Horus
IMSETY
Architectural Design Document
Version 0.3 14th May 2007

Project team: Jeroen Keiren 0569081
Frank Koenders 0575629
Thijs Nugteren 0574426
Joeri de Ruiter 0578312
Stijn Stiefelhagen 0579816
Carst Tankink 0569954
Pim Vullers 0575766
Freek van Walderveen 0566348

Project manager: Egbert Teeselink

Senior management: L. Somers TU/e (HG 7.83)
M. v.d. Brand TU/e (HG 7.44)

Adviser: R.J. Bril TU/e (HG 5.09)
Customer: E. v. Breukelen ISIS

Computer Science, Eindhoven University of Technology, Eindhoven
Abstract

This document contains the architectural design for IMSETY. Software components, their interfaces and connections to other software components are identified and described. The architectural design satisfies the software requirements as found in the SRD [3]. Communication of the software with external systems is described in the IPD [2].
Contents

1 Introduction ............................................................................................................. 1
  1.1 Purpose ............................................................................................................. 1
  1.2 Scope .............................................................................................................. 1
  1.3 List of definitions ............................................................................................. 1
  1.4 List of references ............................................................................................. 2
  1.5 Overview ......................................................................................................... 2

2 System overview ..................................................................................................... 3
  2.1 High-level design ............................................................................................. 3
    2.1.1 Monolithic architecture ............................................................................. 3
    2.1.2 Client-server architecture ......................................................................... 4
    2.1.3 Conclusions ............................................................................................... 5

3 System context ......................................................................................................... 6

4 System design .......................................................................................................... 7
  4.1 Design method ................................................................................................. 7
  4.2 Decomposition description .............................................................................. 7
    4.2.1 Server ......................................................................................................... 7
    4.2.2 Client .......................................................................................................... 9

5 Component descriptions ........................................................................................... 13
  5.1 Common components ...................................................................................... 13
    5.1.1 Administration .......................................................................................... 13
  5.2 Server components ............................................................................................ 15
    5.2.1 Queue manager .......................................................................................... 15
    5.2.2 Scheduler ................................................................................................... 16
    5.2.3 Internal queue ............................................................................................ 18
    5.2.4 Internal upload queue ................................................................................ 18
    5.2.5 Internal download queue ............................................................................ 19
    5.2.6 External queue ........................................................................................... 19
    5.2.7 External upload queue ................................................................................ 20
    5.2.8 Client connectivity ..................................................................................... 20
    5.2.9 Controller ................................................................................................... 21
    5.2.10 Bookings ................................................................................................... 23
    5.2.11 Booking .................................................................................................... 24
    5.2.12 MCC controller ......................................................................................... 25
    5.2.13 MCC connectivity ..................................................................................... 26
    5.2.14 MCS controller ......................................................................................... 27
    5.2.15 MCS connectivity ..................................................................................... 29
    5.2.16 Database abstraction ................................................................................ 30
  5.3 Client components .............................................................................................. 31
## CONTENTS

5.3.1 Graphical User Interface ........................................... 31  
5.3.2 Schedule experiment .................................................. 31  
5.3.3 Observation & Intervention ......................................... 32  
5.3.4 InterventionController ............................................... 34  
5.3.5 ObservationData ....................................................... 36  
5.3.6 Loader ................................................................. 37  
5.3.7 RepresenterInterface .................................................. 38  
5.3.8 Processor ............................................................. 39  
5.3.9 Server connectivity ..................................................... 39  
5.3.10 File system abstraction .............................................. 41  

6 Scenarios ................................................................. 43  
6.1 Server scenarios .......................................................... 43  
6.1.1 Uploading an experiment .............................................. 43  
6.1.2 Requesting a list of opportunities .................................... 45  
6.1.3 Requesting a booking .................................................. 46  
6.1.4 Communication in an internal booking ............................... 47  
6.1.5 Real time communication ............................................. 48  
6.2 Client scenarios .......................................................... 51  
6.2.1 Postprocessing data .................................................... 51  
6.2.2 Other scenarios ........................................................ 51  

7 Feasibility and resource estimates ...................................... 53  
7.1 Resources to build the system ......................................... 53  
7.2 Resources to operate the system ....................................... 53  
7.2.1 Server requirements .................................................. 53  
7.2.2 Client requirements .................................................. 53  

8 Requirements traceability matrix ...................................... 54  
8.1 Software requirements to architectural design .......................... 54  
8.2 Architectural design to software requirements ........................ 56
Document status sheet

<table>
<thead>
<tr>
<th>Document title</th>
<th>Architectural Design Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document identifier</td>
<td>IMSETY/doc/ADD</td>
</tr>
<tr>
<td>Author(s)</td>
<td>Jeroen Keiren, Frank Koenders, Thijs Nugteren, Joeri de Ruiter, Stijn Stiefelhagen, Carst Tankink, Pim Vullers, Freek van Walderveen</td>
</tr>
<tr>
<td>Version</td>
<td>0.3</td>
</tr>
<tr>
<td>Document status</td>
<td>Conditionally approved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 (Revision 708)</td>
<td>10-04-2007</td>
<td>Jeroen Keiren</td>
<td>Created document</td>
</tr>
<tr>
<td>0.1 (Revision 1110)</td>
<td>08-05-2007</td>
<td>Jeroen Keiren, Frank Koenders, Thijs Nugteren, Joeri de Ruiter, Stijn Stiefelhagen, Carst Tankink, Freek van Walderveen</td>
<td>Up for internal review</td>
</tr>
<tr>
<td>0.2 (Revision 1131)</td>
<td>09-05-2007</td>
<td>Jeroen Keiren, Frank Koenders, Thijs Nugteren, Joeri de Ruiter, Stijn Stiefelhagen, Carst Tankink, Pim Vullers, Freek van Walderveen</td>
<td>Up for external review</td>
</tr>
<tr>
<td>0.3 (Revision 1177)</td>
<td>14-05-2007</td>
<td>Jeroen Keiren, Frank Koenders, Thijs Nugteren, Joeri de Ruiter, Stijn Stiefelhagen, Carst Tankink, Pim Vullers, Freek van Walderveen</td>
<td>Fixed remarks from adviser</td>
</tr>
</tbody>
</table>
### Document change report

<table>
<thead>
<tr>
<th>Document title</th>
<th>Architectural Design Document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document identifier</td>
<td>IMSETY/doc/ADD</td>
</tr>
<tr>
<td>Date of changes</td>
<td>11-05-2007 - 14-05-2007</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section number</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Almost all</td>
<td>Fixed external review remarks</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Added processing overview</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Added file system abstraction</td>
</tr>
<tr>
<td>5.3.10</td>
<td>Added file system abstraction</td>
</tr>
<tr>
<td>6</td>
<td>Added references in MSCs to other MSCs</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to identify and describe software components, their interfaces and connections to other software components while conforming to the software requirements as stated in the SRD [3]. In order to do this the high-level design is determined and the components of this design are themselves divided into components. By specifying the interfaces and dependencies of these components, programmers will be able to work independently and in parallel on the detailed design of individual components in the next phase of the project. A part of the design of IMSETY; the interfaces, protocols and techniques used for communication with external systems, are described in the Interface and Protocol Definition (IPD) [2]. The parts of that document which deal with the design of interfaces and protocols should be seen as an integral part of this document.

1.2 Scope

IMSETY will provide scientists and observers with an easy to use program interface to conduct experiments on space based samples and possible reference samples on Earth.

1.3 List of definitions

This section contains the definitions of all used terms, acronyms and abbreviations in this document.

ADD  Architectural Design Document
DBMS  Database Management System
ESA   European Space Agency
GENSO Global Educational Network for Satellite Operations
MCC   Mission Control Client, see the URD [4] for more information.
MCCC  Mission Control Client Controller
MCS   Mission Control Software, see the URD [4] for more information.
MCSC  Mission Control Software Connectivity
OMT   Object Modeling Technique
SRD   Software Requirements Document
UML   Unified Modeling Language
URD   User Requirements Document
1.4 List of references


1.5 Overview

The remainder of this document is divided into the following chapters, as dictated by the ESA Software Engineering Standards [1]: System overview (chapter 2) describes the highest level of design for the system. System context (chapter 3) describes the protocols used to communicate with external systems. The System design (chapter 4) consists of a decomposition of the system into components, each of which solves a sub problem of IMSETY’s main problem. Component descriptions (chapter 5) describe these components. The Scenarios (chapter 6) describe how components work together in complex scenarios. In chapter 7 we give resource estimates for IMSETY, describing the systems needed to build or run the system. The Requirements traceability matrix in chapter 8 links the software requirements found in the SRD [3] to the components defined in this document and vice versa.
Chapter 2

System overview

As described in the URD [4] and the SRD [3] IMSETY shall enable scientists to monitor experiments running on a satellite. In order to facilitate this, the need for a central system, controlling the communication with the systems enabling satellite communication, is evident. In this chapter we elaborate on the various architectures that implement this desire, and based on the further requirements of the system we decide upon a high-level architecture.

2.1 High-level design

Looking at IMSETY, one can identify two major problems the system should solve. First, it should provide an abstraction from the technical difficulties involved in communicating with a satellite. This communication is done through MCSes and MCCs and the difficulties involved are handling both real-time requests and queued data transfers. The second problem is the fact that the processing of data gathered from satellite experiments should be accommodated by user-developed plug-ins. This could lead to a wide variety of different data processing subsystems, none of which is guaranteed to work correctly or with a-priori resource constraints. This is a problem since in an ideal case, ISIS should not be required to predict the necessities of data processing in years to come and, in reaction to that, purchase systems capable of fulfilling these necessities.

In order to cope with these problems, we consider two top level architectures: a monolithic architecture and a client-server architecture following the guidelines described by Lethbridge and Laganière [5]. We will describe the application of these architectures to our problem in the following sections. Finally, we will conclude which architecture we shall use to decompose the system.

