EUROPEAN SPACE AGENCY
DIRECTORATE OF OPERATIONS AND INFRASTRUCTURE
OPS-GI

SOFTWARE DESIGN DOCUMENT (SDD)

EGOS Data Dissemination System (EDDS)

Reference: EGOS-GEN-EDDS-SDD-1001
Version: 15.0
Date: 2018-02-16

Document Title:	Software Design Document (SDD)		
Document Reference:	EGOS-GEN-EDDS-SDD-1001		
Document Version:	15.0	**Date:**	2018-02-16
Abstract			

Approval Table:

Action	Name	Function	Signature	Date
Prepared by:	Rokibul Uddin	EDDS Team	Signature numérique de Rokibul Uddin DN : cn=Rokibul Uddin, o=CS GmbH, ou=Space, email=rokibul.uddin@c-ssystems.de, c=DE Date : 2018.04.26 17:16:25 +02'00'	2018-02-16
Verified by:	Delphine Thomas	Application Quality Assurance Engineer (EDDS)	Signature numérique de Delphine THOMAS DN : cn=Delphine THOMAS, o=CS SI, ou, email=delphine.thomas@c-s.fr, c=FR Date : 2018.04.27 00:16:53 +02'00'	2018-02-16
Approved by:	Rui Santos	EDDS TO	Digitally signed by Rui Santos Date: 2018.05.16 13:10:41 +02'00'	2018-02-16

Authors and Contributors:

Name	Contact	Description	Date
M. Hawkshaw	michael.hawkshaw@cgi.com	Author	2012-02-29
M.Lobjakas	merlin.lobjakas@cgi.com	Contributor	2011-09-12
R. Ots	rauno.ots@cgi.com	Contributor	2011-09-12
K. Panitzek	kamill.panitzek@cgi.com	Contributor	2016-06-04
Rokibul Uddin	rokibul.uddin@c-ssystems.de	Contributor	2017-05-05 2018-02-16

Distribution List:

Document Change Log

Issue	Date	Description
1.0	2007-04-13	Update in line with PDR review outcome.
2.0	2010-10-11	Issue for PA Delivery
2.1	2011-03-29	Issue for Final Acceptance Delivery
3.0	2011-09-12	Issue for EDDS v1.1.0i1
3.1	2011-11-14	Issue for EDDS v1.1.0i3
3.2	2012-02-29	Issue for EDDS v1.1.1i1
3.3	2012-04-04	Issue for EDDS v1.1.1i2
3.4	2012-06-27	Issue for EDDS v1.1.2i1
3.5	2012-09-17	Issue for EDDS v1.2.0i1
3.6	2012-09-27	Issue for EDDS v1.2.0i2
3.7	2012-10-12	Issue for EDDS v1.2.0i3
3.8	2013-03-21	Issue for EDDS v1.2.1i1
3.9	2013-04-10	Issue for EDDS v1.2.1i2
4.0	2013-06-21	Issue for EDDS v1.2.2i1

Issue	Date	Description
5.0	2013-11-25	Issue for EDDS v1.3.0i1
6.0	2014-05-28	Issue for EDDS v1.4.0i1
7.0	2014-12-17	Issue for EDDS v1.5.0i1
8.0	2016-02-04	Issue for EDDS v1.6.0i1
9.0	2016-07-04	Issue for EDDS v2.0.0i1
11.0	2016-10-19	Issue for EDDS v2.1.0i1
13.0	2017-05-05	Issue for EDDS v2.2.0i1
15.0	2018-02-16	Issue for EDDS v2.3.0i1

Document Change Record

DCR No:	01		
Date:	2018-02-16		
Document Title:	Software Design Document (SDD)		
Document Reference:	EGOS-GEN-EDDS-SDD-1001		
Page	**Paragraph**	**Reason for Change**	
11, 12, 22, 24	Figure 2, 4.5.5, 6.1.4, 6.1.5	Updated for edds#934	
35, 43	6.1.7.5, 6.2.5	Updated and added for edds#934	

TABLE OF CONTENTS

TABLE OF TABLES

TABLE OF FIGURES

1. Introduction

1.1 Purpose

This is the Software Design Document of the EGOS Data Dissemination System (EDDS) component and provides a top-level architectural design of the EDDS. This document does not include a Detailed Design that is covered by the JavaDoc. See Section 7.2 for information on how to generate the JavaDoc from the source code.

The readership is expected to be software developers and ESA technical authorities with interest in the EDDS component.

1.2 Scope

This document solely addresses the architectural issues of the EGOS Data Dissemination System (EDDS) component together with its related interfaces.

The EDDS provides controlled access to mission data for users who do not have access to the Mission Control System (MCS) monitoring and control facilities. The EDDS does not provide analytic tools, but rather provides mission data in a format that facilitates the analysis process.

1.2.1 Note about the content of the document (Approach)

The EDDS project is developed following an agile approach. Agile methodologies are based on iterative development where requirements and solutions evolve through collaboration between self-organising cross-functional teams. Moreover it encourages frequent inspections and adaptations.

1.3 Document Overview

Section 1 - *Introduction* (this section) provides the purpose, scope and this document's overview.

Section 2 - *References* provides the list of reference documents.

Section 3 - *Glossary* provides a list of acronyms and terms used throughout this document.

Section 4 - *System Design Overview* introduces the system context and design and discusses the background of the project.

Section 5 - *Design Standard, Convention, Procedures* states and summarises the software standards adopted for the architectural and the detailed designs.

Section 6 - *Software Top Level Architecture* describes the high-level architecture of the software without giving on the component level.

Section 7 - *Software Components Detailed Design* describes the design of each software component.

Appendix A - *Top Level Architecture Traceability* traces the top level architecture to the software requirements.

2. References

2.1 Applicable Documents

Ref.	Document Title	Issue and Revision, Date
[AD-1]	Software Requirements Specification (SRS) for the EDDS, EGOS-GEN-EDDS-SRS-1001	Issue 18.0, 2018-02-16
[AD-2]	Software Development Plan (SDP) for the EDDS, EGOS-GEN-EDDS-SDP-0001	Issue 1.1, 2010-01-18
[AD-3]	EDDS External User Interface Control Document (EUICD) [EGOS-GEN-EDDS-ICD-1001]	Issue 15.0, 2018-02-16
[AD-4]	EGOS High Level Software Architectural Design Document [EGOS-GEN-GEN-SAD-1001-i0r0]	0.0 Draft C, 2006-05-15
[AD-5]	Tailoring of ECSS software Engineering Standards for Ground segments in ESA [BSSC 2005(1)]	Issue 1.0, June 2005
[AD-6]	EGOS Preferred 3rd Party Products [EGOS-GEN-GEN-TN-1001]	Issue 1.2, 2006-12-20
[AD-7]	ESOC Generic Ground Systems (EGGS): Development Requirements Specification [EGGS-ESOC-GS-SRS-1001]	Issue 1.0, 2006-11-10
[AD-8]	SFT documentation, EGOS-GEN-SFT-SUM-1001	Issue 2.0, 2010-04-20
[AD-9]	Configuration and Installation Guide (CIG) EGOS-GEN-EDDS-CIG-1001	Issue 14.0, 2018-02-16
[E-DOT]	EGOS Template Guidelines EGOS-GEN-GEN-TN-0010 CIG Template Guidelines EGOS-GEN-GEN-TN-0020 SRN Template Guidelines EGOS-GEN-GEN-TN-0021 ICD Template Guidelines EGOS-GEN-GEN-TN-0030 ADD Template Guidelines EGOS-GEN-GEN-TN-0040 OUM Template Guidelines EGOS-GEN-GEN-TN-0050 SUM Template Guidelines EGOS-GEN-GEN-TN-0060 SRD Template Guidelines EGOS-GEN-GEN-TN-0070	3.0 01.11.07 3.0 01.11.07 3.0 01.11.07 3.0 01.11.07 3.0 01.11.07 2.0 27.10.06 3.0 01.11.07 2.0 27.10.06

