processing units is sufficient for the current input load. Failure detection and resolution is handled through the *Ping* messages (sometimes implicit) and notifications sent to an administrator.

The components and their functions are listed below.

DocumentProcessingScheduler Component that is responsible for scheduling the packets for processing. It is also responsible for the flow control in the system. Whenever the processing capacity should be changed, it notifies the `Manager`. The scheduler also has to check whether or not the `Manager` and `Broker` are still alive. If one of these should fail, the scheduler notifies an administrator. The scheduler also checks if the processors are still working. This is done by using a timeout for the transactions. When there is no answer, the scheduler assumes the processor failed. In case of processor failure, a notification is sent.

SchedulerBackup Serves as a backup for the main scheduler. The consistency is guaranteed by passive redundancy. The backup is notified by the main scheduler according to an update interval.

Broker This component forms the connection between a `DocumentProcessor` and the scheduler. The broker is responsible for checking if the scheduler is still alive. If not, it sends a notification to an administrator.

Manager The manager is responsible for managing all processing units. When the capacity should be changed, it can request processing units to start/stop processing jobs.

DocumentProcessor Responsible for the processing of raw data packets into documents. It uses transactions for save delivery of finished documents. A `DocumentProcessor` can either be active or inactive. It communicates with the `Broker` to initiate transactions with the scheduler.

SendDocFunctionality Component used for sending the documents to the right recipient.

NotificationHandler Component that handles the notifications sent by manager and scheduler. It forwards the notifications to the right recipient.

OtherFunctionality This component captures all functionality that is not directly related to document generation. This includes the generation of templates and raw data, and looking up documents in the `SendDocFunctionality`.

Behaviour In figures 2 and 3 we zoom in on the two core functionalities of this decomposition. Figure 2 shows how a document is generated, using the transaction and timer methods described earlier. The processor requests a job and receives a packet that needs to be processed. Then a two phase commit protocol is started. The processor sends a *canCommit* message and the scheduler answers with either *true* or *false*. The documents are then respectively written to the database or discarded. The database also responds with an acknowledgement if the files are correctly written. Figure 3 shows how the scheduler checks its load and then asks for more capacity. The `Manager` then starts another `DocumentProcessor` which then requests a job..

Deployment Most of the components are deployed on different nodes. This causes more delay in communication, but allows for higher loads on each node. In case of hardware failure, only one component fails. It is possible that multiple document processors are deployed on one machine, but not all of them need to be on the same node. This is necessary to provide the requested performance. Multiple brokers on different nodes can also be used to allow scaling with a large number of `Document Processors`. The broker and manager can be on the same node since they both communicate between processor and scheduler.

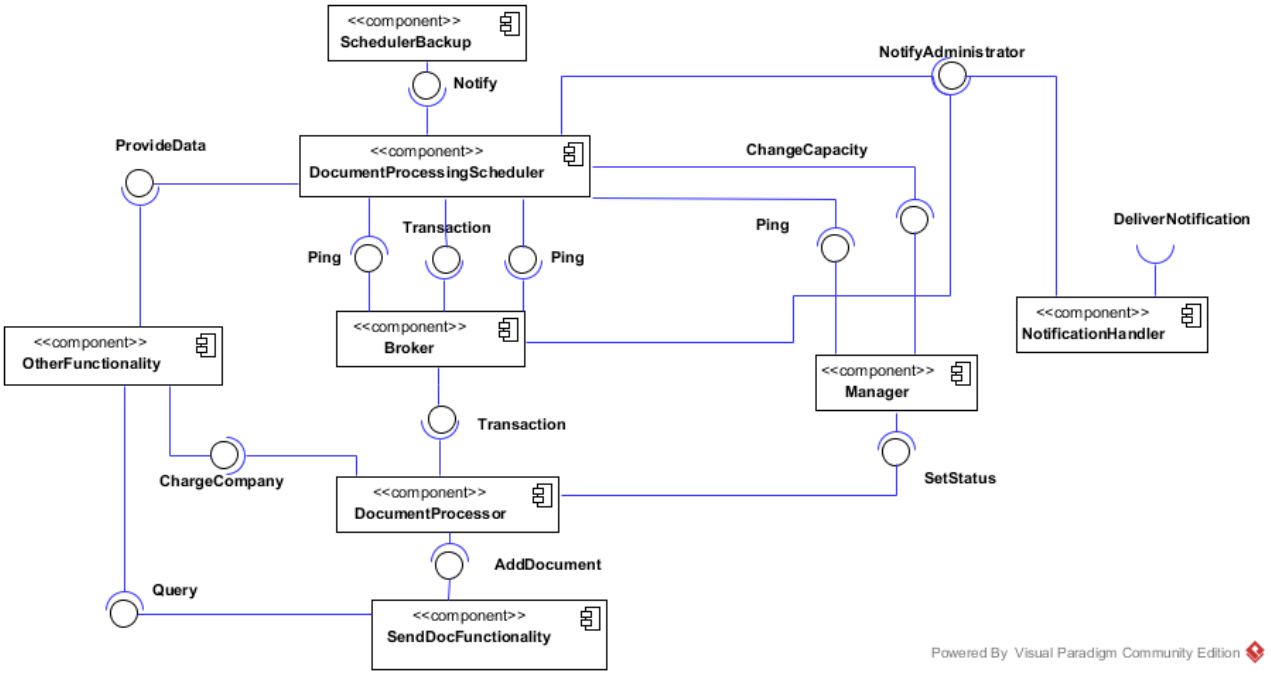


Figure 1: Component-and-connector diagram of this decomposition.