2.1.1 Monolithic architecture

When considering a monolithic architecture, we do not physically separate the functionality of IMSETY. This means that we will develop one single application, possibly accompanied with a database, that will handle both the communications with the satellite and the processing of data received from the satellite. The main advantage of such a system is that maintenance only affects one system, and there is no need to keep backwards compatibility with older versions of the system, excluding possible plug-ins. A second advantage is that during development, one does not need to provide a way to connect between components that are physically separated.

A monolithic system has to run on a system provided and maintained by ISIS, since the MCSes and the MCCs will probably be physically located with ISIS. To illustrate this necessity, consider for a moment a distribution of IMSETY in which each user has his own copy of the system running locally. In such a distribution, each local instance needs to connect with the MCS and the MCC running at ISIS. This means that these connections can also be manipulated by the end user, for example to gain access to parts of the satellite to which they are not authorized. To counter this, some software should be placed with ISIS checking what information is coming in. But with the
web technologies of these days, it is just as easy to provide a full-fledged user interface that is running on the server and showing through a web browser, so we can place all application logic with this checker, hence placing the complete, monolithic program with ISIS.

Now, assuming that we have a system running, providing an interface to the end-user, another problem arises: to observe the data gathered by the payload’s sensors in some sensible way, a scientist needs to process it. Since it is unclear what kind of data the scientist might need and what processing options she needs, IMSETY provides a plug-in system, allowing end-users to create processors that serve their particular needs. These plug-ins have two inherent problems. Firstly, as described before, it is unknown how much resources these processes use and if the resource cost increases, ISIS needs to pay more to keep the system usable. Secondly, there is no control on the plug-ins, meaning the creator of a plug-in could potentially write a plug-in executing harmful code instead of processing data. This risk calls for a moderation system of sorts, placing an extra burden of maintenance to ISIS’s responsibilities. This implies that a monolithic architecture does not solve the second problem in a satisfying way.

To summarize the pros:

• less problems with developing communication between physically separated components,
• provides an abstraction of communication with MCS and MCC,
• little difficulties with maintenance,

and cons:

• does not solve plug-in problem in a satisfying way.

2.1.2 Client-server architecture

Using the client-server architecture has several advantages [5]. The ones important for IMSETY are:

• distributed computation. Since the plug-ins can execute the computations necessary for processing the data on the client, this relieves the burden of the computation from the server,
• separation of design. By dividing the functionality among the client and the server, Horus can be divided into two teams, one working on the server, the other on the client. This can speed up the design process (and the implementation of the design) considerably, since several parts can be done in parallel,
• simultaneous access to data. To allow observers to watch experiment data, this data needs to be somewhere central. In the client-server architecture, this data can be stored in the server and viewed by the clients.

In this architecture, the server provides an abstraction from the MCC and the MCS interfaces while not introducing security issues in the communication with these external system. When this is the case, the client handles the plug-ins and the processing of the data, which needs to be retrieved from the server.

To administrate the system, data about satellites and accounts needs to be stored in the server. To avoid development overhead, a front end for this administration can be provided in the same client.

The main disadvantage of a client-server architecture is one of maintenance. Should the server be updated, it might not work anymore with clients compatible with previous versions of the server. This means that great care should be taken in maintenance of the server and the client.

A diagram of this architecture is provided in 2.1.
2.1.3 Conclusions

From the two sections above, we conclude that the client-server is a better solution for the solving of IMSETY’s problems than a monolithic architecture. The main argument for this decomposition is that the monolithic architecture does not solve the problem concerning the plug-ins. The architecture itself could provide this functionality, but due to the unknown resource constraints, the need to upgrade ISIS’s systems more often than is acceptable always looms on the horizon. The only other solution to this is moderating the plug-ins, but this would burden ISIS personnel with the tasks of maintenance.

Looking at the client-server architecture, the only real disadvantage is the possibility for discrepancies caused by maintenance on either the client or the server. The main precaution to be taken here is that such updates need to comply with the former versions. This means that the services offered by the server should always include those services provided in a previous version, and can only extend on those services. That way, future clients can use the enhanced functionality, but the older versions can still use the newer version. Together with that comes the huge advantage of being able to develop the server and the client in parallel, which speeds up the time-constrained development process.
Chapter 3

System context

The external interfaces are described in the SRD [3], appendix B and the IPD [2].
Chapter 4

System design

4.1 Design method

The method used to design the component models is the Object Oriented design method OMT. We use OMT in order to be able to make a natural decomposition of the system into components. This will also enable us to implement the system in a controlled manner. The presented models are decompositions of the high level architecture into components and are made with UML.

4.2 Decomposition description

4.2.1 Server

High-level decomposition

The model for the server is presented in figure 4.1. We will here discuss the model in more detail, starting from the lower layer.

When we look at the server, there are some components that we can easily identify. These are the components that occur at the lowest level of the system. This layer of the server comprises a component for database abstraction. This way we can provide more flexibility with respect to the used database management system (DBMS). Connectivity to the MCC and MCS is also found on this layer. These components are responsible for providing an abstraction layer over the actual protocol implementation. This way it can be made sure that changes in the protocol will only have to be incorporated locally in these components, as long as the alternative protocols provide the same level of functionality.

One level up from the database abstraction is an administration package with classes that represent the entities in the database. Using these classes, one can manipulate the objects in the database, and store the changes back to the database. Furthermore, on top of the booking, MCC and MCS components we find the queue component. This component is responsible for requesting bookings with the MCC, as well as for sending data to and receiving data from the satellite during a window of opportunity. Requesting of a booking can be done from outside the queue component, in which case the answer is forwarded to the requesting component. Note that bookings that have been done from the outside of the server, which are typically bookings that have been done in order to provide real-time communication, will not be used by the queue component to passively upload or download data. In the sequel we will refer to these bookings as external bookings. Mostly however, the bookings will be done in order to be able to upload or download data, experiments, or measurements. In this case the queue component will request the required bookings and use them for the previously described communication. Note that in order to facilitate this separation between real-time and other communication we identify two queues, one for each purpose.

On top of the queue component, and the set of low-level components that have been already described we find a controller component. This component is responsible for providing authenti-
CHAPTER 4. SYSTEM DESIGN

Figure 4.1: A decomposition of the server

cation, as well as checking whether a user has appropriate rights to access the requested material. Furthermore this component delegates incoming requests from the clients to the other components, and collects the reactions. This way, also functionality that requires information from multiple other components can easily be provided. Another important aspect of the controller component is that it abstracts from the underlying functionality, and provides just the required functionality to the clients. In this way the controller component is a façade. There is one controller component, which is responsible for all client connections.

Stacked onto the controller component there is a component that provides the connectivity with clients. The only thing this component does is translate the communication between clients and server from and to the actual protocol, and provide the server side of the system with an internal abstraction of the protocol. This way, like in the MCC and MCS cases, an abstraction of the protocol is provided, and a change in protocol would require only a local change in the client connectivity component.
CHAPTER 4. SYSTEM DESIGN

Processing overview

Now that we have described a division of the server into various components, one can wonder what the executable structure of the server looks like. Therefore, in this section we provide a description of the various processes and threads that we identify in the server. First of all there is a process for the controller of the server which makes use of the administration package and database connectivity. Also, for the client connectivity, per client a thread is started in order to provide this connectivity. A separate thread is created for the scheduler in order to make sure that other activities in the system do not block the scheduling and queuing of communication with the satellite, and the control of this communication. This thread also creates threads for controlling the connections to various MCCs and MCSes. To facilitate communication with the MCCs and MCSes, for each MCC and MCS which needs a connection, a thread for the respective connectivity component is created. This way we can easily facilitate simultaneous connections to various MCSes and MCCs.

4.2.2 Client

High-level decomposition

As described in section 2.1, the main functionality of the client is divided into providing a front end for administration and providing a means to process experiments and intervene in their execution, as well as managing the plug-ins providing this processing. Keeping this separation in functionality in mind, we decompose the functional logic of the client in the same two packages: Administration and Experiment. The administration package’s functionality is similar to that of the package of the same name in the server component. The difference lies in the fact that the package in the client is used for managing entities on the server and is therefore dependent on the server connectivity instead of the database connectivity. However, this package can easily be shaped to a form which is suitable for both the server and the client, as we will do in chapter 5.

These two packages would be enough, were the client a stand-alone application not required to communicate with either a server or a user. Since this is not the case, we provide two other packages, the Graphical User Interface and the Server connectivity. The Graphical User Interface (GUI) provides an abstraction and a visualization from the functionality provided by the Administration and Experiment packages. The basis for the GUI is the prototype belonging to the SRD 3. The Server connectivity package provides an abstraction from the physical communication with the server. The functions it provides are described in chapter 5.

A diagram of the client’s decomposition is given in figure 4.2.
Figure 4.2: A decomposition of the client
CHAPTER 4. SYSTEM DESIGN

Experiment

The main responsibility of the experiment package is the scheduling of experiments and the observation of experiment data, including real time intervention of running experiments. The observation of data involves a plug-in system which is not trivial, so we split this functionality in two packages, in order to cope with the plug-ins without the overhead of experiment management. The functionality provided by the Schedule package is trivial. In the next section, we shall provide a decomposition of the Observation & Intervention package.

Observation & Intervention

One of the most complicated packages of the client is decomposed in figure 4.3. This Observation & Intervention package is therefore described on a very low level in chapter 5.3.3. For now, we note that this package solves the following problems:

- provide a way to dynamically load plug-ins,
- allow the client to dispatch newly created data to the correct plug-ins, for either visualization or post processing,
- give a means to batch process data
CHAPTER 4. SYSTEM DESIGN

Observation & Intervention

InterventionController
DataPublisher
-observers: [(Type, Observer)]
+addObserver(t: Type)
+notifyObservers(t: Type, d: ObservationData)

ObservationData
-data: Data
-type: Type
+getType(): Type
+getData(): Data

DataReader
+update(d: ObservationData)

ProcessorBatch
-queue: [ProcessorId] 1..* 0..*
+addProcessor(p: ProcessorId)
+process(d: Data, t: Type)

ProcessorInterface
-inType: Type
-outType: Type
+process(d: Data): Data
+configure(p: Parameters)

RepresenterInterface
-inType: Type
+represent(ui: GUI)
+configure(p: Parameters)

Representer

Processor

Loader

FileSystemAbstraction
+store(d: Data, location: URL)

Figure 4.3: A decomposition of the Observation & Intervention package
Chapter 5

Component descriptions

This chapter describes, per component, the interfaces, as well as a compact description of the components functionality. For the pre- and post conditions in the functions of the described interfaces use is made of the associations from the logical model as found in the SRD [3]. Note that in the cases where the sections “Resources”, “References”, “Processing” and “Data” are fully trivial or empty we omit them. Fully trivial can for example include disk space, processor time and internal memory that is not beyond the requirements of a modern date personal computer.