2.2 Reference Documents

Ref.	Document Title	Issue and Revision, Date
[RD-1]	Producer-Archive Interface Specification	Draft White Book, December-2005
[RD-2]	Deleted	Deleted

Ref.	Document Title	Issue and Revision, Date
[RD-3]	SCOS-2000 TM Data Retrieval Services SRD [S2K-MCS-SRD-0004-TOS-GIC]	Issue 3.1, 2003-08-29
[RD-4]	Data Disposition System Software Requirements Document [EGOS-MCS-GDDS-SRD-5720]	Issue 1.4, 2004-08-30
[RD-5]	WEB-RM User & Software Requirements Baseline [ESA/TOS-GIC]	Issue 1.4, 2003-11-24
[RD-6]	MUST Repository ICD [No reference] RID523	Draft 1.0 (No Date)
[RD-7]	MUST Rosetta Performance TN [No reference] RID319	Issue 1.1 (No Date)
[RD-8]	MUST Server Software Requirements Specification [No reference]RID319	Issue 1.0, 2006-07-04
[RD-9]	XML Formatted Data Unit (XFDU) Structure and Construction Rules RID255	BLUE BOOK 661.0-B.1, Set 2008
[RD-10]	EGOS High Level Architectural Design Document [EGOS-GEN-GEN-SAD-1001-i0r0]RID494 RID529	0.0DraftC, 2005-05-15
[RD-11]	TDRS External Interfaces Control Document [S2K-MCS-ICD-0017-TOS-GIC]	Issue 4.3, 2004-03-31
[RD-12]	Generic Data Delivery Interface Document (GDDID) [EGOS-MCS-GDDS-ICD-1003]	Issue 1.2, 2004-08-30
[RD-13]	CCSDS Definition Cross Support Reference Model – Part 1, Space Link Extension Services [910.4-B-1]	Issue 1, May-1996
[RD-14]	ESACERT information including Security Plan template and guidelines: http://forum.esacert.esa.int/guidelines.html	N/A
[RD-15]	Implementation of the ESA Network Security Policy for OPSNET [DOPS-COM-POL-34555-OPS-ECT]	Issue 2 Revision 3, 2007-11-09
[RD-16]	EGOS ICD Raw Data Media Production System [EGOS-GEN-EDDS-ICD-1001]	Issue 1, 2007-04-13
[RD-17]	EDDS SDD [EGOS-GEN-EDDS-SDD-1001]	Issue 13, 2017-05-05
[RD-18]	Analysis of Web Services and PARC, Enhancements to support EDDS [EGOS-MDW-TN-1002]	Issue 1, 2007-04-13
[RD-19]	Deleted	Deleted
[RD-20]	MONITORING & CONTROL SCOS-2000 - SFM SERVICE ICD [EGOS-MCS-S2K-ICD-1004]	Issue 6 2008-11-07
[RD-21]	SYSTEM MANAGEMENT SCOS-2000 - SFM SERVICE ICD [EGOS-MCS-S2K-ICD-1005]	Issue 5 2008-11-07
[RD-22]	XML FORMATTED DATA UNIT (XFDU) STRUCTURE AND CONSTRUCTION RULES - Blue book [CCSDS 661.0-B-1]	September 2008
[RD-23]	DARC ICD [EGOS-GEN-DARC-ICD-1001]	Issue 2.0 2014-06-06
[RD-24]	Software Design Document DARC [EGOS-GEN-DARC-SDD-0040]	Issue 2.0 2014-06-06
[RD-25]	EGOS USER DESKTOP ICD [EGOS-MDW-UDK-ICD-0001]	Issue 2.2 2008-11-25

Ref.	Document Title	Issue and Revision, Date
[RD-26]	EGOS USER DESKTOP SDD [EGOS-MDW-UDK-SDD-0001]	Issue 2.1 2008-07-28
[RD-27]	Session Manager ICD [EGOS-MDW-LLC-ICD-1005]	Issue 1.3a 2008-12-12
[RD-28]	Session Manager SDD [EGOS-MDW-COR-SDD-1005]	Issue 1.3a 2008-12-12
[RD-29]	EDDS Client Software User Manual (SUM) [EGOS-GEN-EDDS-SUM-1001]	Issue 12.0 2017-05-05
[RD-30]	Common Development and Distribution License (CDDL) version 1.0 + GNU General Public License (GPL) version 2 https://glassfish.dev.java.net/public/CDDL+GPL.html	
[RD-31]	Apache License, Version 2.0 http://www.apache.org/licenses/LICENSE-2.0	

3. Glossary

3.1 Acronyms

Acronyms	Description
AES	Advanced Encryption Standard
APID	Application Process ID
BSSC	Board for Software Standardisation and Control
CCSDS	Consultative Committee for Space Data Systems
CDR	Critical Design Review
CGI	Common Gateway Interface
CPD	Client Packet Distributor
CQD	Command Query Display
CRUD	Create, Read, Update and Delete
DAO	Data Access Object
EDDS	EGOS Data Dissemination System
EGOS	ESA Ground Operation System
ESA	European Space Agency
GDDS	Generic Data Disposition System
GNU	GNU's Not Unix
GUI	Graphical User Interface
HCI	Human-Computer Interface
HTML	Hyper Text Mark-up Language
HTTP	Hypertext Transfer Protocol
JAR	Java Archive
JDBC	Java Database Connectivity
JNLP	Java Network Launching Protocol
MATIS	Mission Automation System
MCS	Mission Control System
MIB	Mission Information Base
MIME	Multipurpose Internet Mail Extensions
MISC	Miscellaneous subsystem. Translate resource name, path etc
MQD	Monitoring Query Display
MUST	Mission Utility & Support Tools
MTOM	Message Transmission Optimization Mechanisms
OBQM	On-Board Queue Model
OBSM	On-Board Software Maintenance
OBT	On-Board Time
OOL	Out Of Limit
PARC	Packet Archive
PDR	Preliminary Design Review
PHP	PHP: Hypertext Pre-processor
RCP	Rich Client Platform
RDM	Raw Data Medium
RSA	Rivest, Shamir and Adleman. An encryption standard.
SCET	Spacecraft Event Time / Spacecraft Elapsed Time
SCOS	Spacecraft Control and Operations System
SDD	Software Design Document
SIP	CCSDS Submission Information Packages

Acronyms	Description
SMTP	Simple Mail Transfer Protocol
SMTPs	Simple Mail Transfer Protocol Secure
SOAP	Originally: Simple Object Access Protocol – this acronym has been dropped by W3C
SPID	SCOS-2000 Packet ID
SQL	Structured Query Language
SSC	Source Sequence Count
SNMP	Simple Network Management Protocol
SPEL	Synthetic Parameter Expression Language
SUM	Software User Manual
TAR	Tape Archive
TBC	To Be Confirmed.
TCP/IP	Transmission Control Protocol / Internet Protocol
TDRS	Telemetry Data Retrieval System
URL	Uniform Resource Locator
UTC	Co-ordinated Universal Time
WSDL	Web Services Description Language
XFDU	XML Formatted Data Unit
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