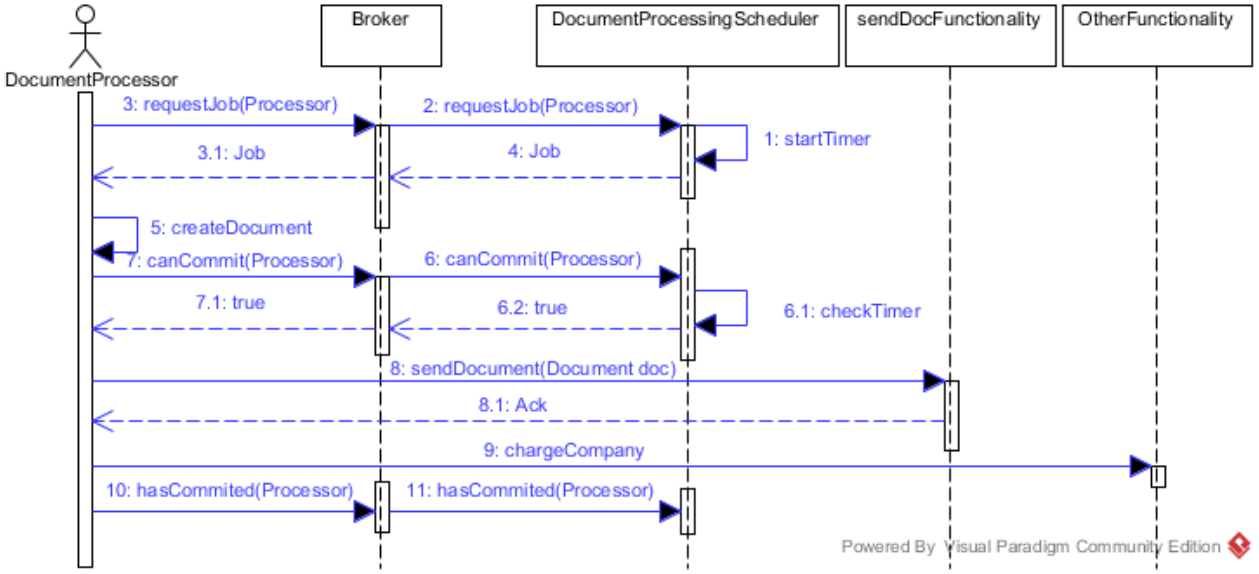


Figure 2: Sequence diagram of the transaction procedure

2.1.5 Interfaces for child modules

This section describes the interfaces assigned to the components detailed in the above section.

Manager

- ChangeCapacity
  - `void increaseCapacity()`
    - Effect: The manager will wake up a `DocumentProcessor`
    - Exceptions: None
  - `void lowerCapacity()`

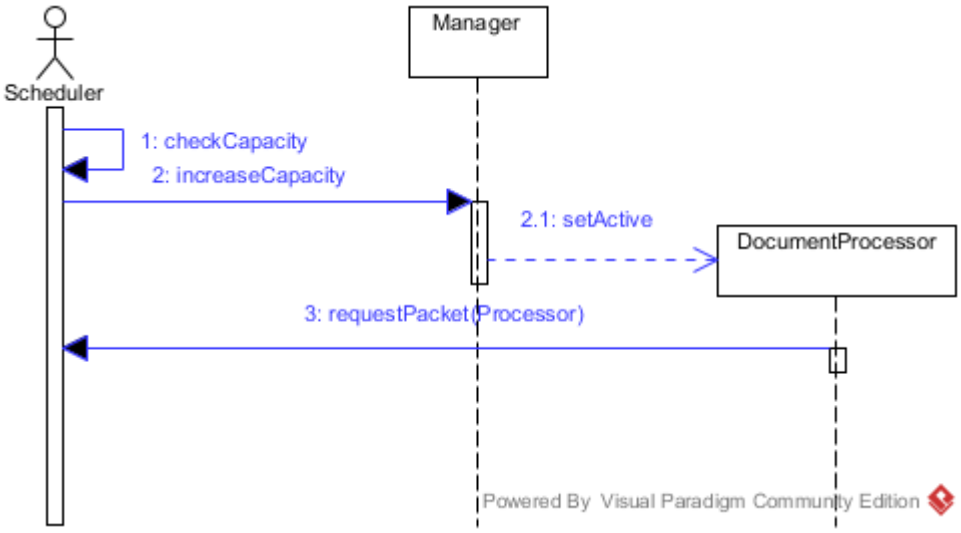


Figure 3: Sequence diagram concerning capacity changes

    - Effect: The manager will notify an active `DocumentProcessor`. It will become inactive after finishing its current job.
    - Exceptions: None

- Ping
  - `Echo ping()`
    - Effect: Returns an echo to the caller indicating that the callee is available.
    - Exceptions: None

Broker

  - Transaction
    - `Packet requestPacket(DocumentProcessor this)`
      - Effect: The DocumentProcessor requests to receive a packet it can process.
      - Exceptions: None
    - `Boolean canCommit(DocumentProcessor)`
      - Effect: The processor needs to know if it can commit the created document to the database.
      - Exceptions: None
    - `void hasCommited(DocumentProcessor)`
      - Effect: The DocumentProcessor notifies he has succesfully commited the processed document to the database.
      - Exceptions: None
  - Ping
    - `Echo ping()`
      - See Manager::Ping

DocumentProcessor

- SetStatus
  - `void setActive())`
    - Effect: The `DocumentProcessor` is activated.
    - Exceptions: None
  - `void setInactive())`
    - Effect: The `DocumentProcessor` is deactivated after finishing its current packet.
    - Exceptions: None

DocumentProcessingScheduler

- Transaction
  - `Packet requestPacket(DocumentProcessor)`
    - See Broker::requestPacket
  - `Boolean canCommit()`
    - See broker::canCommit()
  - `void hasCommited(DocumentProcessor)`
    - See Broker::hasCommited(DocumentProcessor)
- ProvideDataToBeProcessed
  - `void ProvideDataToBeProcessed(List<Packet>)`
    - Effect: The list of packets is added to the queue of the scheduler.
    - Exceptions: None.

SendDocFunctionality

- SendDocument
  - `Ack SendDocument(Document))`
    - Effect: The `Document` is sent to the next level in processing so it can be delivered according to the preferences. An acknowledgment is returned.
    - Exceptions: None
- Query
  - `Document Query(Query q))`
    - Effect: The database is queried and returns the document(s) that correspond to it.
    - Exceptions: None

SchedulerBackup

- Notify
  - `void Notify(UpdateInformation))`
    - Effect: The internal status of `SchedulerBackup` is updated to that of the `DocumentProcessingScheduler`.
    - Exceptions: None

NotificationHandler

- NotifyAdministrator
  - `void NotifyAdministrator(Notification))`
    - Effect: The given `Notification` will be send to an eDocs administrator.
    - Exceptions: None

2.1.6 Data type definitions

Packet This data element represents a packet of raw data.

Notification A message for the eDocs administrator concerning a failed component.

Ack An acknowledgment.

UpdateInformation All the information the backup needs to update its internal status to that of the `DocumentProcessingScheduler`.

Query A query to search documents in the Document DB.

2.1.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for. If requirements are split into parts, a letter is added to the name of the requirement (e.g. *UC4**a***).

OtherFunctionality

- *UC1*: Log in
- *UC2*: Log out
- *UC3a*: Initiate document processing
- *UC4a*: Store the payslip
- *UC5a*: Store the invoice
- *UC4b*: Sending the payslip
- *UC4c*: Charge the customer organisation
- *UC5b*: Sending the invoice
- *UC6*: Deliver document via e-mail
- *UC8*: Deliver document via postal mail
- *UC9*: Deliver invoice via Zoomit
- *UC10*: Confirm document delivery (Zoomit)
- *UC11a*: Deliver document via personal document store
- *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
- *UC16*: Register to personal document store
- *UC17*: Unregister from personal document store
- *UC18a*: Register customer organisation
- *UC19*: Unregister customer organisation
- *UC20*: Update document template
- *UC21*: Consult status of all document processing jobs
- *Av2*: Personal document storage failure
- *Av3*: Zoomit failure
- *P2*: Document lookups
- *P3*: Status overview for customer administrators
- *M1*: New type of document: bank statements
- *M2*: Multiple print & postal services
- *M3*: Dynamic selection of the cheapest of print & postal services

DocumentProcessingScheduler

- *UC3b*: mark jobs as initiated

NotificationHandler

- *UC7*: Notify of e-mail delivery failure
- *UC11b*: Notify recipient
- *UC18b*: Notify customer organisation
- *UC22*: Notify customer administrator

SchedulerBackup

- None

Broker

- None

Manager

- None

DocumentProcessor

- None

2.2 Decomposition 2: Personal Document Store (Av2, P2, UC11, UC12, UC13)

2.2.1 Module to decompose

In this decomposition, we decompose the `SendDocFunctionality` component and a part of the `OtherFunctionality` to add the personal document store to the system.