We first describe components that are common between the client and the server. Then we describe all components for the server, and we conclude with a description of the client components.

5.1 Common components

5.1.1 Administration

Type
Package
Purpose
Implements software requirement(s) SRFUR1, SRFUR2, SRFUR3, SRFUR4, SRFUR5, SRFUR6, SRFUR7, SRFUR8, SRFUR9, SRFUR10, SRFUR11, SRFUR13, SRFUR14, SRFUR15, SRFUR16, SRFUR17, SRFUR18, SRFUR19, SRFUR20, SRFUR21, SRFUR22, SRFUR23, SRFUR24, SRFUR25, SRFUR26, SRFUR27, SRFUR28, SRFUR29, SRFUR30, SRFUR33, SRFUR34, SRFUR42, SRFUR43, SRFUR44, SRFUR45, SRFUR46, SRFUR47, SRFUR51, SRFUR52, SRFUR62, SRFUR63, SRFUR67, SRFUR68, SRFUR69, SRFUR70, SRFUR71, SRFUR85, SRFUR92, SRFUR98, SRFUR99.

Function
This package comprises the several administration components in the system. It is used both by server and client.

Subordinates
The following classes are included in this package:

- Account
- Experiment
- Data
Dependencies
This package depends on services provided by the Server Connectivity package on the client side and on the Database abstraction on the server side of the system.

Provides interfaces
The individual classes provide interfaces based on the specific requirements in the SRD [3].

The package itself provides three interfaces, two interfaces for the server and one for the client.

To the server it provides the following:

- an interface to the controller,
- an interface to the queue manager

and to the client:

- an interface to the graphical user interface.

Requires interfaces
The classes in this packages require several functions from the Server connectivity package on the client side and from the Database querying package on the server side of the system.

Account

1. `getAccountParams(name: string): parameters`
   To retrieve the parameters of the account with name `name`.

2. `getType(name: string): string`
   To retrieve the type of the account with name `name`.

3. `hasRightsTo(name:string, p:payloadId): Boolean`
   To check if the user with name `name` has rights to the payload with identifier `p`.

4. `hasRightsTo(name:string, e:experimentId): Boolean`
   To check if the user with name `name` has right to the experiment with identifier `e`.

Experiment

1. `getExperimentParams(p: payloadId, name: string): parameters`
   To retrieve the parameters of the experiment with name `name` on the payload with identifier `p`.

Payload

1. `getPayloadParams(p: payloadId): parameters`
   To retrieve the parameters of the payload with identifier `p`.

Satellite
CHAPTER 5. COMPONENT DESCRIPTIONS

1. `getSatelliteParams(s: satelliteId): parameters`
   To retrieve the parameters of the payload with identifier `s`.

Log

1. `retrieveLog(f: filter): Log`
   To retrieve the logs which pass filter `f`.

References

The description of the specific requirements, on which the separate classes are based, can be found in the SRD [3].

Data

The data which is needed to support the individual classes can be derived from the specific requirements in the SRD [3].

5.2 Server components

5.2.1 Queue manager

Type

Package

Purpose

Implements software requirement(s) SRFUR36.

Function

This package is responsible for scheduling and distributing the communication with the MCSes and MCCs. That means that the queue manager distributes tasks to a scheduler about tasks that have to be carried out, and that it keeps queues for real-time and other communication. In case new data is available, the queue manager will notify the controller which will in his turn provide subscribers with this data.

Subordinates

Scheduler, External queue, Internal queue

Dependencies

The queue manager depends on the administration package, bookings, the MCC controller and the MCS controller.

Provides interfaces

The requires interfaces of this component consists of the requires interfaces of the scheduler component.

Requires interfaces

This component requires the interfaces of the components on which it depends.
5.2.2 Scheduler

**Type**
Thread

**Purpose**
Implements software requirement(s) SRFUR35.

**Function**
This thread is responsible for the scheduling of downloads and uploads. It will use a ‘First Come First Serve’ policy to handle all the download and upload requests. The scheduler distinguishes internal and external bookings. An internal booking is a booking that will only be used to upload experiments and to download data. Besides internal bookings, the scheduler also determines external bookings, also called real time bookings, which will only be used to download thumbnails in case of real time intervention.

**Subordinates**
None

**Dependencies**
This component depends on the internal queue package and the external queue package.

**Provides interfaces**

- `notifyBookingTimeout(bId : BookingId, t : TimeoutMessage)`
  
  **Description:** Receives a booking timeout for the booking with bookingId `b` with timeout message `t`.
  
  **Pre:** `(\exists b \in Bookings :: b.id = bId)`
  
  **Post:** Timeout `t` is logged and reported to the controller.

- `callbackMCS(s : Satellite, i : BookingId, j : Status)`
  
  **Description:** Pass on the received status `j` for request `i` from satellite `s`.
  
  **Pre:** `true`
  
  **Post:** The status `s` is passed on.

- `notifyNewData(s : Satellite, d : Data)`
  
  **Description:** Receives new data `d` for the satellite with identifier `s`.
  
  **Pre:** `(\exists s' \in Satellites :: s' = s)`
  
  **Post:** The scheduler updates its queues and sends data `d` to the controller.

- `uploadExperiment(eId : ExperimentId)`
  
  **Description:** Uploads experiment `e`.
  
  **Pre:** `(\exists e \in Experiments :: e.id = eId)`
  
  **Post:** Experiment `e` is added to the internal upload queue.

- `downloadHighResolutionData(D : dataId)`
  
  **Description:** Downloads the high-resolution data of the thumbnail with data identifier `d`.
  
  **Pre:** Identifier `d` is the data identifier of a thumbnail that exists in the database.
  
  **Post:** The high resolution data of the thumbnail with data identifier `d` is added to the internal download queue.
CHAPTER 5. COMPONENT DESCRIPTIONS

deleteData(d : DataId)
Description: Deletes the data with identifier d from the satellite from which it was retrieved.
Pre: There exists a data element in the database with data identifier d.
Post: The delete command w.r.t the data with identifier d is added to the internal upload queue.

executeCommand(s : Satellite, p : PayloadId, c : String)
Description: Executes the command c on payload p on satellite s.
Pre: true
Post: Command c is added to the external upload queue.

isWindowOfOpportunity(pId : PayloadId, ti : Time) : Boolean
Description: Returns whether there is a window of opportunity for the payload with payload identifier pId on time ti.
Pre: (∃p ∈ Payloads :: p.id = pId)
Return: (∃b ∈ Booking :: b.Start ≤ ti ≤ b.End)

Requires interfaces

callbackList(s: Satellite, i: BookingId, l: ListOfOpportunities, j: (success, failed, rejected))
Description: Passes on the received list of opportunities l with status j for request i from satellite s
Pre: true
Post: The list of opportunities l with status j for request i from satellite s is passed on

callbackBooking(s: Satellite, b: BookingInfo, j: (success, failed, rejected))
Description: Passes on the received booking information b from satellite s if the status j is success
Pre: true
Post: The booking information b from satellite s is passed on if the status j is success.
The status of the corresponding booking object in the database

callbackMCS(s: Satellite, i: BookingId, j: (success, failed, rejected))
Description: Pass on the received status j for request i from satellite s
Pre: true
Post: The status s is passed on

notifyNewData(s: Satellite, d: Data)
Description: Receives new data d for the satellite with identifier s.
Pre: (∃s' ∈ Satellites :: s' = s)
Post: The scheduler updates its queues and sends data d to the controller.

Processing
For the internal control and data flow, see chapter 6.

Data
The internal data of the scheduler consists of the internal upload queue, the internal download queue and the external upload queue.
5.2.3 Internal queue

**Type**
Package

**Purpose**
Implements software requirement(s) SRFUR37, SRFUR38.

**Function**
This package contains the internal upload queue and the internal download queue which will be used to keep track of data that needs to be uploaded to or downloaded from a satellite. The scheduler will assign a higher priority to internal uploads than to internal downloads, because it is more important that an experiment will be uploaded to the satellite than that high-resolution data will be downloaded.

**Subordinates**
Internal upload queue, internal download queue.

**Dependencies**
None

**Provides interfaces**
This package provides the functionality of its subordinates.

**Requires interfaces**
None

5.2.4 Internal upload queue

**Type**
Class

**Purpose**
Implements software requirement(s) SRFUR37, SRFUR38.

**Function**
This class provides several functions to the scheduler with respect to the non-real time uploading of data.

**Subordinates**
None

**Dependencies**
None
CHAPTER 5. COMPONENT DESCRIPTIONS

Provides interfaces
This class provides functionality to manage the internal upload queue. The exact functions will be specified during the detailed design phase.

Requires interfaces
None

5.2.5 Internal download queue
Type
Class

Purpose
Implements software requirement(s) SRFUR37, SRFUR38.

Function
This class provides several functions to the scheduler with respect to the non-real time downloading of data.

Subordinates
None

Dependencies
None

Provides interfaces
This class provides functionality to manage the internal download queue. The exact functions will be specified during the detailed design phase.

Requires interfaces
None

5.2.6 External queue
Type
Package

Purpose
Implements software requirement(s) SRFUR37, SRFUR38.

Function
This package contains the external upload queue which will be used to keep track of data that needs to be uploaded to a satellite during a window of opportunity for which a booking must be arranged.
CHAPTER 5. COMPONENT DESCRIPTIONS

Subordinates
External upload queue

Dependencies
None

Provides interfaces
This package provides the functionality of its subordinate.