3.2 Definition of Terms

Terms	Description
Context	A mission may have a number of potential scenarios such as live operation, testing or training. When a scenario is in use it is referred to as a context (taken from [RD-6]).
Domain	Each domain is an instantiation of SCOS-2000 and is independent from all other domain's SCOS-2000 instantiations (taken from [RD-6]).
EGOS Environment	The runtime environment in which EGOS components operate. This includes the networks, machines, software libraries, configuration, and other dependent EGOS components (e.g. EGOS framework, EGOS core & EGOS common components).
Mission	A mission is a group of related domains (taken from [RD-6]).
Multi-domain	Multi-domain is a single system that is able to manage and control multiple entities referred to as domains (taken from [RD-6]).
Multi-mission	Multi-mission is a single system that is able to manage and control multiple missions (taken from [RD-6]).
OPSLAN	A name given to the set of networks that provides a secure Ethernet-IP Local Area Network at ESOC. OPSLAN allows data to be exchanged between systems used for critical operational activities. The OPSLAN is an ESA Restricted Network and acts as the centre of the IP-OPSNET network, used to communicate with the ESTRACK Ground Stations. The OPSLAN is considered a secure and restricted network being implemented on dedicated independent LAN switches. Switch and trunk redundancy is provided. Communication with external systems is denied by default. If required, the security devices connecting the OPSLAN to other security environments can be configured to permit data exchange with other computer networks (refer to RELAY LANs).
RELAY_LAN	The RELAY_LANs are the Ethernet-IP Local Area Networks at ESOC used to exchange operational data with computers outside ESOC. Direct data exchange from/to computers located outside the OPSLAN to/from the computers connected to the OPSLAN is forbidden. Data must first be relayed to computer systems connected to the RELAY_LAN. It is only after the required security checks are performed, that the data can be relayed to their final destination. The RELAY_LANs are implemented on an independent switch fabric, different from the OPSLAN one. Security systems are used to regulate the traffic to and from the RELAY_LANs and any external or internal networks. RELAY_LANs are within the ESA External Service Networks domain. Any provision of services to external ESA users would normally imply a server (or proxy server) on the RELAY_LAN.
Office LAN	The Office LAN is the Ethernet-IP Local Area Network at ESOC for office automation and corporate applications. Computer systems connected to this network can communicate with similar corporate networks at other ESA sites via the so-called ESACOM. All ESA corporate networks are protected by corporate security systems (Firewalls) regulating data traffic from and to other non-ESA networks. The OFFICE_LAN is within the ESA Internal Service Networks domain. Access is required to EDDS services

Terms	Description
	by users on this LAN.
External Users	Users which access the EDDS services through the RELAY_LAN.
Internal Users	Users which access the EDDS services through the Office LAN or directly from the OPSLAN

4. System Design Overview

This section gives an overview of the EDDS design and its context.

4.1 Introduction

The EDDS provides controlled access to mission data for users who do not have access to the mission control system (MCS) monitoring and control facilities. The data includes telemetry, science and non-science data, telecommand history data, and auxiliary data such as flight dynamics data and data report.

The EDDS provides services that users can access through various client applications. [AD-1] provides background information to the history of the EDDS.

4.2 EDDS Services

The EDDS services can be categorised into three service types:

- Data Services
- Management Services
- Notification Services

4.2.1 Management Services

Management services provide the mechanism to allow user account management and configuration management of the EDDS. User access to the EDDS system and its data is restricted to authorised users within a policy mechanisms based on missions, users, privileges and roles. Many aspects of the EDDS functionality are configurable without software change, the management of which is provided through Management Services.

4.2.2 Data Services

Data Services enable an EDDS user to request and receive mission data. Data services are categorised into:

- Batch Services
- Stream Services

4.2.2.1 Batch Services

Batch services allow clients to make requests for mission data and receive a response that is a finite stored data set. A request contains a list of data types and a set of filters to be applied to each data type. The response data is sent to the client via the delivery method selected in the request. A batch service request can be viewed as a transient request in the sense that the EDDS processes the request, builds the data set (by retrieving the relevant data from mission archives) and then delivers the data set to the user. The request is then considered complete. It should be noted that batch requests can contain schedule information that asks for a request to be run at some future date and could indicate that the request is to be cyclically activated.

4.2.2.2 Stream Services

Stream services allow a client application to receive a continuous stream of mission data rather than a finite stored data set. A client application makes a request for the stream service to the EDDS and then actively requests the streamed data from the EDDS.

4.2.3 Notification Services

Notifications services provide users with the latest changes from the EDDS system. The notifications are asynchronous and are initiated by the EDDS system. This allows clients to receive updates for the

request statuses, user rights changes, quota updates or about new log messages on the fly without polling for new data manually.

4.3 Client Applications

Users interface to the EDDS through a number of client applications:

- EDDS Web Portal (through web browser)
- EDDS MMI Standalone Client Application
- EDDS MMI Web Application (through web browser)
- User client application

A *web browser* is used to interface with the EDDS web site (hosted by an EDDS Web Server). The EDDS web site acts as a portal to the EDDS system and provides access to the download of the EDDS Standalone Client Application and EDDS documentation.

The EDDS Standalone Client Application is downloaded from the EDDS web site and has to be installed on the user's machine. See the EDDS Client SUM [RD-31] for more information on installation configuration and usage. The EDDS Client Application provides a GUI interface to all EDDS services (management and data services).

The EDDS MMI Web Application is an alternative to the standalone application. It offers much the same functionality, but is run in a web browser instead of as a separate application on the client machine. In this document EDDS MMI and EDDS Client Application refer to both of these two applications unless mentioned otherwise.

The EDDS utilises Web Services technology to provide a programmatic interface for application-to-application communication. A user can develop a client application that uses the EDDS web services interface. A full description of the public interface is given in the EDDS External User Interface Control Document (EUICD) [AD-3].

4.4 System Context

The figure below describes the high-level context for the EDDS application.