2.2.2 Selected architectural drivers

We selected following non-functional drivers:

- *Av2*: Personal document storage failure
- *P2*: Document lookups

The most important use cases concerning the personal document store are listed below.

- *UC11*: Deliver document via personal document store
- *UC12*: Consult personal document store
- *UC13*: Search documents in personal document store

Rationale Av2 is of high importance to the system. Since P2 is closely related (they both concern the personal document store), we decided to combine them in this decomposition. The personal document store is one of the key components of document delivery, so the chosen drivers are valuable.

2.2.3 Architectural design

Provisional storage for Av2 The system needs to have a degraded mode, in which the personal document store is not fully functioning. There cannot be loss of documents, so we added the `ProvisionalDB`. Whenever a document is written to the `PrimaryDocStoreDB`, it is also written to the provisional database. Documents in the provisional storage are only kept for three hours. When the system recovers from a failure, the provisional database flushes its contents to the primary database. The degraded mode can be communicated to the end users of the system by notifying the `Reader`. This way the system fails gracefully.

Replication for P2 The database should be available at all times. This is done using *passive replication*. Each write to the `PrimaryDocStoreDB` is synced to the `BackupDocStoreDBs`. This system only guarantees eventual consistency, but the delay is acceptable. Each document will still be available within minutes. In contrast to standard passive replication, reads are performed on the backup databases and cannot be performed on the primary database so a large amount of reads does not affect other parts of the eDocs system. The `Readers` make sure the load is balanced over all backup databases. This is used to avoid putting large loads on a single database.

Concurrency for P2 To ensure that multiple users can access their documents at the same time, a concurrency system is used. The users have access to a set of `Readers` through an `Interface`. The readers perform queries in a way that the load is balanced on all active backup databases.

Provisional database failure detection for Av2 The `DocstoreManager` detects failure of the `ProvisionalDB` by using the *Ping/Echo* tactic. When failure occurs, an administrator is notified. The provisional database is then repaired. When the provisional database is being repaired, all writes to the databases are stalled. The `DocstoreManager` keeps them in its buffer until the provisional database is fully functional again.

Primary database failure detection for Av2 Whenever the `PrimarayDocStoreDB` fails, it is detected by the `DocstoreManager` using *Ping/Echo*. An administrator is notified of the failure, and the primary database is repaired. The manager also selects one of the `BackupDocStoreDBs` as a replacement for the failed primary.

Exceptions for Av2 The `Reader` is aware of database failure. This happens through absence of query response, or a `SetMode` message from the manager. This message can put the system in degraded mode. Whenever the `Interface` wants to perform a query and the `Reader` cannot fulfill the request, an exception is sent to the interface which handles this and shows a message to the user. This way the systems fails gracefully.

Alternatives considered

Alternative for passive replication Another option for the replication of the `PrimaryDocStoreDB` is to use *active replication*. We considered this, but there is no improvement in performance. Active replication does not scale in case of large amounts of requests. Our method (sending requests to a backup, with load balancing), has better scaling for heavily used systems.

Alternative for provisional database To prevent the loss of documents in case of database failure, it would be possible to use active replication with multiple equivalent databases, but this would lead to large delays, since the writer should wait for a response of all the replicas.

Alternative for exceptions To notify the `Interface` of database failure, one could also use a notification sent by the `Reader`. When a reader is notified of the failure (through *SetMode* or a non-answered query), it sends a message to the interface. The interface then adapts to the degraded mode, by notifying the users of the failure before they enter a query, instead of after the query. This could lead to a lot of useless messages sent between reader and interface, for example when there are no queries performed in the period of database outage. By using the exceptions, the user is only notified of failure if he actually performs a query.

Alternative for reader The reader introduces an extra level of indirection and thus adds the complexity of creating this component to the system. An alternative could be to remove it from the system and let the interface directly connect with an assigned `BackupDocstoreDB`. However this way a failure of a backup database is no longer hidden. With the chosen solution a replacement database can be dynamically chosen and the interface does not need any information about the databases.

2.2.4 Instantiation and allocation of functionality

Decomposition The result of this decomposition can be seen in figure 4. It is clear that this part of the system represents a database, which can be queried by an interface. Internally, the database is split in three parts. The primary database normally contains all documents in the document store. The backup database is synced with the primary database and can be used in case of system failure. The provisional database is used to store the documents generated in the last three hours. When the system recovers from failure, the

provisional database writes its contents to the primary database, to make sure no documents are lost. A backup database ensures availability after failure, and the `Interface` in combination with the `Reader` makes sure users can access documents in their personal storage in a timely manner. The partial architecture of the eDocs system after this decomposition is shown in figure 5.

DocForwarder This component is responsible for sending the generated document to the right delivery channel.

DocStoreManager This component sends documents to the primary and the provisional database simultaneously. It also check whether these two databases are still alive using the *Ping/Echo* tactic. This manager also notifies the `Readers` when degraded mode is active and which `BackupDocStoreDBs` are active.

PrimaryDocStoreDB This component is the main database of the personal document store. When the system is functioning normally, it contains all personal documents. The component periodically synchronizes its state with the `BackupDocStoreDBs`.

BackupDocStoreDB This component is a synced copy of the `PrimaryDocStoreDB`. When the system fails, this backup is used to guarantee a minimun downtime. This database accepts queries from the `Reader`.

ProvisionalDB This part of the system stores all documents that were generated in the last three hours. It can also flush its content to the main database after failure.

NotificationHandler See `NotificationHandler` in decomposition 1 (2.1.3)

Reader Performs queries on the `BackupDocStoreDBs`, that were sent to it by the `interface`.

Interface Represents the user interface of the personal document store. Allows users to consult the datastore and search for documents.

OtherFunctionality2 This component contains all other functionality of the system.