Requires interfaces
None

5.2.7 External upload queue

Type
Class

Purpose
Implements software requirement(s) SRFUR37, SRFUR38.

Function
This class provides several functions to the scheduler with respect to the real time uploading of data.

Subordinates
None

Dependencies
None

Provides interfaces
This class provides functionality to manage the external upload queue. The exact functions will be specified during the detailed design phase.

Requires interfaces
None

5.2.8 Client connectivity

Type
Thread

Purpose
Implements software requirement(s) None.
CHAPTER 5. COMPONENT DESCRIPTIONS

Function

This thread provides a networking interface which is used by a specific client in order to retrieve the information stored by the server. Furthermore the component provides an abstraction of the actual protocol to the controller component of the server. This means that there is one client connectivity thread per client.

Subordinates

None

Dependencies

Client connectivity depends on the controller for handling the requests done here.

Provides interfaces

This component provides the interfaces required by the client (in the client connectivity component).

Requires interfaces

This component requires the full provides interface of the controller component.

5.2.9 Controller

Type

Process

Purpose

Implements software requirement(s) SRFUR86, SRFUR88, SRFUR90, SRFUR91, SRFUR97, SRFUR100, SRFUR102, SRFUR103, SREFR8.

Function

The controller is the main process of the server. It distributes incoming requests from the client to the appropriate subordinates. While doing so, it does also handle and enforce the access rights, such that a user cannot access data he does not have the rights to access. This component is able to receive callbacks from the queue manager, and it can send callbacks to the client on events the clients have subscribed to. For communicating with the clients, a client connectivity thread is started for each active client.

Subordinates

None

Dependencies

Administration package, Queue manager
CHAPTER 5. COMPONENT DESCRIPTIONS

Provides interfaces

**notifyNewData()**
*Description:* Callback handle which should be called when the queue has new data
*Pre:* true
*Post:* All clients registered have received notification of the data

**getListOfCommands(p: PayloadId, n:String)**
*Description:* Retrieve all commands applicable to the payload
*Pre:* true
*Return:* All commands applicable to the payload

**getExperimentParams(p: PayloadId, n:String)**
*Description:* Retrieve all parameters for the experiment
*Pre:* true
*Return:* All parameters for the experiment with name \( n \) on the payload with id \( p \)

**getAccountParams(name:String)**
*Description:* Retrieve all parameters for the account
*Pre:* true
*Return:* All parameters for the account with name \( name \)

**getType(name:String)**
*Description:* Retrieve the type of the account
*Pre:* true
*Return:* The type of the account with name \( name \)

**getPayloadParams(id: PayloadId)**
*Description:* Retrieve the parameters of the payload
*Pre:* true
*Return:* The parameters of the payload with id \( id \)

**getSatelliteParams(id: SatelliteId)**
*Description:* Retrieve the parameters of the satellite
*Pre:* true
*Return:* The parameters of the satellite with id \( id \)

**markForDownload(d: DataId)**
*Description:* Marks the data item for download for retrieving high resolution data
*Pre:* true
*Post:* \( d \) is marked for download

**markForDeletion(d: DataId)**
*Description:* Marks the data item for deletion
*Pre:* true
*Post:* \( d \) is marked such that it can be deleted from the satellite

**queueForUpload(p:PayloadId, c:Command)**
*Description:* Queue the command for upload
*Pre:* true
*Return:* Command \( c \) has been queued to be uploaded to payload \( p \)
isWindowOfOpportunity(p:PayloadId, ti:Time)
Description: Returns whether there is a window of opportunity for payload p in which ti is included
Pre: true
Return: true if ti lies within a window of opportunity booked for payload p

getListOfCommands(p:PayloadId)
Description: Returns the list of commands defined on the payload
Pre: true
Return: The list of commands defined on the payload with id p

retrieveLog(f: Filter)
Description: Returns the log entries satisfying the filter
Pre: true
Return: The log entries satisfying filter f

hasRightsTo(name: String, p:PayloadId)
Description: Returns whether the user has rights to the payload
Pre: true
Return: The user with name name has rights to access the payload with id p

hasRightsTo(name: String, e:ExperimentId)
Description: Returns whether the user has rights to the experiment
Pre: true
Return: The user with name name has rights to access the experiment with id e

Requires interfaces
The controller requires precisely the interfaces as they are provided by the administration package and the queue manager.

5.2.10 Bookings
Type
Package

Purpose
Implements software requirement(s) SRFUR39, SRFUR40, SRFUR41.

Function
This package provides the functionality of the Booking class, extended with a function which gets the list of known bookings from the system.

Subordinates
Booking

Dependencies
None
CHAPTER 5. COMPONENT DESCRIPTIONS

Provides interfaces

getListOfBookings():[Booking]
Description: Returns the list of all known bookings in the system
Pre: true
Return: [b : db.Booking]true

Furthermore this component provides all functionality of the Booking component.

Requires interfaces

This component requires all interfaces of the Booking component.

5.2.11 Booking

Type

Class

Purpose

Implements software requirement(s) SRFUR39, SRFUR40, SRFUR41.

Function

This class describes a booking as it is represented by the system.

Subordinates

• Type : enum(one-way, two-way)
• Source : enum(internal, external)
• Status : enum(created, pending, cancelled, accepted, running, finished)

Dependencies

None

Provides interfaces

create(b : bookingId)
Description: Instantiates a booking object from the database with bookingId b.
Pre: (∃b' ∈ Booking :: b' = b)
Post: Booking object with bookingId b is instantiated

create(ty: Type, so: Source, sa : Satellite, ts: Time, te: Time, st : Status)
Description: By making an instance of this class, there is a booking starting at time ts and ending at time te of type ty for satellite sa with a new unique bookingId id and status st. Source so indicates whether this booking is internal or external.
Pre: true
Post: this.type = ty ∧ this.start = ts ∧ this.end = te ∧ this.satellite = sa ∧ this.source = so ∧ this.status = st ∧ this.bookingId = id ∧ ¬(∃b' ∈ Booking : b' ≠ this : b'.bookingId = id)
CHAPTER 5. COMPONENT DESCRIPTIONS

retrieve()
Description: Retrieve the information stored in the booking.
Pre: true
Return: (this.bookingId, this.type, this.source, this.satellite, this.start, this.end, this.status)

getInternalBooking(s : satelliteId)
Description: Returns whether there are internal bookings for satellite s
Pre: s.registered = true
Return: [(b.id | b ∈ Booking ∧ b.satelliteId = s ∧ b.source = internal)]

Requires interfaces
This component requires the appropriate database querying from the Database abstraction.

Data
- status : Status
- satellite : Satellite
- source : Source
- start : Time
- end : Time
- type : Type

5.2.12 MCC controller

Type
Thread

Purpose
Implements software requirement(s) SRFUR82, SRFUR83, SRFUR84, SREFR4.

Function
This thread keeps track of the connections to the various MCCs. When a connection to a new MCC is needed, threads are created for this, in order to facilitate multiple simultaneous connections to different MCCs. The MCC controller has a callback function that is used by the MCC connectivity to report the results. The controller has to be provided with a callback function that will be used to pass the results from the MCC connectivity. The MCC controller also maps a BookingId to an Id received from a MCC.

Subordinates
For each active MCC an MCC connectivity thread is used by this component.

Dependencies
MCC connectivity
CHAPTER 5. COMPONENT DESCRIPTIONS

Provides interfaces

\[
\text{callbackList}(s: \text{Satellite}, i: \text{BookingId}, l: \text{ListOfOpportunities}, j: (\text{success, failed, rejected}))
\]

Description: Passes on the received list of opportunities \(l\) with status \(j\) for request \(i\) from satellite \(s\)

Pre: true

Post: The list of opportunities \(l\) with status \(j\) for request \(i\) from satellite \(s\) is passed on

\[
\text{callbackBooking}(s: \text{Satellite}, b: \text{BookingInfo}, j: (\text{success, failed, rejected})
\]

Description: Passes on the received booking information \(b\) from satellite \(s\) if the status \(j\) is success

Pre: true

Post: The booking information \(b\) from satellite \(s\) is passed on if the status \(j\) is success. The status of the corresponding booking object in the database

\[
\text{callbackController}(s: \text{Satellite}, i: \text{Id}, d: \text{RawData})
\]

Description: Callback function used to receive data from a MCC connectivity

Pre: true

Post: The data for satellite \(s\) is received

\[
\text{requestBooking}(s: \text{Satellite}, i: \text{BookingId}, b: \text{Booking}): \text{Boolean}
\]

Description: Requests the booking for satellite \(s\)

Pre: true

Return: Whether the request has been sent successfully to the MCC

\[
\text{getListOfOpportunities}(s: \text{Satellite}, i: \text{BookingId}, t: \text{Period}): \text{Boolean}
\]

Description: Gets the list of possible opportunities for period \(t\) to schedule an experiment for satellite \(s\)

Pre: true

Return: Whether the request is successfully send to the MCC

Requires interfaces

This component requires all functions provided by the MCC connectivity.

5.2.13 MCC connectivity

Type

Thread

Purpose

Implements software requirement(s) SRFUR82, SRFUR83, SRFUR84, SREFR4.

Function

This thread provides the implementation of the actual networking protocol used to communicate with the MCC. This way it provides an abstraction from the actual implementation of the protocol to the rest of the system. It is used to communicate with a specific MCC.
CHAPTER 5. COMPONENT DESCRIPTIONS

Subordinates

None

Dependencies

This component should only be used by the MCC controller.

Provides interfaces

requestBooking(b: Booking): Id

Description: Request a booking b.

Pre: true

Return: The id that is used to identify the request in responses

getListOfOpportunities(t: Period): Id

Description: Gets the list of possible opportunities for time t to schedule an experiment

Pre: true

Return: The id that is used to identify the request in responses

Requires interfaces

callbackController(s: Satellite, d: RawData)

Description: Callback function used to receive data from a MCC connectivity

Pre: true

Post: The data for satellite s is received

5.2.14 MCS controller

Type

Thread

Purpose

Implements software requirement(s) SRFUR72, SRFUR73, SRFUR75, SRFUR76, SRFUR77, SRFUR78, SRFUR79, SRFUR80, SRFUR81.