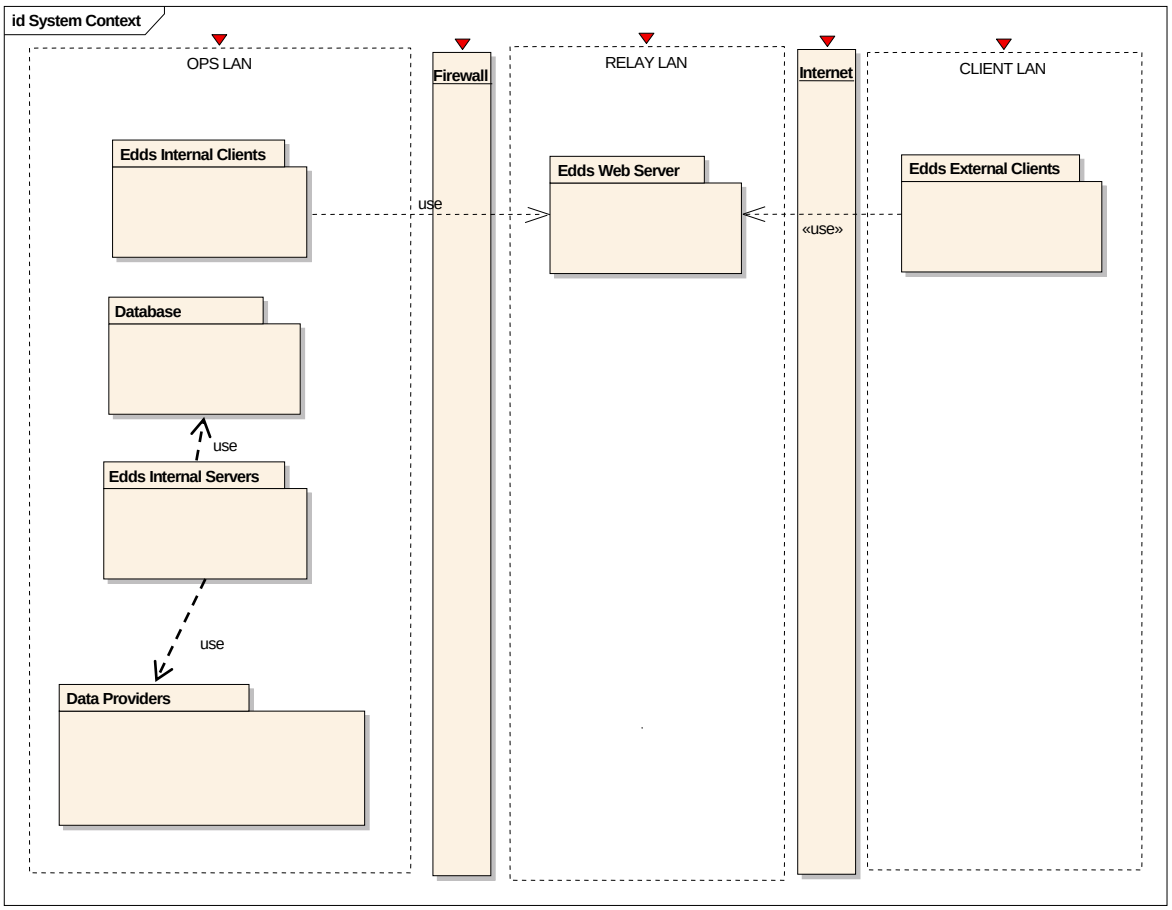

Figure 1 - System Context

The prime function of the EDDS Web Server is to provide access to mission data outside of the OPS LAN environment.

Two categories of clients can be defined:

- Internal Clients: located inside ESOC and accessing the EDDS from the OPS LAN or in the Office LAN;
- External Clients: located inside/outside ESOC and accessing the EDDS through the RELAY LAN.

The EDDS Web Server located in the RELAY LAN provides External Client interfaces to EDDS Servers.

The EDDS Web Server located in the Office LAN provides Internal Client interfaces to EDDS Servers. The EDDS Web Server includes also a Delivery Manager component that allows distributing the requested data to the client via e-mail or via the FTP or SFTP protocol.

The access to the EDDS services can be achieved using:

- EDDS Client Application: Eclipse based standalone or web application;
- Web Portal: to download the EDDS standalone client and EDDS documentation;
- User Client Application: generic application which has access to the EDDS services through a dedicated API (Web Services) (not provided by the EDDS delivery);

The EDDS Server connects with the data providers and executes the requests (batch or stream) received from the EDDS Clients.

Please note that the EDDS Server and the corresponding data providers may be deployed behind a firewall to satisfy the ESA security policy rules.

4.5 External Interfaces

The figure below shows an overview of the system level EDDS interfaces. The interfaces are categorised as:

- Client;
- Web Server;
- EGOS Ground System;
- ActiveMQ Broker;
- Secure File Transfer;
- LDAP;
- Database;
- Request Submitter;
- Stream Client;
- FTP Server;
- Mail User Agent;
- File Server

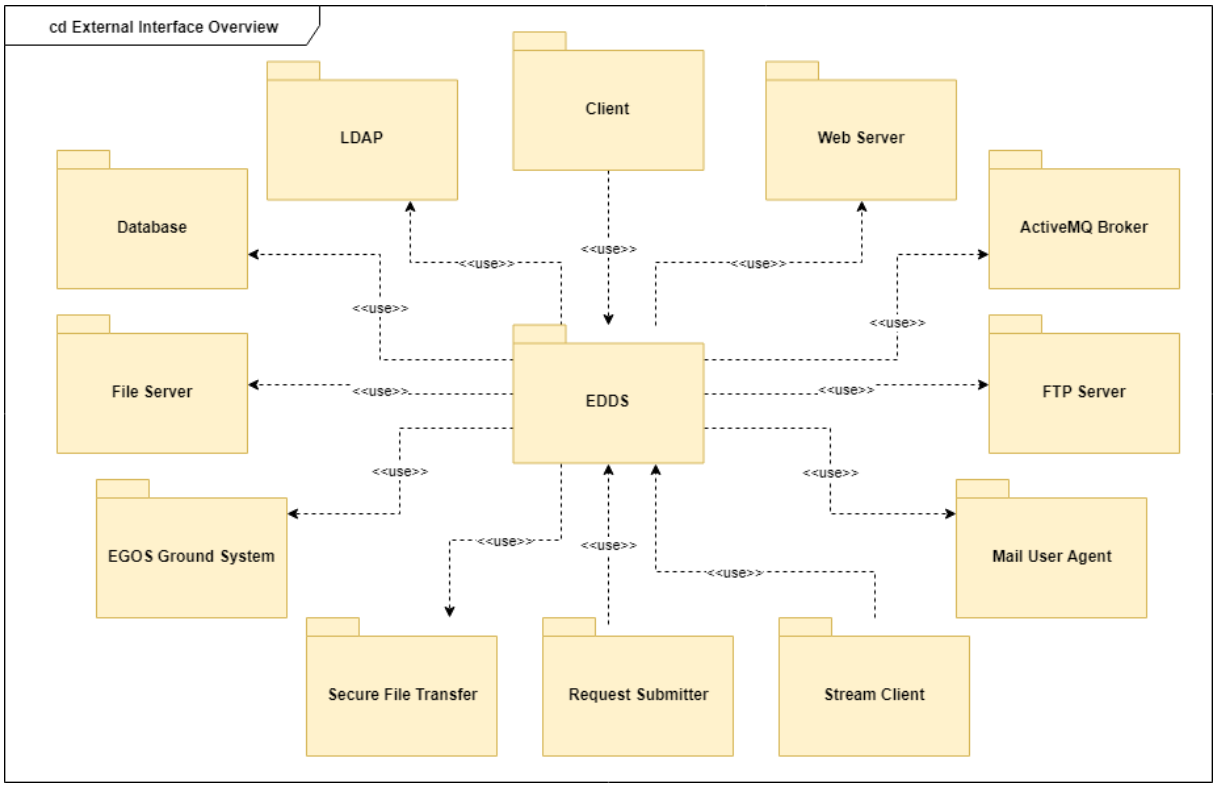

Figure 2 EDDS External Interface Overview

4.5.1 Client

Client applications interface to the EDDS through the EDDS Web Server. Client applications may reside on ESA internal networks or ESA external networks (e.g. Internet). The EDDS External User Interface Control Document (EUICD) describes this interface [AD-3].

4.5.2 EGOS Ground System

The EDDS uses the following EGOS components: in order to access:

- PARC to access Telemetry, Telecommand, Event packets;
- DARC to access Telemetry parameters;
- SMON to access Telemetry parameters;
- FARC to access Mission Files;
- SCOS-2000 to create the MCS reports.

4.5.3 Secure File Transfer

EDDS relies on the SFT library to move data across the ESOC firewall in order to satisfy the ESOC security policy. The mechanism is based on a Secure File Transfer application, which is capable to transfer data files from/to two LANs separated by a firewall. The SFT application uses the TCP/IP protocol The Port used by the application is statically configurable.

4.5.4 Request Submitter

The EDDS Request Submitter is a lightweight standalone application that polls a configurable directory for request files. Any valid request files either in EDDS or GDDS format placed in the directory will be processed and submitted to the EDDS Web Server. Responses are returned as defined in the request.

The processed request file will be moved to an ‘Outbox’ directory. Any failed requests will be moved to a ‘Failed Request’ directory.

4.5.5 Stream Client

The EDDS Stream Client is a lightweight standalone application that can listen to an existing streaming request and save all the received data in files. The client, for now, can only be used for TM type of requests. Received TM packets can be saved in xml, ProtoBuf and GDDS binary format in a specific folder indicated by the user at a certain regular interval.

4.5.6 Web Server

EDDS receives service requests through a Web Server. The adopted protocol for the service requests is SOAP over the HTTP(S).

4.5.7 ActiveMQ Broker

The EDDS uses the ActiveMQ Message Bus to pass messages between EDDS components.