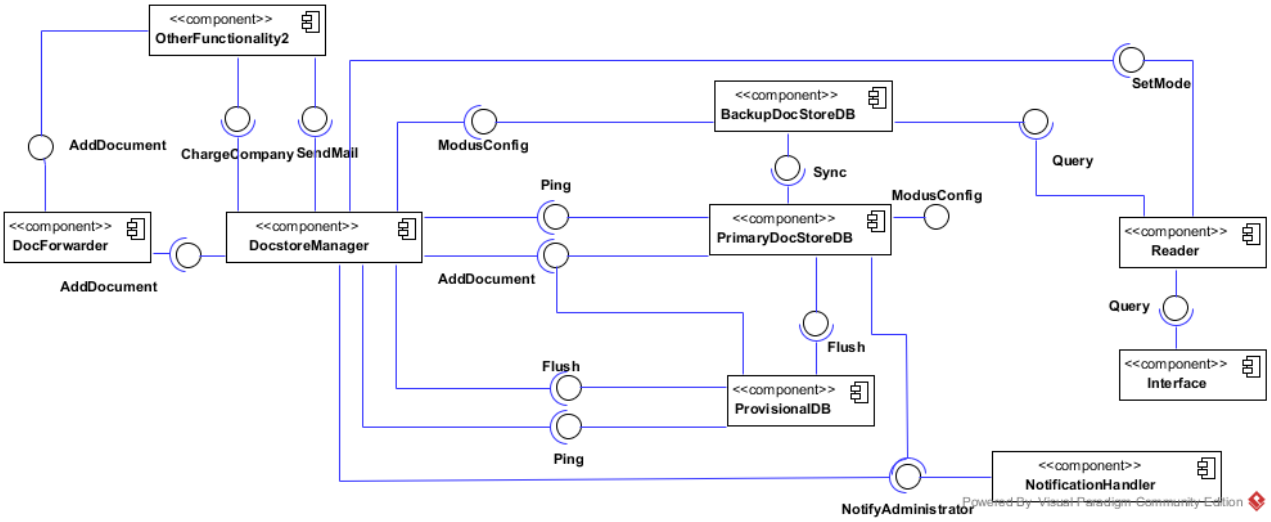


Figure 4: Component-and-connector diagram of the personal document store decomposition.

Behaviour In figure 6 we focus on adding a document to the personal document store, as described in UC11. The `DocForwarder` receives a processed document, and sends it to the `DocStoreManager`. The manager then writes it to the `ProvisionalDB` and the `PrimaryDocStoreDB`. The primary database then synchronizes the document with its `BackupDocStoreDBs`.

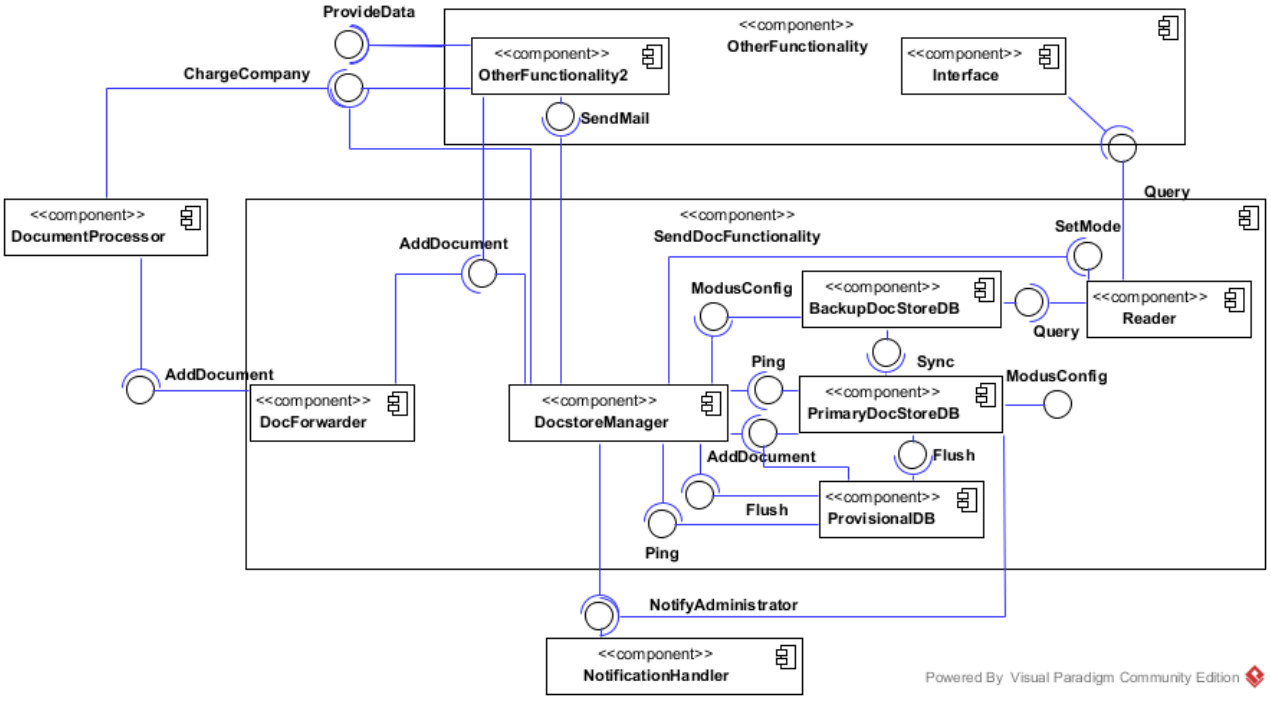


Figure 5: Partial architecture after 2 decompositions.

Deployment The `DocForwarder`, `NotificationHandler` and OtherFunctionality2 are kept on the same node. The provisional database needs to be on a single node to make sure documents can not be lost. The primary and backup databases all need to be on different nodes so they can guarantee the performance requirements. Multiple readers and the `DocstoreManager` can be on the same node. For a visual representation see figure 13 (this also contains components from iteration 3).

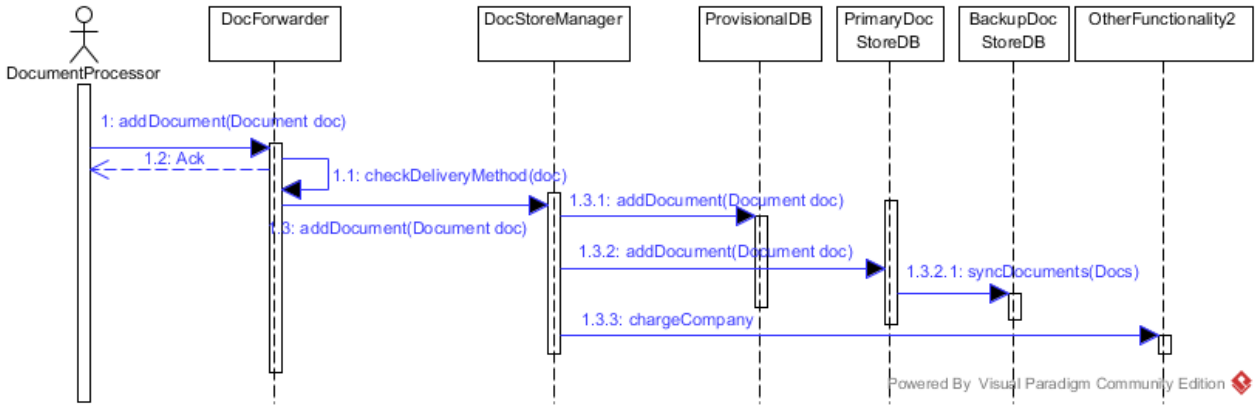


Figure 6: Sequence diagram of adding a document to the personal document store

2.2.5 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for. If requirements are split into parts, a letter is added to the name of the requirement (e.g. *UC4**a***).