Function

This thread keeps track of the connections to the various MCSes. When a connection to a new MCS is needed, threads are created for this, in order to facilitate multiple simultaneous connections to different MCSes. The MCS controller has a callback function that is used by the MCS connectivity to report the results. The controller has to be provided with a callback function that will be used the pass the results from the MCS connectivity. The MCS controller also maps a BookingId to a Id received from a MCS.

Subordinates

For each active MCS an MCS connectivity thread is used by this component.

Dependencies

MCS connectivity
CHAPTER 5. COMPONENT DESCRIPTIONS

Provides interfaces

**callbackMCS(s: Satellite, i: BookingId, j: (success, failed, rejected))**

*Description:* Pass on the received status j for request i from satellite s

*Pre:* true
*Post:* The status s is passed on

**notifyNewData(s: Satellite, d: Data)**

*Description:* Receives new data d for the satellite with identifier s.
*Pre:* \((\exists s' \in \text{Satellites} :: s' = s)\)
*Post:* The scheduler updates its queues and sends data d to the controller.

**exec(s: Satellite, i: BookingId, p: PayloadId, c: Command): Boolean**

*Description:* Execute the command c on payload p on satellite s
*Pre:* c is a valid Command
*Return:* Whether the request is successfully send to the MCS

**fetch(s: Satellite, i: BookingId, p: PayloadId, d: DataId): Boolean**

*Description:* Download the data d from the payload p on satellite s
*Pre:* true
*Return:* Whether the request is successfully send to the MCS

**store(s: Satellite, i: BookingId, p: PayloadId, e: ExperimentInfo): Boolean**

*Description:* Upload the experiment e for the payload p on satellite s
*Pre:* true
*Return:* Whether the request is successfully send to the MCS

**erase(s: Satellite, i: BookingId, p: PayloadId, d: DataId): Boolean**

*Description:* Remove the data d from the payload p on satellite s
*Pre:* true
*Return:* Whether the request is successfully send to the MCS

**list(s: Satellite, i: BookingId, p: PayloadId, pt: PathId): Boolean**

*Description:* List the available data for the given path pt on the payload p on satellite s
*Pre:* true
*Return:* Whether the request is successfully send to the MCS

**schedule(s: Satellite, i: BookingId, p: PayloadId, e: ExperimentId, t: Time): Boolean**

*Description:* Schedule the experiment e on the payload p on satellite s to start execution on time t
*Pre:* true
*Return:* Whether the request is successfully send to the MCS

**run(s: Satellite, i: BookingId, p: PayloadId, e: ExperimentId): Boolean**

*Description:* Start the experiment e on the payload p on satellite s
*Pre:* true
*Return:* Whether the request is successfully send to the MCS
CHAPTER 5. COMPONENT DESCRIPTIONS

kill(s: Satellite, i: BookingId, p: PayloadId, e: ExperimentId):
Boolean

Description: Stop the experiment e on the payload p on satellite s
Pre: true
Return: Whether the request is successfully send to the MCS

Requires interfaces
The MCS controller requires all functions provided by the MCS connectivity.

5.2.15 MCS connectivity

Type
Thread

Purpose
Implements software requirement(s) SRFUR72, SRFUR73, SRFUR75, SRFUR76, SRFUR77, SRFUR78, SRFUR79, SRFUR80, SRFUR81, SRFUR5, SRFUR6.

Function
This thread provides the implementation of the actual networking protocol used to communicate with the MCS. This way it provides an abstraction from the actual implementation of the protocol to the rest of the system. One MCS connectivity thread is responsible for the communication with a single MCS.

Subordinates
None

Dependencies
This component should only be used by the MCS controller.

Provides interfaces

exec(p: PayloadId, c: Command): Id
Description: Execute the command c on payload p
Pre: c is a valid Command
Return: The id that is used to identify the request in responses

fetch(p: PayloadId, d: DataId): Id
Description: Download the data d from the payload p
Pre: true
Return: The id that is used to identify the request in responses

store(p: PayloadId, e: ExperimentInfo): Id
Description: Upload the experiment e for the payload p
Pre: true
Return: The id that is used to identify the request in responses

ADD 0.3 29
erase(p: PayloadId, d: DataId): Id
Description: Remove the data d from the payload p
Pre: true
Return: The id that is used to identify the request in responses

list(p: PayloadId, pt: PathId): Id
Description: List the available data for the given path pt on the payload p
Pre: true
Return: The id that is used to identify the request in responses

schedule(p: PayloadId, e: ExperimentId, t: Time): Id
Description: Schedule the experiment e on the payload p to start execution on time t
Pre: true
Return: The id that is used to identify the request in responses

run(p: PayloadId, e: ExperimentId): Id
Description: Start the experiment e on the payload p
Pre: true
Return: The id that is used to identify the request in responses

kill(p: PayloadId, e: ExperimentId): Id
Description: Stop the experiment e on the payload p
Pre: true
Return: The id that is used to identify the request in responses

Requires interfaces

callbackController(s: Satellite, i: Id, d: RawData)
Description: Callback function used to receive data from a MCS connectivity
Pre: true
Post: The data d for request i on satellite s is received

5.2.16 Database abstraction

Type
Class

Purpose
Implements software requirement(s) None.

Function
The Database abstraction class is responsible for delivering a convenient abstraction from the
database implementation details.

Subordinates
None

Dependencies
None
Provides interfaces

This component provides the interfaces required by the *Administration* package.

Requires interfaces

query(q: string)
Description: Executes query *q* on the database from which this class abstracts.
Pre: There is a connection to the database
Return: Result of query *q*

5.3 Client components

5.3.1 Graphical User Interface

Type

Package

Purpose

Implements software requirement(s) SRFUR87, SRFUR89, SRFUR93, SRFUR94, SRFUR95, SRFUR96, SRFUR97, SRFUR101, SRFUR102, SRFUR103, SRFR7.

Function

Provide an interface for all applicable functional requirements, there should be no application logic inside this component.

Subordinates

User interface elements, which will be specified in the detailed design phase.

Dependencies

None

Provides interfaces

We refer to the SRD [3], appendix A for a description of which functions are provided by the user interface.

Requires interfaces

The user interface requires all interfaces provided by the *Administration* and *Experiment* packages.

Resources

In order to use the user interface, the target machine should be running a window system to display the interface.

5.3.2 Schedule experiment

Type

Thread
Purpose

Implements software requirement(s) SRFUR31, SRFUR32.

Function

This component provides functionality for scheduling experiments.

Dependencies

This class depends on services provided by the Server Connectivity package and the Administration component. It provides services to the Graphical User Interface.

Provides interfaces

schedule(start: time, end: time)
Description: A request to schedule the experiment from time start to time end is made.
Pre: true
Post: this.scheduledFrom = start ∧ this.scheduledTo = end

unschedule(start: time, end: time)
Description: A request to unschedule the experiment.
Pre: true
Post: this.scheduledFrom = ⊥ ∧ this.scheduledTo = ⊥

Data

• scheduledFrom : Time
• scheduledTo : Time

5.3.3 Observation & Intervention

Type

Package

Purpose

Implements software requirement(s) SRFUR46, SRFUR47, SRFUR51, SRFUR53, SRFUR54, SRFUR55, SRFUR56, SRFUR57, SRFUR58, SRFUR59, SRFUR60, SRFUR61, SRFUR62, SRFUR63, SRFUR64, SRFUR65, SRFUR66, SRFUR100.

Function

This package allows the client to handle incoming data in a uniform way. To do this, its data handling functionality is described according to the Observer design pattern. The specific instances of this pattern are:

Observable DataPublisher handles this functionality. This class keeps track of observers on certain data. The difference with the pattern is that there is one DataPublisher per experiment and subscribers can subscribe to new instances of specific data types. This is due to the fact that for each change, a new data item is created.

Observer This is handled by the DataReader interface. This interface is basically the same as the Observer interface as described in the pattern. The difference in this instantiation is that on an update, the newly created data is sent through the update function, since the subscribers are interested in the entire data item.
ConcreteObservable This functionality is given by the ProcessorBatch, which represents a queue of Processor plug-ins. Objects of this class receive data to be processed from the server. When an ObservationData object is created, the subscribers to the type of data are informed with the data object.

ConcreteObserver The concrete observers in this instantiation of the pattern is the RepresenterInterface. This class displays data objects of certain types. It is therefore subscribed to these types. If the type of this data is not equal to the first processor in the batch, it is stored in an ObservationData object. If these types are equal, it is processed in a pipes-and-filters manner. The final data is then stored in an ObservationData object.

Subordinates
The subordinates of this package have partly been described in the Function section, above. We will describe the remaining classes in this section.

- The InterventionController class is a class wrapping the functionality of intervening in the experiment while observing. It is described in section 5.3.4.
- The ObservationData class is a wrapper for the physical data created by the ProcessorBatch. Next to the data itself, it contains the type of the data.
- The ProcessorInterface and the Processor plug-in are described in section 5.3.8.
- The RepresenterInterface and the physical Representer plug-ins are used to display items of a certain data type on screen. These plug-ins have a represent function giving them access to the GUI. When data of a specific type is ready, it can then be represented on the User Interface.
- The Loader object has as its the task the loading of plug-ins adhering to the specific interfaces. It is described in 5.3.6

Dependencies
The Observation & Intervention package depends on the Server connectivity package. The GUI package depends on this package.

Provides interfaces
The most important interface provided by this package is process function. This function is called when there is new data received from the server, and this data is then given to the processor batch. Data created by the individual processors in the batch is stored, to prevent any disaster by cause of data loss.

Next to that, several interfaces are provided by its children.

Requires interfaces
This package requires no more interfaces than its children require.

References
The Observer design pattern and the Pipes-and-filters architectural pattern are described by Lethbridge and Laganière [5].
CHAPTER 5. COMPONENT DESCRIPTIONS

Processing

For each experiment observed, new instances of the DataPublisher, and the ProcessorBatch are created. These instances, as well as the separate DataReaders need to run in separate threads, to disallow computation on certain data to block computation on other data.