4.5.8 Database

The EDDS requests (including scheduled requests with the Quartz Scheduler) are stored in a MariaDB database. This approach allows the persistence of the requests and simplifies the creation of the EDDS reports.

EDDS will use a JDBC driver to connect with MariaDB, in particular the MariaDB driver. Further information on the configuration of MariaDB is provided in the Configuration and Installation Guide (CIG) [AD-9].

4.5.9 LDAP

The users' information is stored in an LDAP server. This allows the creation and manipulation of the users' information at runtime. In addition, it is possible to configure EDDS to authenticate specific EDDS users against a separate LDAP server to the EDDS LDAP server. This is useful for situations when an LDAP server for user accounts already exists, and users would like to use this username and password for logging in to EDDS. It also makes it easier to manage users centrally – removing the user from the central LDAP server would prevent the user from logging into EDDS as well. The users still need to be entered into the EDDS user management however the check box indicating the user is managed in a separate LDAP database (so called "Central LDAP") should be ticked. In this case, the password does not need to be provided. The user can then be given the required permissions for using EDDS within the EDDS application as normal.

Note: At the time this document has been updated the Session Management functionality provided by the EGOS core components is not sufficient to cover the EDDS requirements and does not provide any interface with the LDAP.

4.5.10 File Server

The delivery manager is capable of delivering batch service responses and Acknowledgement messages to a file server running on the client's site. The delivery manager uses the FTP and secure FTP protocols to deliver the batch response using the PUT method. The client's file server must support the protocol.

The design and architecture of the client's file server is not in the scope of this document.

4.5.11 Mail User Agent

The EDDS can deliver Acknowledgement messages to a user via email. Access to a Mail User Agent must be provided by the EDDS operating environment. The e-mail messages are sent as ASCII content via the standard SMTP protocol. The e-mail user agent is accessed through a standard library API (e.g. Java Mail). This interface is described in the EUICD [AD-3].

EDDS will send e-mails only: it will not read emails.

Note: EDDS relies completely on the service provided by the corporate mail service. It is expected that the E-mail server is accessible from the same LAN where the Web server is deployed.

4.5.12 FTP Server

Users can request their results to be kept on the File Server managed by EDDS. The EDDS accounts are also used for FTP access.

5. Design Standards, Conventions and Procedures

5.1 Architectural Design Method

UML2 is used for the Architectural Design. The diagrams presented in this design document are:

- Component Diagrams: for the high level description of the system components;
- Sequence Diagrams: for the description of the interactions between component parts;
- Deployment Diagrams: for the description of the deployment of the EDDS.

5.1.1 Tools

The following tools have been used:

- Enterprise Architect 6 – used as the architectural modelling tool;
- Eclipse 3.5 - used to generate the EDDS MMI.
- Altova XMLSpy 2011 for generating documentation from WSDL schemas

5.2 Detail Design Method

The detailed design of the EDDS is composed by the JavaDocs.

5.3 Documentation Standards

The documentation for the EDDS has been created using the document templates described in [AD-6] and [E-DOT].

Development should follow ESOC Generic Ground systems (EGGS) Development Requirements Specification standards [AD-9], unless specifically indicated in the SDD to the contrary.

5.4 Naming Conventions

General naming conventions have been applied following [AD-9], [AD-6], [AD-16] and [AD-17].

5.5 Programming Standards

The programming standards to be followed during implementation are to follow the specification of the standards in [AD-9].

5.6 Reuse of Pre-developed Software

The following section describes the use of pre-developed software within the EDDS.

5.6.1 COTS

The following COTS products are listed for use in the implementation of the EDDS. Refer also to the Configuration and Installation Guide (CIG) [AD-9]:

- Ant 1.9.4 as build tool;
- Apache Maven 3.2.5 as build tool;
- Apache Tomcat 8.0.23 as web server;
- Eclipse 4.4 as framework for RCP support;

- RAP 2.2.0 as EDDS client web application framework
- MyBatis 3.2.8 for the interface with the EDDS database. MyBatis uses JDBC as a low level communication protocol;
- MyBatis Generator 1.3.2 for generating the required classes and mapper files from the EDDS database for connecting to the database with Java;
- MyBatis Spring 1.2.2 for connecting to the EDDS database via Spring;
- Java 1.8.0_72 64-bit (for compiling and running EDDS);
- JUnit 4.11 for unit testing;
- MariaDB Database 10.0.22 for the EDDS internal database;
- OpenLDAP 2.4.12-7.14;
- Quartz 2.2.1 for parsing Cron Expressions for scheduling;
- Log4j 1.2.17 for logging;
- SLES 12 as the operating system;
- XFDU library 1.1 for creating XML Formatted Data Units;
- Spring Framework 4.1.3 for dependency injection;
- Spring LDAP 2.0.2 for connecting to LDAP via Spring;
- ActiveMQ 5.10.0 as JMS provider;
- CometD 3.0.3 Java libraries for web server and client asynchronous communication;
- Camel 2.14.1 as routing engine for transferring data between endpoints;
- Apache Commons Codec 1.8 for encoding and decoding data in Base64;
- Apache Commons Lang 3.3.2 Java utility for creating hash codes;
- Mockito 1.10.8 for mocking libraries in unit testing;
- Cobertura 2.1.1 for checking unit test code coverage.

The Apache Tomcat and Metro COTS products have been identified as the COTS which provide support for WSDL and SOAP based web services. They are available under open software licences, respectively: Apache licensing [RD-33] and Common Development and Distribution License (CDDL) version 1.0 + GNU General Public License (GPL) version 2 [RD-32]. Quartz and Log4j also satisfy the Apache licensing [RD-33].

5.6.2 ESOC Bespoke Software

The following ESOC packages are used for the EDDS Implementation:

- An EUD4S2K Jar file containing the compiled SCOS IDL files;
- Secure File Transfer for the transport of the data through the ESOC firewall;
- Egos User Desktop 3.1 for the Client Implementation;
- FARC Java client API jars;
- DARC Java client jar.

6. Software Top Level Architectural Design

6.1 Overall Architecture

This section provides a summary of the software architecture and a top-level overview of the components and interfaces that are used within EDDS.

6.1.1 Top Level Component Architecture

The figure below illustrates the EDDS top-level architecture.