OtherFunctionality2

- *UC1*: Log in
- *UC2*: Log out

- *UC3*: Initiate document processing
- *UC4a*: Charge the customer organisation
- *UC5a*: Charge the customer organisation
- *UC4b*: Store the payslip
- *UC5b*: Store the invoice
- *UC6a*: Deliver document via e-mail
- *UC8a*: Deliver document via postal mail
- *UC9a*: Deliver invoice via Zoomit
- *UC10*: Confirm document delivery (Zoomit)
- *UC11b*: Send e-mail
- *UC14*: Consult document in personal document store
- *UC15*: Download document via unique link
- *UC16*: Register to personal document store
- *UC17*: Unregister from personal document store
- *UC18a*: Register customer organisation
- *UC19*: Unregister customer organisation
- *UC20*: Update document template
- *UC21*: Consult status of all document processing jobs
- *Av3*: Zoomit failure
- *P2b*: Concurrent consults of specific documents and pdf downloads
- *P3*: Status overview for customer administrators
- *M1*: New type of document: bank statements
- *M2*: Multiple print & postal services
- *M3*: Dynamic selection of the cheapest of print & postal services

DocForwarder

- *UC6b*: Initiate delivery via e-mail
- *UC8b*: Initiate delivery via postal mail
- *UCb9*: Initiate delivery via Zoomit

NotificationHandler

- *UC7*: Notify of e-mail delivery failure
- *UC18b*: Notify customer organisation
- *UC22*: Notify customer administrator

DocumentProcessingScheduler

- None

SchedulerBackup

- None

Broker

- None

Manager

- None

DocumentProcessor

- None

DocstoreManager

- None

PrimaryDocStoreDB

- None

BackupDocStoreDB

- None

ProvisionalDB

- None

Reader

- None

Interface

- None

2.3 Decomposition 3: Document processing (P1, M1, UC3, UC4, UC5)

2.3.1 Module to decompose

In this decomposition, we decompose the `DocumentProcessor` and a part of `OtherFunctionality2` to provide a more concrete view on document processing. The initialisation of this processing is also added.

2.3.2 Selected architectural drivers

We selected the following non-functional drivers:

- *P1*: Document generation
- *M1*: New type of document: bank statements

The most important related use cases that were not fully added to the system in earlier decompositions are:

- *UC3*: Initiate document processing
- *UC4*: Generate payslip
- *UC5*: Generate invoice

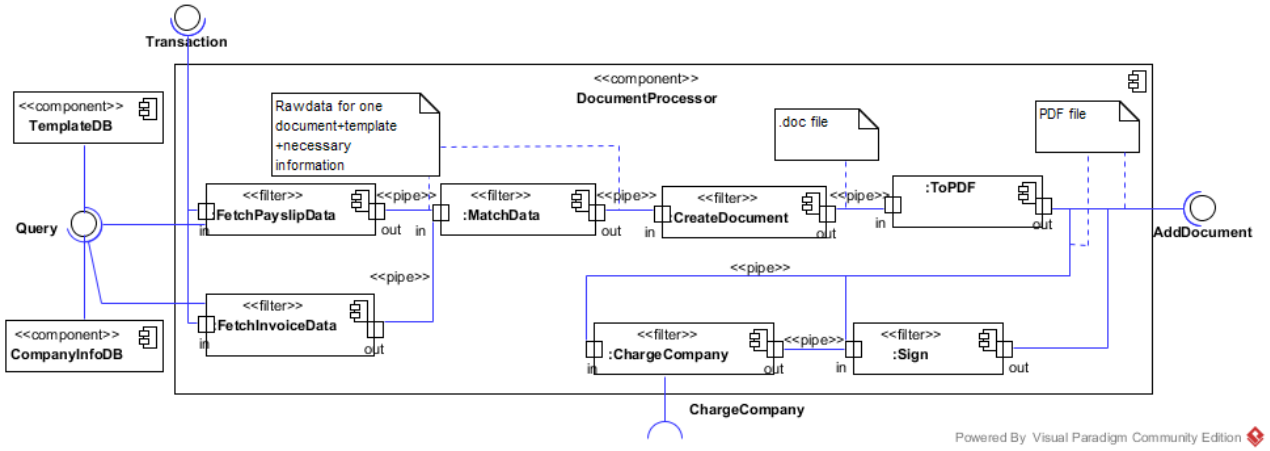


Figure 7: Document generation

Rationale The selection of non-functional requirements is based on the incompleteness of the document generation part of the system. The *M1* requirement has medium priority and was not yet addressed. By elaborating the document generation part the requirements of *P1* can also be tackled more deeply. We added *UC3* to finish the entire document processing part of the eDocs system.

2.3.3 Architectural design

Pipe-and-filter for packet creation for P1 When the processing of raw data into documents is started, the data should be validated. The compliance to the SLA should be checked. After that, the raw data is split into packets. Finally, each packet should be validated. This are four actions that concern the same batch of raw data. By using the pipe-and-filter pattern, more batches can be started per time unit. The overhead introduced by this pattern is minimal in this situation, since there are only three filters used. The pattern also allows for batches to be processed concurrently. For interactive systems this might pose a threat, but after the starting of a job, there is no interaction required until the document is generated. The filter structure for packet creation is shown in figure 8b.

Pipe-and-filter for document generation for P1 The document generation process contains different steps. First, the compatibility between template and raw data should be validated. After that, the template should be filled with the raw data. Then, a PDF file is created from the document. In case of an invoice, the PDF file should be signed. A filter to charge a customer organisation can be added to the pipeline if necessary. This problem of different actions on the same file can be easily addressed using a pipe-and-filter pattern. This way, the throughput of a processing unit is also larger since each filter can be assigned to a different thread. Depending on the kind of document that is generated, other filters or sequences of filters might be used (see figure 7).

Adding other document types for M1 The first modifiability requirement states that is should be possible to add a different kind of document. This can be done easily, by modifying the used set of filters to generate the document.