5.3.4 InterventionController

Type

Class

Purpose

Implements software requirement(s) SRFUR100.

Function

InterventionController allows the retrieving of a list of commands for a payload. When the retrieve is first executed, it gets a list from the server, storing it locally. Subsequent retrieves are executed on the local list. When the list is updated remotely, the local list should be updated.

During a window of opportunity, commands from this list can be sent to the payload.

Subordinates

None

Dependencies

This class depends on services provided by the Server Connectivity package. It provides services to the Graphical User Interface.

Provides interfaces

The InterventionController provides the following functions:

\textbf{isWindowOfOpportunity}(p: PayloadId, ti: Time): Boolean

\textit{Description:} Returns whether \( ti \) is within a window of opportunity booked for payload \( p \).

\textit{Pre:} true

\textit{Return:} serverConnectivity.isWindowOfOpportunity\((p, ti)\)

\textbf{retrieveCommands}(p: PayloadId): \{Command\}

\textit{Description:} Retrieves the commands of \( p \) from the server.

\textit{Pre:} isWindowOfOpportunity\((\text{now}, p)\)

\textit{Return:} commandsOfPayload\([p]\)

\textbf{sendCommand}(p: PayloadId, c: Command)

\textit{Description:} Requests to send \( c \) to payload \( p \)

\textit{Pre:} \( c \in \text{commandsOfPayload}\([p]\) \land \text{isWindowOfOpportunity}(\text{now}, p) \land cs = \text{ObservationInterventionRequiresInterface.getQueuedForUpload}(p) \)

\textit{Post:} \( \text{ObservationInterventionRequiresInterface.getQueuedForUpload}(p) = cs + [c] \)
CHAPTER 5. COMPONENT DESCRIPTIONS

Requires interfaces

This class requires the following functions from the Server connectivity package:

1. getListCommands(p:Payload): [Command]

2. queueForUpload(p:Payload, c:Command)

3. isWindowOfOpportunity(p:Payload, ti:Time): Boolean

The way the required and provided functions are used is displayed in figure 5.1.

References

The InterventionController stores a local copy of the list of commands retrieved from the server. In that way, it works as a proxy [5].
CHAPTER 5. COMPONENT DESCRIPTIONS

Data

\[ \text{commandsOfPayload: Payload} \rightarrow [\text{Command}] \text{. Initially: } \forall p \in \text{Payload} (\text{commandsOfPayload}(p) = \perp) \].

5.3.5 ObservationData

Type

Class

Purpose

Implements software requirement(s) SRFUR46, SRFUR47, SRFUR48, SRFUR49, SRFUR50, SRFUR51, SRFUR62, SRFUR63.

Function

This class contains the necessary functionality to store obtained data. This data is created by the ProcessorBatch by executing its processors on received data from the server. The creation of the data retrieved from the server is handled by the server connectivity package.

Subordinates

None

Dependencies

This class depends on the File system abstraction class.

Provides interfaces

\[ \text{create}(d: \text{Data}, t: \text{Type}) \]

*Description:* Creates an ObservationData object containing \( d \) of type \( t \).

*Pre:* true

*Post:* \( \text{this.getData()} = d \land \text{this.getType()} = t \)

\[ \text{getType}(): \text{Type} \]

*Description:* Returns the type of the data object stored in this

*Pre:* true

*Return:* \( \text{this.type} \)

\[ \text{getId}(): \text{DataId} \]

*Description:* Returns the identifier of the data item.

*Pre:* true

*Return:* \( \text{this.Id} \)

\[ \text{markInteresting}() \]

*Description:* Mark a thumbnail to distinguish it from the less interesting thumbnails.

*Pre:* \( \text{this.interesting} = \text{false} \)

*Post:* \( \text{this.interesting} = \text{true} \)

\[ \text{markUnInteresting}() \]

*Description:* Remove the interesting mark of a thumbnail.

*Pre:* \( \text{this.interesting} = \text{true} \)

*Post:* \( \text{this.interesting} = \text{false} \)
CHAPTER 5. COMPONENT DESCRIPTIONS

markForDownload()
Description: Mark a thumbnail such that its high-volume data will be downloaded.
Pre: true
Post: serverConnectivity.markForDownload(getId())

markForDeletion()
Description: Mark a thumbnail for deletion so that its associated high-volume data on the satellite will be deleted.
Pre: true
Post: serverConnectivity.markForDeletion(getId())

store(location:URL)
Description: Stores the data at location.
Pre: true
Post: FileSystemAbstraction.store(data, location)

Requires interfaces
This class requires a store and a retrieve function from the File system abstraction.

Data
- data: Binary
- type: Type
- id: DataId
- interesting: Boolean

5.3.6 Loader

Type
Class

Purpose
Implements software requirement(s) SRFUR53.

Function
The Loader class lets the system load a plug-in, either a representer or a processor, pointed to by some URL. The Loader creates the physical plug-in objects.

Subordinates
Processor and Representer objects.

Dependencies
The instances of the Representer and the Processor class depend on the Loader to create them. Furthermore, the Loader needs some way to load files from the file system.
CHAPTER 5. COMPONENT DESCRIPTIONS

Provides interfaces

\textbf{loadProcessor(url:String)}

\textit{Description:} Loads the data processor located at url into the system.

\textit{Pre:} \( \exists o \in \text{A precompiled plug-in} (o.location = url \land o \text{ implements ProcessorInterface}) \)

\textit{Post:} \( \exists p \in \text{Processor} (p = o) \)

\textbf{loadRepresenter(url:String)}

\textit{Description:} Loads the representer located at url into the system.

\textit{Pre:} \( \exists o \in \text{A precompiled plug-in} (o.location = url \land o \text{ implements RepresenterInterface}) \)

\textit{Post:} \( \exists p \in \text{Representer} (p = o) \)

\textbf{Requires interfaces}

To fulfill its tasks, the Loader needs the following functions:

- \texttt{RepresenterInterface.create(p:Parameters)},
- \texttt{ProcessorInterface.create(p:Parameters)}.

5.3.7 \textbf{RepresenterInterface}

\textbf{Type}

Interface

\textbf{Purpose}

Implements software requirement(s) SRFUR53, SRFUR57, SRFUR58, SRFUR61.

\textbf{Function}

This interface dictates the functionality a Representer plug-in should provide, if it is to be loaded by a Loader. In order to do this, it has a way to set up a part of the use interface, and a function to update the display when new data comes in. The latter function is inherited from the DataReader interface in figure 4.3.

\textbf{Subordinates}

Depends on the concrete implementation, no subordinates required by interface.

\textbf{Dependencies}

All necessary processors should have processed the data before it is fed to a Representer.

\textbf{Provides interfaces}

\texttt{represent(w: Widget)}

\textit{Description:} Sets up the user interface for displaying observation in \( w \).

\textit{Pre:} \( w \) is a valid widget

\textit{Post:} \( w \) contains all necessary user interface elements to be able to display new observations \( \Rightarrow \texttt{this.isSetUp} \)

\textbf{Requires interfaces}

\textit{None}
Data

- isSetUp: Boolean. Initially, isSetUp is false.

5.3.8 Processor

Type

Class

Purpose

Implements software requirement(s) SRFUR56, SREFR2, SREFR3.

Function

Enable processing of data through the use of plug-ins.

Subordinates

None

Dependencies

The processor class depends on the Loader from the Observation & Intervention package.

Provides interfaces

getParams(): parameters

Description: Returns the parameters of the representer.

Pre: true

Return: this.parameters

create(p: parameters)

Description: Instantiates a representer object.

Pre: true

Post: this.getParams() = p

process(d: data, p: pluginParameters): data

Description: Process data d with parameters p.

Pre: true

Return: Result of applying the plugin on data d with parameters p.

Requires interfaces

None

Data

None

5.3.9 Server connectivity

Type

Class
CHAPTER 5. COMPONENT DESCRIPTIONS

Purpose

Implements software requirement(s) None.

Function

This class provides an abstraction from the communication with the server. It consists of several
functions, calls of which are translated into messages sent to the server. Results are likewise
retrieved from the server.

Subordinates

Dependencies

The Server connectivity class provides interfaces to the Administration and Experiment packages.
It depends on a connection with the server.

Provides interfaces

getAccountParams(name:String):Parameters
Description:  Gets parameters from account with name from the server.
Pre:  true
Return:  connection.getAccountParams(name)

getType(name:String):String
Description:  Gets the type of account with name from the server.
Pre:  true
Return:  connection.getType(name)

hasRightsTo(name:String,p:PayloadId):Boolean
Description:  Returns whether or not account with name has rights to access payload p.
Pre:  true
Return:  connection.hasRightsTo(name,p)

hasRightsTo(name:String,e:ExperimentId):Boolean
Description:  Returns whether or not account with name has rights to access experiment e.
Pre:  true
Return:  connection.hasRightsTo(name,e)

getPayloadParams(p:PayloadId):Parameters
Description:  Gets the parameters of payload p.
Pre:  true
Return:  connection.getPayloadParams(p)

getSatelliteParams(s:SatelliteId):Parameters
Description:  Gets the parameters of satellite s.
Pre:  true
Return:  connection.getParameters(s)

getExperimentParams(p:PayloadId,name:String):Parameters
Description:  Gets the parameters of experiment with name on payload p.
Pre:  true
Return:  connection.getExperimentParams(p,name)
CHAPTER 5. COMPONENT DESCRIPTIONS

retrieveCommands(p:PayloadId):[Command]
Description: Gets the list of commands executable on p.
Pre: true
Return: connection.retrieveCommands(p)

isWindowOfOpportunity(p:PayloadId,t:Time)
Description: Returns whether t belongs to of window of opportunity booked for p.
Pre: true
Return: connection.isWindowOfOpportunity(p, t)

sendCommands(p:PayloadId,c:Command)
Description: Requests the server to upload c to the satellite running p.
Pre: true
Post: connection.sendCommands(p, c)

retrieveLog(f:filter):Log
Description: Returns the log entries satisfying the filter f
Pre: true
Return: connection.retrieveLog(f)

markForDownload(d: DataId)
Description: Mark a thumbnail such that its high-volume data will be downloaded.
Pre: true
Post: connection.markForDownload(d)

markForDeletion(d: DataId)
Description: Mark a thumbnail for deletion so that its associated high-volume data on the satellite will be deleted.
Pre: true
Post: connection.markForDeletion(d)

Requires interfaces
This class requires the interfaces it provides, but then from the server side.