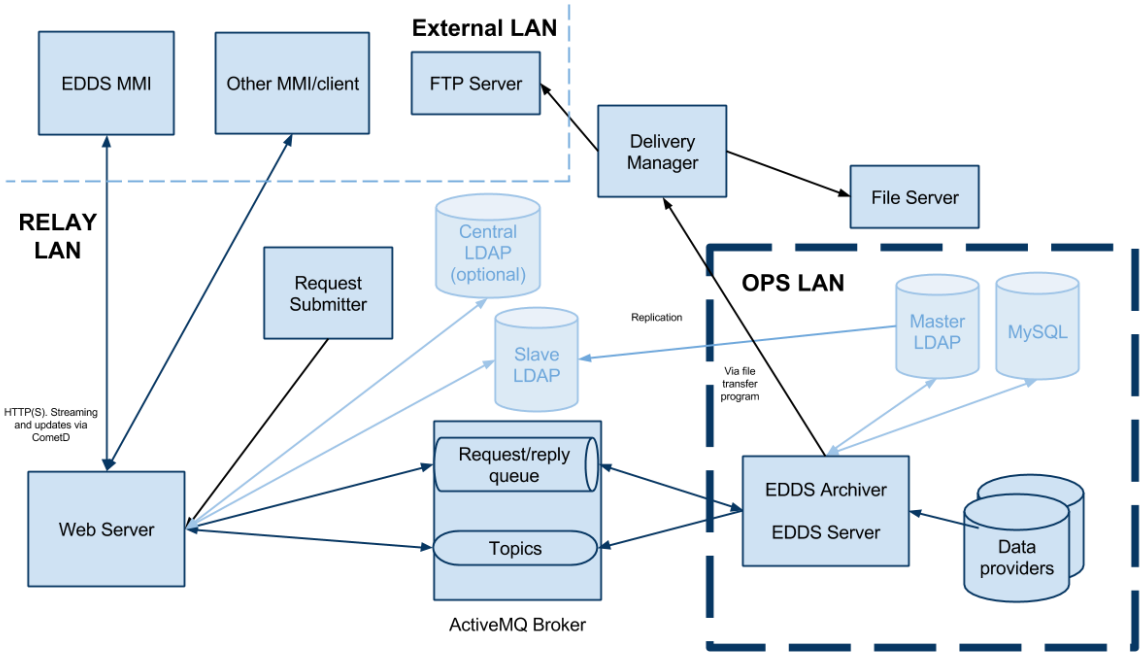

Figure 3 - High Level Architecture

The EDDS architecture foresees a Client MMI application (which is downloaded from a web browser or used directly with the browser). Using the GUI a user can build requests (batch, stream and user management) which are then submitted as SOAP service requests to the EDDS Web Server. Multiple EDDS MMI clients can issue requests concurrently. EDDS provides a Java library which can be used from any other client to access the EDDS services.

The EDDS Web Server takes care of user authentication. The system authenticates users as soon as possible in the process chain, especially prior to passing the request across the OPSnet firewall to the OPS LAN.

SOAP requests (batch, stream and user management) received by the Web Server are defined as XML data structures that are then passed on to a message bus and later stored in a relational database by the EDDS Archiver. The approach allows the persistency of the requests and simplifies the management/creation of the EDDS reports. The EDDS DB will be co-located with the EDDS Server and Archiver. The EDDS Archiver contains a scheduler for scheduling recurring batch requests. The acknowledgement messages associated to each request will also be stored in the relational database.

The EDDS Server interfaces with the different data archives in order to execute the EDDS service requests (batch and streaming). The EDDS server processes the service requests to the corresponding data provider. The data provider could be a CORBA interface, or a proprietary Java API.

Currently, the EDDS interfaces with the FARC, DARC, SMON, Data Provision Services and PARC in order to retrieve files (FARC), parameters (DARC, SMON) and packets (PARC, Data Provision Services). The EDDS Web Server acts as a data provider for the EDDS reports.

The EDDS Server receives requests that are delivered on the message bus. The EDDS Web Server is responsible for the validation of a request and to verify if a user possesses enough privileges (Authorisation) to execute the request.

The result of the service request (batch and stream) is stored into files and packaged depending on the request definition. The packages are thereafter stored in the EDDS local file system. The files can either be picked up by the client or delivered by the daemon (Delivery manager) either via FTP or SFTP to a location specified by the client or sent via e-mail (acknowledgement).

The EDDS clients interact with the web server through the EDDS client API. The purpose of the APIs is to provide a façade to the client that hides the details associated with the access to the web services.

ActiveMQ is used as the Java Messaging System (JMS) compatible Enterprise Message Bus. There are two important concepts to understand in any JMS system – queues and topics. A message can be placed on a queue or a topic. A message placed on a queue is guaranteed to be delivered to only one of the listeners. This allows for scaling whereby multiple applications on different systems can be running to load balance a particularly busy queue where response times are important. A message placed on a topic is sent to all the listeners. This allows for, say, an update of the status of a request to be received by all the client applications listening on that topic. If a queue or topic is persistent, the message will be retained by the broker until the consumer starts listening to the queue or topic again.

6.1.2 EDDS deployment

The EDDS is a distributed application, and for the different components the following deployment is foreseen:

- One instance of MariaDB database for each type of user (External or Internal). The EDDS DB has to be located in the same LAN where the EDDS Server and Archiver are deployed. The DB is used by the EDDS server to store the Service Requests;
- At least one instance of the EDDS server. The EDDS server includes the Data Provider drivers. The location of the EDDS server depends on the location of the Data Providers. The possible deployment scenarios are described in the section below in more detail. The EDDS application is able to support multiple instances of Data Providers. Please refer to Section 6.1.4 for more details;
- One master/slave instance of OpenLDAP is used to store the user information. The Slave LDAP server must be located in the RELAY LAN. The Master LDAP server must be accessible by the EDDS Server.

The possible scenarios are described below:

1. Data Providers deployed in the OPS LAN;
2. Data Providers replicated in the RELAY LAN;
3. Data Providers split between OPS LAN and RELAY LAN;
4. Data Providers replicated in the Office LAN.

6.1.2.1 Data Providers deployed in the OPS LAN

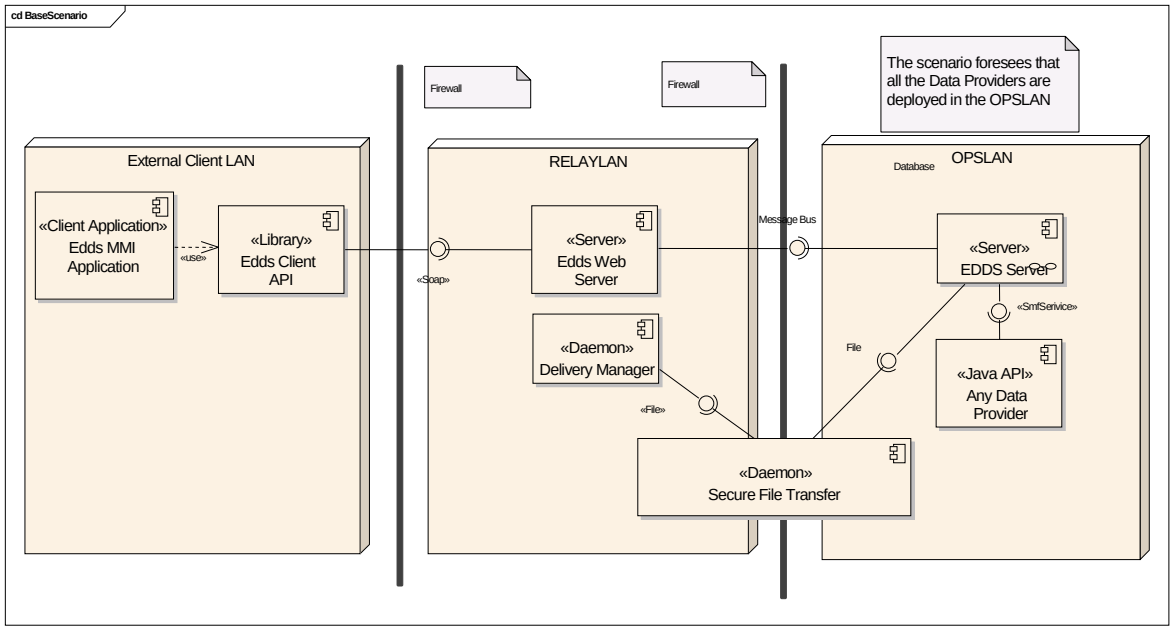

Figure 4 - Data Providers deployed in the OPS LAN

The figure above shows the base deployment scenario. The EDDS server and the Data Providers are all deployed in the OPS LAN. The figure considers the case of External Users accessing the EDDS services. In the RELAY LAN the (a) EDDS Web server (b) the File Server are deployed, and in the OPS LAN the (c) the EDDS DB (MariaDB) and (c) the LDAP Server are deployed. Note that the EDDS DB and LDAP must always be deployed on the same LAN as the EDDS Server. This scenario can also be applied for the internal users: in which case, the EDDS Server and the EDDS DB would be deployed in the internal LAN based on technical and service suitability. The generated Response Files are moved across the firewall through the Secure File Transfer.