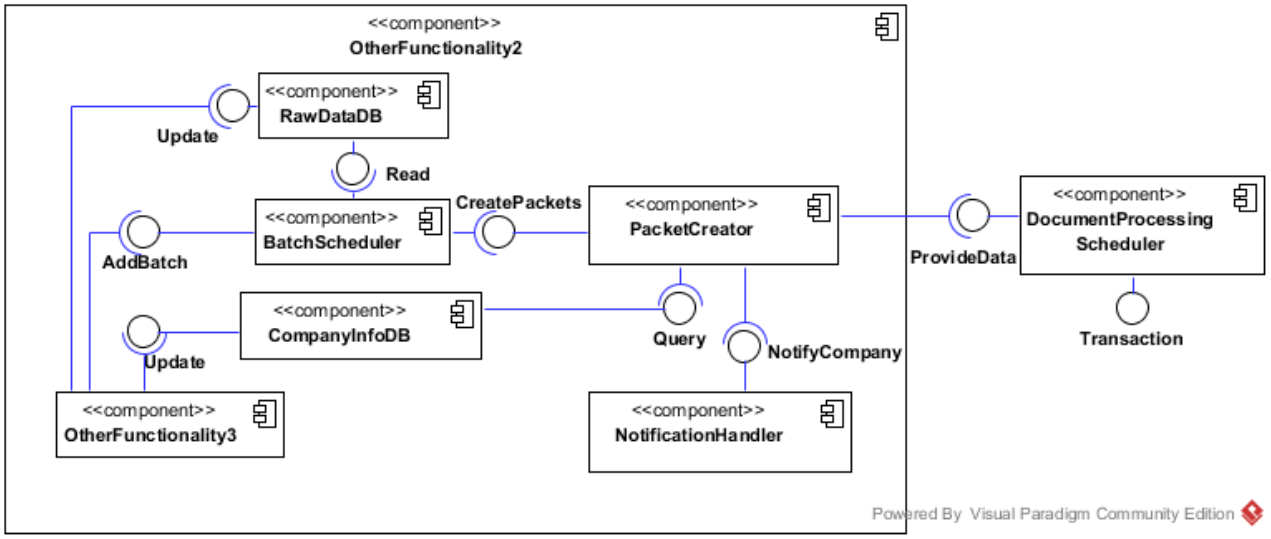
Alternatives considered

Alternative for pipe-and-filter An alternative for the splitting of the document generation process into different steps that can run concurrently, is to create a component that does these steps at once on each input. This would make the processing units overly complex and would require a different kind of component for each type of file. By splitting the process in smaller steps, specialised, smaller units can be used. The splitting also leads to a pipeline. This causes an higher overall throughput, since one document can start at step 1, when the previous document is only at step 2. With a single processing unit, each document should wait until the previous document is fully processed.

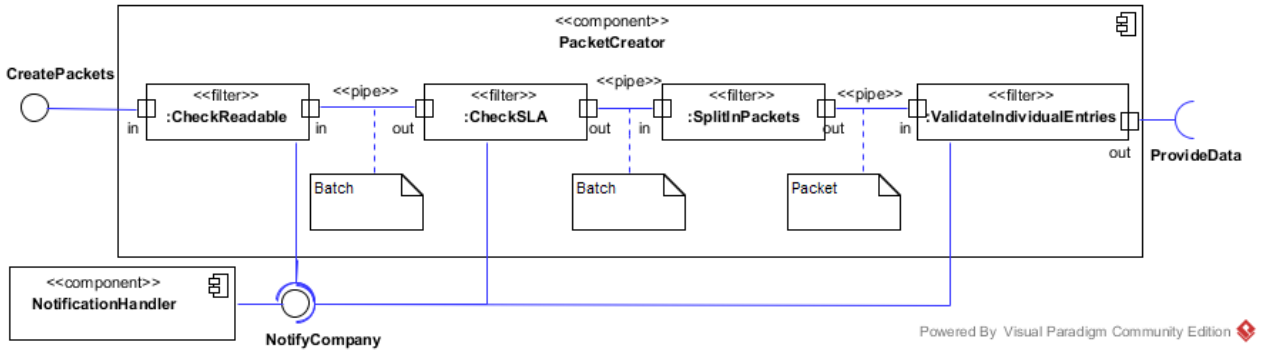
Alternative for modifiability Related to the previous alternative, different kinds of documents can be processed on different nodes. This would lead to lots of complex processing units in case of large amounts of document types. By using the pipe-and-filter as we did, this is less of a problem. There is only need for some extra filters, and some filters can be reused from document types. We see the filters as a box of building blocks. Depending on the document type, a different chain of filters is constructed.

2.3.4 Instantiation and allocation of functionality

Decomposition The result of the decomposition can be seen in figure 8. The decomposed part of the system is responsible for the whole document generation process. It starts with the `BatchScheduler` reading the needed raw data from the database. After that, `PacketCreator` splits the batch into packets. These packets are then forwarded to the `DocumentProcessingScheduler`. In figure 8 the `PacketCreator` is decomposed further. A pipeline is constructed to validate and split the batch into packets of the appropriate size. The partial architecture after this decomposition can be seen in figure 9.



(a) Initiation of document processing



(b) Packet creation

Figure 8: Result of the decomposition

TemplateDB This component is used to store all document templates for all different customer organisations.

RawDataDB This component is used to store all raw data provided by customer organisations. The data is kept in the database after the documents are generated.

BatchScheduler This component processes the requests coming from different customer organisations. It keeps all the deadlines and recurring jobs. It sends all raw data to the `PacketCreator` in a timely manner.

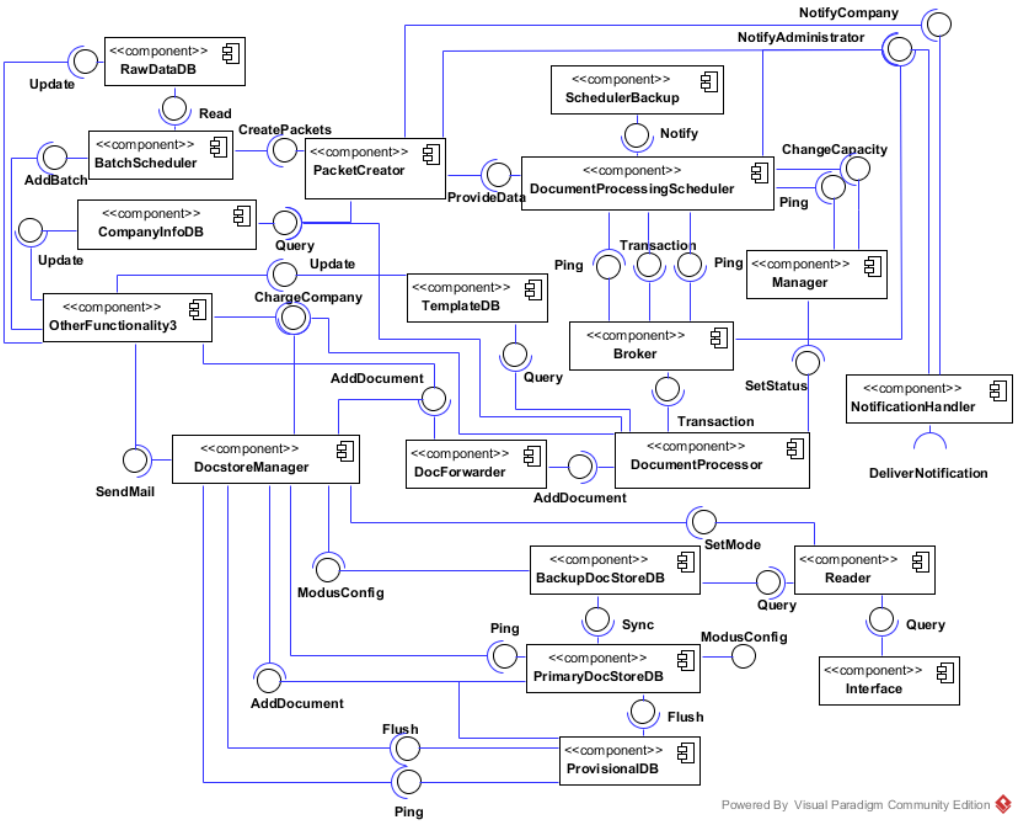


Figure 9: Partial architecture after 3

PacketCreator This component creates smaller packets of raw data from the provided batches. It checks SLA compliance and forwards the packets to the `DocumentProcessingScheduler`.