Resources
A secure and reliable connection to the server is needed.

5.3.10 File system abstraction

Type
Class

Purpose
Implements software requirement(s) None.

Function
The file system abstraction provides a simple store and retrieve interface to the client. Doing this, it abstracts from file systems.
Subordinates

None.

Dependencies

None.

Provides interfaces

This class provides a store() function as indicated in figure 4.3.

Requires interfaces

None.
Chapter 6

Scenarios

In this chapter we present scenarios as message sequence charts (MSCs), and we provide a description with them in order to get a better understanding of the internal workings of the system. Scenarios are presented both for the server and the client.

6.1 Server scenarios

6.1.1 Uploading an experiment

In figure 6.1 the server gets a request from a client to upload a certain experiment. The Client connectivity (CC) receives the request and passes it to the controller. The controller will then verify that the user placing the request has got appropriate rights. In case of insufficient rights, a message informing the client of this is sent. In the original client, such a request should not be possible, however when using a custom client a malevolent user might try to send such a command, and therefore the system needs to cope with this.

When access has been granted, the controller retrieves to which payload and satellite the experiment is associated, after which the request is passed on to the queue manager. The queue manager checks whether an internal booking for the satellite (to which the experiment is associated) already exists.

If there is no such booking, the queue manager will make one. First a booking object is created (of which the actual times will be filled in later). Furthermore a list of opportunities is retrieved from the MCC, with the possible bookings between the current time and the time the scheduled start time of the experiment, also a bookingId is included in the request, in order to be able to link the reply to the appropriate booking later on in the process (because of the asynchronous nature of the communication with the MCC). The answer of the MCC is described in MSC 6.2. According to this answer a booking is created, using one of the opportunities provided.

As an alternative, there can already be an internal booking for the satellite (with the status of either pending or accepted). As per design decision only one internal booking for a satellite will exist. It can be the case that the end time of this internal booking is after the scheduled execution time of the experiment. We are not able to make guarantees on the actual execution of the experiment on the satellite, therefore we reject it, and a notification is sent to the client. When the end time of the internal booking is before the scheduled execution time of the experiment we have a valid booking. Note that the client is only notified when something goes wrong. (No news is good news.)

In case of a valid booking an upload experiment command is added to the internal queue.

Once the internal booking has been arranged, the real-time bookings (that were requested by the user), are requested to the MCC. For each of these bookings an external booking will be created according to MSC 6.3.
Figure 6.1: Upload experiment
6.1.2 Requesting a list of opportunities

In MSC 6.2 we describe the answer of the MCC controller (in short MCCC) to a request for a list of opportunities. After the request has been sent to the MCCC we distinguish two possibilities. We could get a list of opportunities from the MCCC, or a time out occurs.

When the MCCC answers with a list of opportunities (which includes a bookingId), the scheduler retrieves the booking with bookingId from the system. Then a check is made whether the list of opportunities is empty or not. If the list is not empty the state of the booking is set to pending, the start and end time of the booking are filled in, and the booking with the first start time in the list is requested. Note that this caters for keeping invariant that every internal booking of the system is the first booking possible for the satellite to which it belongs, that means there is no possible booking that starts at an earlier time. When the list of opportunities is empty, this means that there is no possible booking in the requested window, and the experiment cannot be uploaded in time for execution. For this experiment an error is reported, and the booking object is removed, and the queue entry for this experiment is removed as well. All real-time bookings for this experiment are cancelled as well. Now, if there are any more experiments for the same satellite in the queue, that were created after initiating the upload process for this experiment, the process is repeated in order to get a booking for the other pending experiments.

In case of a time out given by the MCCC, the booking process is repeated unless the scheduled starting time is before the current time.
6.1.3 Requesting a booking

In MSC we describe the answer of the MCCC to a request for a booking. After the request has been sent we distinguish three possibilities. We could receive a booking status of accepted, failed (also time out) or rejected.

When the booking status is accepted, the status of the booking object is updated, and the success is logged. If the status is failed requesting a booking is repeated with the same booking information. Finally, when the status is rejected, getting a list of opportunities is repeated in order to try again using a new opportunity. In case of rejection of an external booking the controller is notified.
CHAPTER 6. SCENARIOS

Figure 6.3: MCC answers with a booking as response to a request booking

6.1.4 Communication in an internal booking

When the time of a scheduled booking comes, an internal booking becomes active. This means that at that moment a satellite passes over a ground station, and communication between the system and the satellite can take place. Whether the queue manager polls for the satellite connection, or the MCS notifies the system will be worked out in the detailed design (DD) phase. The communication that happens then is described in this section, as is given in MSC 6.4.

In this description we assume that the connection with the MCSC has been established. Now as long as the booking is active the scheduler gets pending commands from the upload queue and executes these on the MCSC. Note that the result of executing the command is checked, and if it was not successful the system sends the command again. Then once all commands that were to be uploaded have been executed, in the same way all commands that are in the download queue are handled. Note that during an internal booking a download stream from the satellite is always coming in to the system, and will need to be handled as well.

When the booking is over, that is the communication with the satellite is lost, it is checked whether there are any pending commands in the queue for the satellite. If there are such commands, then a fresh booking is created to handle the rest of the communication, this is similar to the process described in MSC 6.1.
6.1.5 Real time communication

During real time communication with the satellite, communication with the user, as well as interaction of the user with the satellite are extremely important. In this section we describe a scenario for this communication as it is depicted in MSC 6.5. Note that we assume that the connection to the satellite has already been established, as this is similar to the process described in section 6.5.

As soon as the connection has been established the system receives a stream of data, which are in fact, when the satellite is idle, thumbnails. The scheduler forwards this data to the controller, which in turn stores it in the database and notifies the client.

The client is also able to send commands that it want to be executed on the satellite. As only certain commands may be executed on a payload, the controller check with the payload if
the command is valid. If the command is not valid the client is notified of this error. Otherwise
the command is sent to the scheduler to be executed. The scheduler adds the command to the
external upload queue, as the client might send commands in quick succession. Now as long as
there are commands and the booking is active, commands that are in the external upload queue
are executed. This means that a command is sent to the MCSC. Always when performing a
command during real time communication feedback is provided to the client, and no retries for
sending the command are made.

When the booking has reached its end, a notification is sent to the client with all commands
that have not been executed yet.
6.2 Client scenarios

6.2.1 Postprocessing data

The MSC in figure 6.6 describes the actions taken by various components in the client after data has been received from the server. This data is processed by a ProcessorBatch, which executes several processors after each other. In this case, the number of processors is two. When the last processor in the queue is done, an ObservationData is created, and the subscribers to the ObservationData’s type are notified. There can also be several representers, which update their data and show a representation when asked to.

6.2.2 Other scenarios

All other scenarios in the client are already handled in the appendix Prototype description in the SRD [3]. The coupling with the architectural design is rather trivial. All actions concerning the administration of the system can be found in the administration component (see section 5.1.1). For example, adding new users is handled in the class Account which is a subordinate of that component. It should be noted that the creation, modification and removal of experiments is also part of the administration component.

The only things left now are the scheduling of experiments, the management of observation data and the server connectivity. The scheduling of experiments is handled in the schedule experiment component (see section 5.3.2). The management of observation data is handled in the ObservationData component (see 5.3.5). The server connectivity is implicit in the prototype, but is made explicit in the server connectivity component (see section 5.3.9).
Figure 6.6: Processing of data
Chapter 7

Feasibility and resource estimates

7.1 Resources to build the system

In order to be able to build the various components of the IMSETY system, the following resources are needed:

- A C++ compiler,
- the necessary libraries (which for the client include Qt4),
- an x86 compatible computer with about 1.5GHz processor and 512MB primary memory.

Note that for a client any of a Linux, Windows 200/XP/Vista, Mac OS X, or Sun Solaris should suffice, whereas for the server a Linux system is needed.

7.2 Resources to operate the system

Because the system is divided in two parts, the client and the server, it is needed to define separate system requirements. Although the different parts can run on the same system, in practice they will run on separate boxes.

7.2.1 Server requirements

- Operating system: Linux,
- x86 compatible computer with about 1.5GHz processor and 512MB primary memory
- 20GB of free disk space

7.2.2 Client requirements

- Operating system: Windows 2000/XP/Vista, Mac OS X, Sun Solaris or Linux,
- x86 compatible computer with about 1.5GHz processor and 512MB primary memory
Chapter 8