The advantages of such a deployment are:

- It is the simplest scenario to be configured;
- Simple interaction with the components in the OPS LAN.

The disadvantages of such a deployment are:

- High level of traffic through the OPS LAN firewall from the file transfers;
- Possible slowness of large file transfers through the firewall.

6.1.2.2 Data Providers replicated in the RELAY LAN

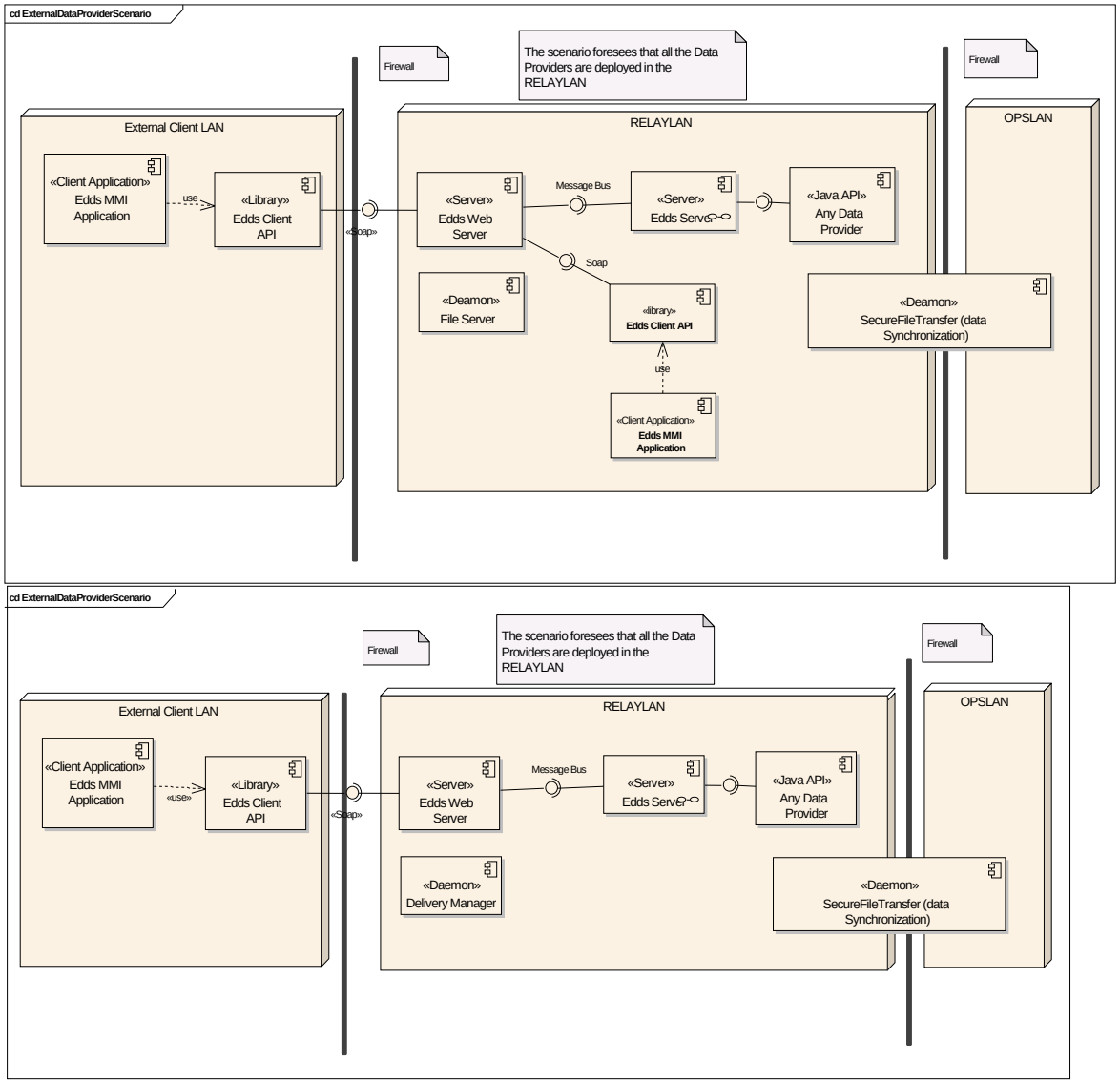

Figure 5 - Data Providers replicated in the RELAY LAN

The figure above shows the scenario where the Data providers are deployed outside the OPS LAN. The figure considers the case of External and Internal Users accessing the EDDS services. In the RELAY LAN the following are deployed: (a) EDDS Web server (b) the Delivery Manager, (c) the EDDS DB, (d) the EDDS Server (e) the EDDS Archiver (f) LDAP (g) the Data Providers (h) EDDS Client Application (i) EDDS Client API.
The Secure File transfer is used to synchronise the data providers contained in the OPS LAN with the data contained in the RELAY LAN. This is outside of the scope of EDDS.

The advantages of such a deployment are:

- Reduced level of traffic through the OPS LAN firewall;
- The data are moved only for the Data Provider synchronisation.

The disadvantages of such a deployment are:

- Some EDDS services would not be available as the necessary data providers cannot be moved outside the OPS LAN;
- Data replication;

- Data has to be replicated outside the OPS LAN for each type of user request;
- Decrease of security since the data is replicated in a less secure LAN;
- Possible slowness of large file transfers through the firewall.

6.1.2.3 Data Providers split between OPS LAN and RELAY LAN

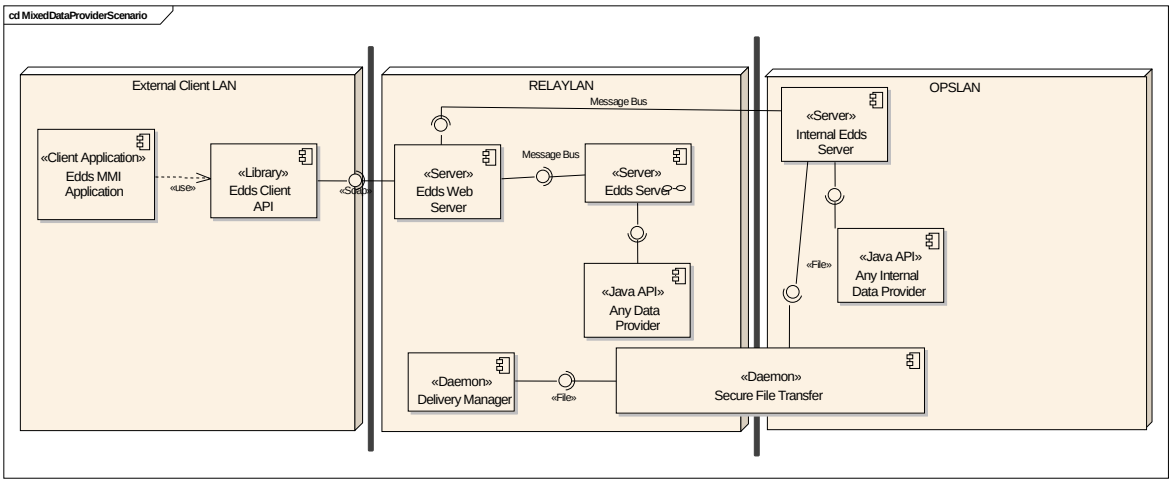

Figure 6 - Data Providers split between OPS LAN and RELAY LAN

The figure above shows the case where the data providers are partially deployed in the OPS LAN and partially deployed outside the OPS LAN. This scenario occurs when EDDS services (e.g. Reports) cannot be copied outside the OPS LAN. In this case, the scenario foresees two separate instances of the EDDS server one inside the OPS LAN and one outside the OPS LAN. The two EDDS instances process only a subset of the available services according to the data provider location. See Section 4.3 of the Configuration and Installation Guide (CIG) [AD-9]

The advantages of such a deployment are:

- Better performance on request execution if the data providers are deployed outside the OPS LAN;
- Capability to support all the EDDS services;
- Reduced level of traffic through the OPS LAN firewall.