CompanyInfoDB This component keeps the information about the different customer organisations. The SLAs are stored in this database.

NotificationHandler Forwards the notifications sent by the `PacketCreator` to the right customer organisation.

DocumentProcessingScheduler See first decomposition (2.1.4).

DocumentProcessor See first decomposition (2.1.4).

Deployment The `BatchScheduler` and `PacketCreator` are kept on the same node, since they work closely together. The databases containing company information and templates are also deployed on the same node. They provide the same interface and are both used by the `DocumentProcessors`. The RawDataDB is kept on an achiveDB node that can later on be used to also store the finished documents. This makes it easier to track the status of the documents. For a visualization, see figure 13.

2.3.5 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for. If requirements are split into parts, a letter is added to the name of the requirement (e.g. *UC4**a***).

OtherFunctionality3

- *UC1*: Log in
- *UC2*: Log out

- *UC4a*: Charge the customer organisation
- *UC5a*: Charge the customer organisation
- *UC4b*: Store the payslip
- *UC5b*: Store the invoice
- *UC6a*: Deliver document via e-mail
- *UC8a*: Deliver document via postal mail
- *UC9a*: Deliver invoice via Zoomit
- *UC10*: Confirm document delivery (Zoomit)
- *UC11b*: Send e-mail
- *UC14*: Consult document in personal document store
- *UC15*: Download document via unique link
- *UC16*: Register to personal document store
- *UC17*: Unregister from personal document store
- *UC18*: Register customer organisation
- *UC19*: Unregister customer organisation
- *UC20*: Update document template
- *UC21*: Consult status of all document processing jobs
- *Av3*: Zoomit failure
- *P2b*: concurrent consults of specific documents and pdf downloads
- *P3*: Status overview for customer administrators
- *M2*: Multiple print & postal services
- *M3*: Dynamic selection of the cheapest of print & postal services

DocForwarder

- *UC6b*: Initiate delivery via e-mail
- *UC8b*: Initiate delivery via postal mail
- *UCb9*: Initiate delivery via Zoomit

NotificationHandler

- *UC7*: Notify of e-mail delivery failure
- *UC18b*: Notify customer organisation
- *UC22*: Notify customer administrator

DocumentProcessingScheduler

- None

SchedulerBackup

- None

Broker

- None

Manager

- None

DocumentProcessor

- None

DocstoreManager

- None

PrimaryDocStoreDB

- None

BackupDocStoreDB

- None

ProvisionalDB

- None

Reader

- None

Interface

- None

RawDataDB

- None

TemplateDB

- None

BatchScheduler

- None

CompanyInfoDB

- None

PacketCreator

- None

3 Resulting partial architecture

This section provides an overview of the architecture constructed through three phases of ADD.

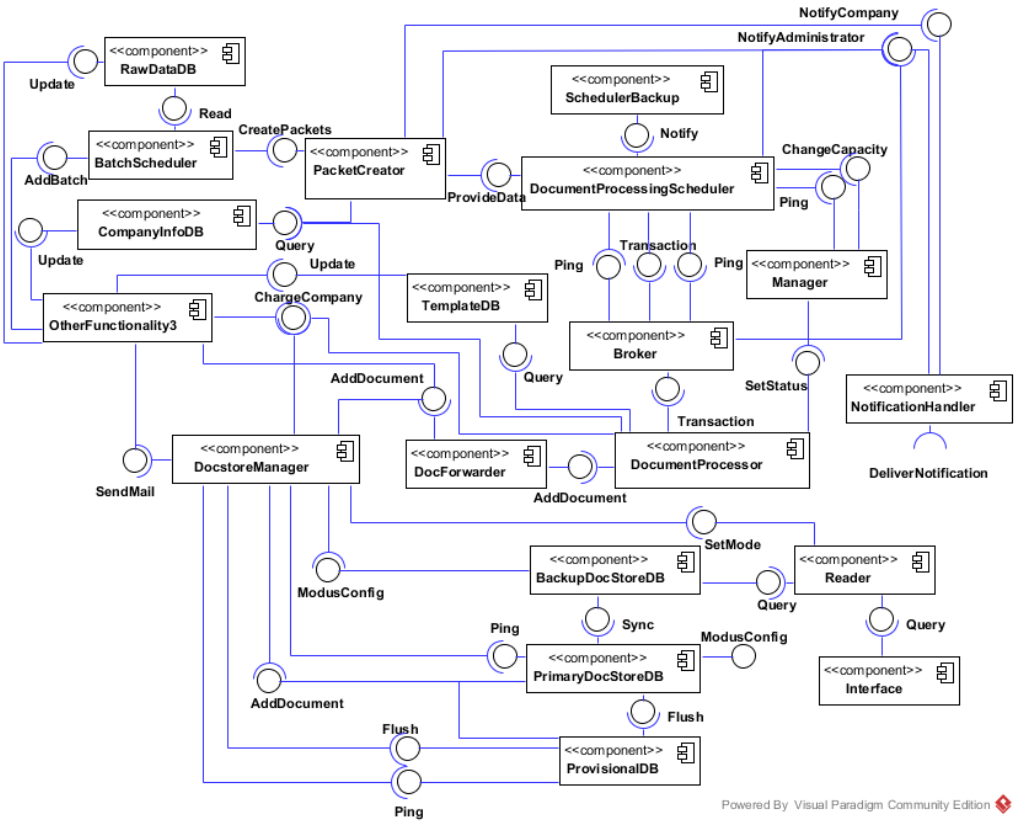


Figure 10: Primary diagram for the component-and-connector view.