Requirements traceability matrix

8.1 Software requirements to architectural design

<table>
<thead>
<tr>
<th>SR</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRFUR1</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR2</td>
<td>Administration Processor</td>
</tr>
<tr>
<td>SRFUR3</td>
<td>Administration Processor</td>
</tr>
<tr>
<td>SRFUR4</td>
<td>Administration MCC controller MCC connectivity</td>
</tr>
<tr>
<td>SRFUR5</td>
<td>Administration MCS connectivity</td>
</tr>
<tr>
<td>SRFUR6</td>
<td>Administration MCS connectivity</td>
</tr>
<tr>
<td>SRFUR7</td>
<td>Administration Graphical User Interface</td>
</tr>
<tr>
<td>SRFUR8</td>
<td>Administration Controller</td>
</tr>
<tr>
<td>SRFUR9</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR10</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR11</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR12</td>
<td><em>Because of new understanding this requirement will not be implemented.</em></td>
</tr>
<tr>
<td>SRFUR13</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR14</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR15</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR16</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR17</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR18</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR19</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR20</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR21</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR22</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR23</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR24</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR25</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR26</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR27</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR28</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR29</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR30</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFUR31</td>
<td>Schedule experiment</td>
</tr>
<tr>
<td>SRFUR32</td>
<td>Schedule experiment</td>
</tr>
<tr>
<td>SRFUR33</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU34</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU35</td>
<td>Scheduler</td>
</tr>
<tr>
<td>SRFU36</td>
<td>Queue manager</td>
</tr>
<tr>
<td>SRFU37</td>
<td>Internal queue, Internal upload queue, Internal download queue, External upload queue, External download queue</td>
</tr>
<tr>
<td>SRFU38</td>
<td>Internal queue, Internal upload queue, Internal download queue, External upload queue, External download queue</td>
</tr>
<tr>
<td>SRFU39</td>
<td>Bookings, Booking</td>
</tr>
<tr>
<td>SRFU40</td>
<td>Bookings, Booking</td>
</tr>
<tr>
<td>SRFU41</td>
<td>Bookings, Booking</td>
</tr>
<tr>
<td>SRFU42</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU43</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU44</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU45</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU46</td>
<td>Administration, Observation &amp; intervention, ObservationData</td>
</tr>
<tr>
<td>SRFU47</td>
<td>Administration, Observation &amp; intervention, ObservationData</td>
</tr>
<tr>
<td>SRFU48</td>
<td>ObservationData</td>
</tr>
<tr>
<td>SRFU49</td>
<td>ObservationData</td>
</tr>
<tr>
<td>SRFU50</td>
<td>ObservationData</td>
</tr>
<tr>
<td>SRFU51</td>
<td>Administration, Observation &amp; intervention, ObservationData</td>
</tr>
<tr>
<td>SRFU52</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU53</td>
<td>Observation &amp; intervention, Loader, Representer interface</td>
</tr>
<tr>
<td>SRFU54</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU55</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU56</td>
<td>Observation &amp; intervention, Processor</td>
</tr>
<tr>
<td>SRFU57</td>
<td>Observation &amp; intervention, Representer interface</td>
</tr>
<tr>
<td>SRFU58</td>
<td>Observation &amp; intervention, Representer interface</td>
</tr>
<tr>
<td>SRFU59</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU60</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU61</td>
<td>Observation &amp; intervention, Representer interface</td>
</tr>
<tr>
<td>SRFU62</td>
<td>Administration, Observation &amp; intervention, ObservationData</td>
</tr>
<tr>
<td>SRFU63</td>
<td>Administration, Observation &amp; intervention, ObservationData</td>
</tr>
<tr>
<td>SRFU64</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU65</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU66</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFU67</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU68</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU69</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU70</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU71</td>
<td>Administration</td>
</tr>
<tr>
<td>SRFU72</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU73</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU74</td>
<td><em>Because of new understanding this requirement will not be implemented.</em></td>
</tr>
<tr>
<td>SRFU75</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU76</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU77</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU78</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU79</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU80</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU81</td>
<td>MCS controller, MCS connectivity</td>
</tr>
<tr>
<td>SRFU82</td>
<td>MCC controller, MCC connectivity</td>
</tr>
</tbody>
</table>
### CHAPTER 8. REQUIREMENTS TRACEABILITY MATRIX

<table>
<thead>
<tr>
<th>SRFUR83</th>
<th>MCC controller</th>
<th>MCC connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRFUR84</td>
<td>MCC controller</td>
<td>MCC connectivity</td>
</tr>
<tr>
<td>SRFUR85</td>
<td>Administration</td>
<td></td>
</tr>
<tr>
<td>SRFUR86</td>
<td>Controller</td>
<td></td>
</tr>
<tr>
<td>SRFUR87</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR88</td>
<td>Controller</td>
<td></td>
</tr>
<tr>
<td>SRFUR89</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR90</td>
<td>Controller</td>
<td></td>
</tr>
<tr>
<td>SRFUR91</td>
<td>Controller</td>
<td></td>
</tr>
<tr>
<td>SRFUR92</td>
<td>Administration</td>
<td></td>
</tr>
<tr>
<td>SRFUR93</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR94</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR95</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR96</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR97</td>
<td>Controller</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>SRFUR98</td>
<td>Administration</td>
<td></td>
</tr>
<tr>
<td>SRFUR99</td>
<td>Administration</td>
<td></td>
</tr>
<tr>
<td>SRFUR100</td>
<td>Controller</td>
<td>Observation &amp; intervention</td>
</tr>
<tr>
<td>SRFUR101</td>
<td>Graphical User Interface</td>
<td></td>
</tr>
<tr>
<td>SRFUR102</td>
<td>Controller</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>SRFUR103</td>
<td>Controller</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>SREFR1</td>
<td>Administration</td>
<td></td>
</tr>
<tr>
<td>SREFR2</td>
<td>Administration</td>
<td>Processor</td>
</tr>
<tr>
<td>SREFR3</td>
<td>Administration</td>
<td>Processor</td>
</tr>
<tr>
<td>SREFR4</td>
<td>Administration</td>
<td>MCC controller MCC connectivity</td>
</tr>
<tr>
<td>SREFR5</td>
<td>Administration</td>
<td>MCS connectivity</td>
</tr>
<tr>
<td>SREFR6</td>
<td>Administration</td>
<td>MCS connectivity</td>
</tr>
<tr>
<td>SREFR7</td>
<td>Administration</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>SREFR8</td>
<td>Administration</td>
<td>Controller</td>
</tr>
</tbody>
</table>

### 8.2 Architectural design to software requirements

<table>
<thead>
<tr>
<th>Component</th>
<th>SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration</td>
<td>SRFUR1 SRFUR2 SRFUR3 SRFUR4 SRFUR5 SRFUR6 SRFUR7 SRFUR8 SRFUR9 SRFUR10 SRFUR11 SRFUR13 SRFUR14 SRFUR15 SRFUR16 SRFUR17 SRFUR18 SRFUR19 SRFUR20 SRFUR21 SRFUR22 SRFUR23 SRFUR24 SRFUR25 SRFUR26 SRFUR27 SRFUR28 SRFUR29 SRFUR30 SRFUR33 SRFUR34 SRFUR42 SRFUR43 SRFUR44 SRFUR45 SRFUR46 SRFUR47 SRFUR51 SRFUR52 SRFUR62 SRFUR63 SRFUR67 SRFUR68 SRFUR69 SRFUR70 SRFUR71 SRFUR85 SRFUR92 SRFUR98 SRFUR99</td>
</tr>
<tr>
<td>Queue manager</td>
<td>SRFUR36</td>
</tr>
<tr>
<td>Scheduler</td>
<td>SRFUR35</td>
</tr>
<tr>
<td>Internal queue</td>
<td>SRFUR37 SRFUR38</td>
</tr>
<tr>
<td>Internal upload queue</td>
<td>SRFUR37 SRFUR38</td>
</tr>
<tr>
<td>Internal download queue</td>
<td>SRFUR37 SRFUR38</td>
</tr>
<tr>
<td>External download queue</td>
<td>SRFUR37 SRFUR38</td>
</tr>
<tr>
<td>External upload queue</td>
<td>SRFUR37 SRFUR38</td>
</tr>
<tr>
<td>Client connectivity</td>
<td>None</td>
</tr>
<tr>
<td>Requirement</td>
<td>Function/Component</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Controller</td>
<td>SRFUR86 SRFUR88 SRFUR90 SRFUR91 SRFUR97 SRFUR100 SRFUR102 SRFUR103 SREFR8</td>
</tr>
<tr>
<td>Bookings</td>
<td>SRFUR39 SRFUR40 SRFUR41</td>
</tr>
<tr>
<td>Booking</td>
<td>SRFUR39 SRFUR40 SRFUR41</td>
</tr>
<tr>
<td>MCC controller</td>
<td>SRFUR82 SRFUR83 SRFUR84 SREFR4</td>
</tr>
<tr>
<td>MCC connectivity</td>
<td>SRFUR72 SRFUR73 SRFUR75 SRFUR76 SRFUR77 SRFUR78 SRFUR79 SRFUR80 SRFUR81</td>
</tr>
<tr>
<td>MCS controller</td>
<td>SRFUR72 SRFUR73 SRFUR75 SRFUR76 SRFUR77 SRFUR78 SRFUR79 SRFUR80 SRFUR81 SREFR5 SREFR6</td>
</tr>
<tr>
<td>MCS connectivity</td>
<td>SRFUR72 SRFUR73 SRFUR75 SRFUR76 SRFUR77 SRFUR78 SRFUR79 SRFUR80 SRFUR81 SREFR5 SREFR6</td>
</tr>
<tr>
<td>Database abstraction</td>
<td>None</td>
</tr>
<tr>
<td>Graphical User Interface</td>
<td>SRFUR87 SRFUR89 SRFUR93 SRFUR94 SRFUR95 SRFUR96 SRFUR97 SRFUR101 SRFUR102 SRFUR103 SREFR7</td>
</tr>
<tr>
<td>Schedule experiment</td>
<td>SRFUR31 SRFUR32</td>
</tr>
<tr>
<td>Observation &amp; intervention</td>
<td>SRFUR46 SRFUR47 SRFUR51 SRFUR53 SRFUR54 SRFUR55 SRFUR56 SRFUR57 SRFUR58 SRFUR59 SRFUR60 SRFUR61 SRFUR62 SRFUR63 SRFUR64 SRFUR65 SRFUR66 SRFUR100</td>
</tr>
<tr>
<td>Intervention controller</td>
<td>SRFUR100</td>
</tr>
<tr>
<td>ObservationData</td>
<td>SRFUR46 SRFUR47 SRFUR48 SRFUR49 SRFUR50 SRFUR51 SRFUR62 SRFUR63</td>
</tr>
<tr>
<td>Loader</td>
<td>SRFUR53</td>
</tr>
<tr>
<td>Representer interface</td>
<td>SRFUR53 SRFUR57 SRFUR58 SRFUR61</td>
</tr>
<tr>
<td>Processor</td>
<td>SRFUR56 SREFR2 SREFR3</td>
</tr>
<tr>
<td>Server connectivity</td>
<td>None</td>
</tr>
<tr>
<td>File system abstraction</td>
<td>None</td>
</tr>
</tbody>
</table>