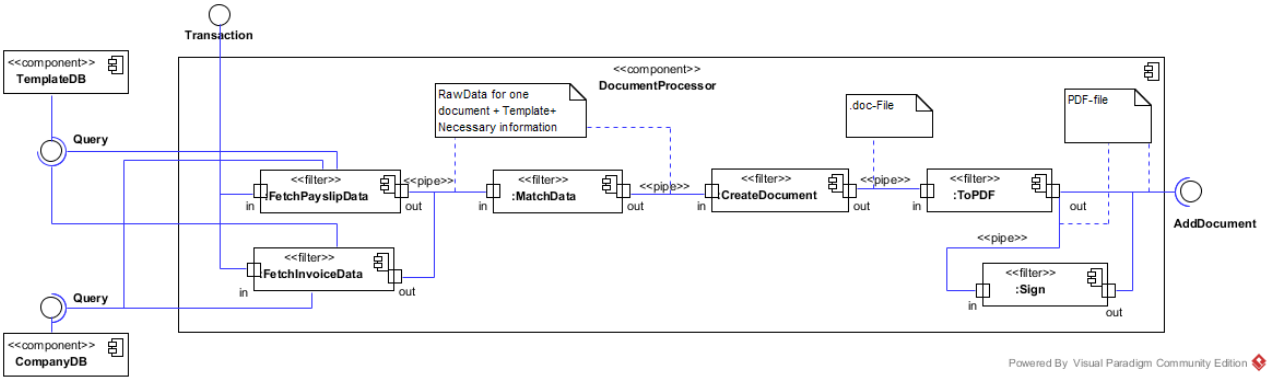


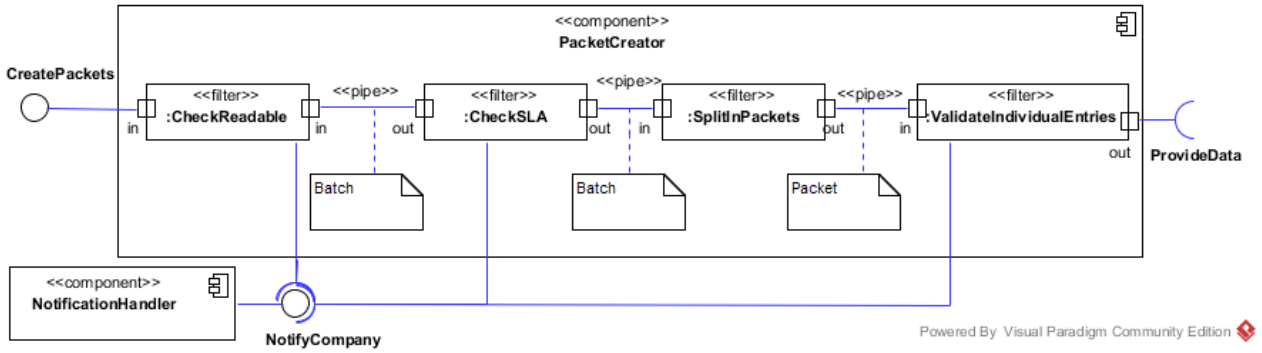
Figure 11: Decomposition of `DocumentProcessor` from figure 10

3.1 Component-and-connector view

A component-connector view on the system after three decompositions can be seen in figure 10. In figures 11 and 12 you can see a more detailed decomposition of the `DocumentProcessor` and `PacketCreator` components in the overview.

3.2 Deployment view

The mapping of the components on different hardware nodes is shown in figure 13.



from figure 10

Figure 12: Decomposition of `PacketCreator`

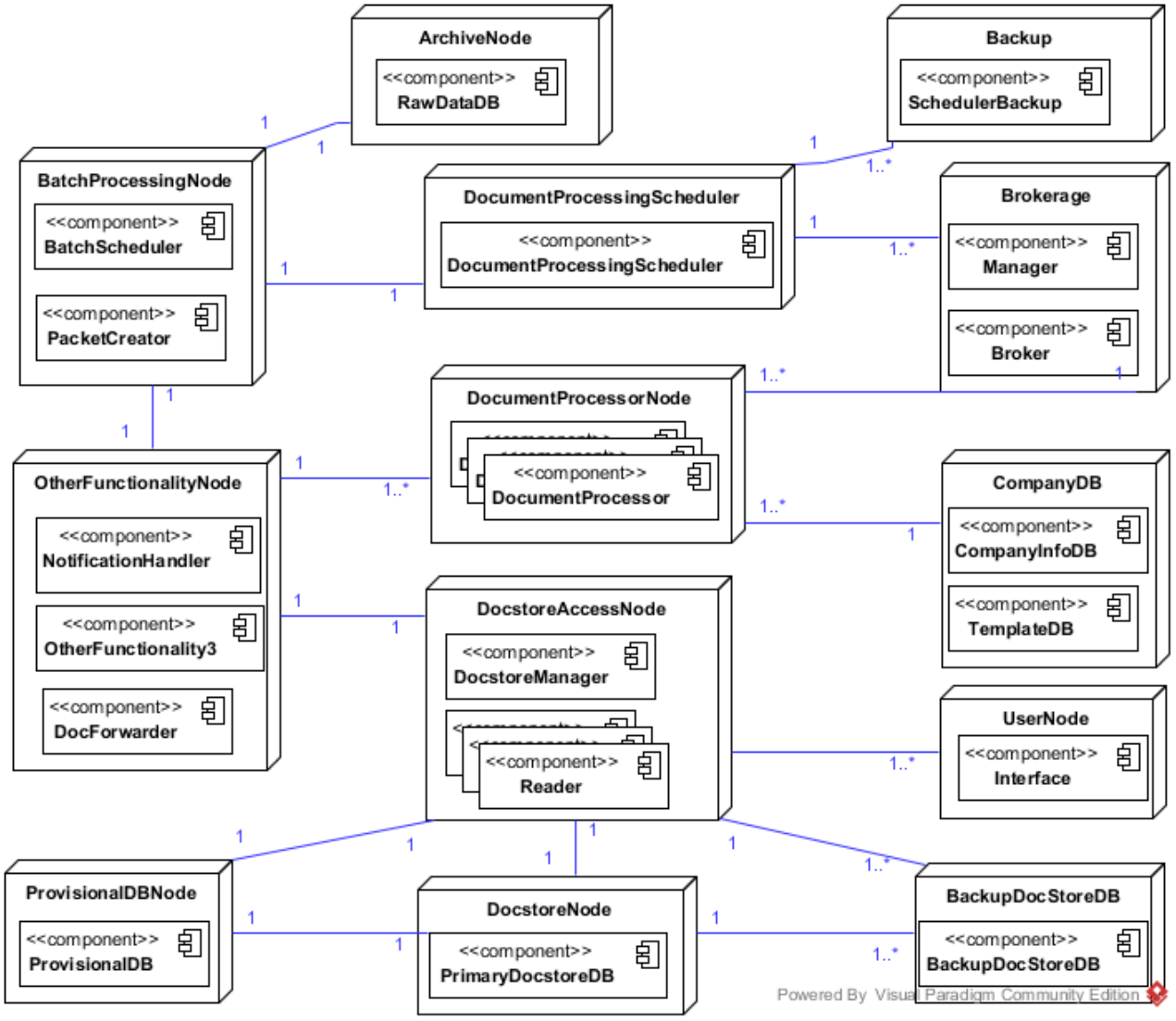


Figure 13: Partial deployment diagram