The disadvantages of such a deployment are:

- Data replication; Data have to be replicated outside the OPS LAN for each type of user.
- Complex configuration;
- Possible slowness of large file transfers through the firewall.

6.1.2.4 Data Providers replicated in the Office LAN

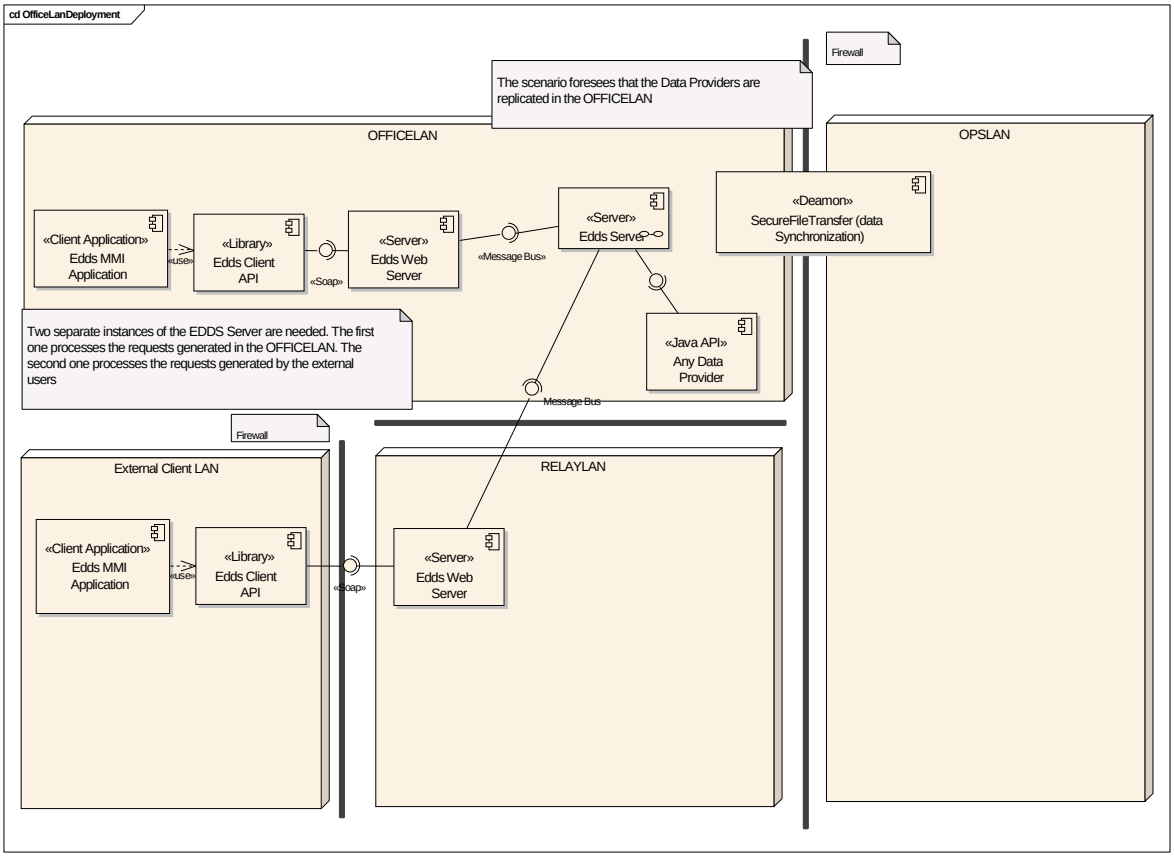

Figure 7 – Data Providers replicated in the Office LAN

The final scenario described next, foresees the Data Providers replicated only on the Office LAN. Each type of user accesses its own instance of Web Server and EDDS Server.

The advantages of the deployment are:

- Better performance on request execution (particularly for the internal users);
- Data replicated only once;
- Reduced level of traffic through the OPS LAN firewall.

The disadvantages of the deployment are:

- Complex configuration;
- Possible security issues since the data are replicated in a less secure (although still internal) LAN.
- Possible slowness of large file transfers through the firewall.

6.1.2.5 E-mail server deployment

The EDDS sends acknowledgement messages to the user via e-mail. As such access to the e-mail server is needed, it is assumed that the e-mail server is always deployed or accessible from the LAN where the EDDS Server is deployed, i.e. OPS LAN.

EDDS is an email client only and is not able to receive emails.

6.1.3 EDDS vs. ESOC Network Security requirements

The EDDS satisfies the ESOC Network Security requirements, which define mainly that no ingoing connections are allowed and no permanent connections are established between the OPS LAN and an external LAN, and any connection is initiated from the OPS LAN.

If the data providers are deployed outside the OPS LAN, the only interaction with the OPS LAN is synchronisation of the data providers. This is achieved using the Secure File Transfer (SFT) daemon, which has been developed to satisfy all the ESOC Network Security requirements. The SFT is an independent product and supports the transfer of data files in both directions through the firewall using TCP/IP.

If the data providers are deployed inside the OPS LAN the EDDS server accesses the ActiveMQ message bus broker located in the same LAN where the EDDS Web Server is deployed The connection with the broker is open only when needed and it is open from inside the OPS LAN. All the security requirements are hence satisfied. The communication with the broker is via the TCP/IP protocol. In particular, EDDS will use the Java JMS API to contact the broker through the default port of 61616.The data responses are transferred back to the RELAY LAN using the SFT daemon.

The LDAP master and slave synchronisation is initiated from and LDAP application (Slurpd) which is located inside the OPS LAN. When the Slurpd daemon notices that there are changes to propagate to the slave “slapd” instance(s), it forks one copy of itself for each SLAP slave to be updated (one in the typical EDDS deployment). Each child process thereafter binds to the slave LDAP server, and sends the changes. The Slurpd has an option which allows running it in a “one shot” mode, which will guarantee that the connection across the firewall for the synchronisation will be not permanent.

All the communications across the firewall are based on the TCP/IP protocol configured to use static TCP ports.

6.1.4 EDDS Scalability

The EDDS application provides a high level of scalability.

EDDS supports a multi mission and multi domain environment. In order to achieve this it is foreseen the following deployment of the EDDS components:

- Web Server: One instance shared from the different missions. In case a mission requires a complete separated environment, multiple web servers accessible via separated web addresses can be deployed and the user has to configure the MMI in order to connect to a specific Web Server.
- EDDS Archiver: Only one EDDS Archiver is deployed for a multi or single mission deployment (see following text for more details);
- EDDS Server: For each mission, a separate instance of the EDDS Server is deployed (see following text for more details);
- EDDS Data Providers: For each mission and domain a separate instance of the Data Provider driver is deployed;
- Delivery Manager: Only one delivery manager should be deployed. The EDDS Servers should be configured to place completed files in the same folder that the Delivery Manager has been configured to poll.
- Stream Client: Many Stream Client should be deployed for each mission (optional).
- Secure File Transfer: For each mission and domain, two or more instances of Secure File Transfer are deployed. One instance is needed on each side of the firewall.

Note: the applications can be deployed on a single physical process or instantiated on multiple processes running on the same or separate machines.

Note: In the context of EDDS, the concept of domain and mission can be extended to support multiple instances of the same Data Provider. Multiple instances of Data Providers are seen from EDDS as either separate missions or separated domains. Hence, multiple instances of the same Data Provider are supported from separate instances of EDDS servers and EDDS Data Providers.

Only one copy of the EDDS Archiver should be run, otherwise duplicate log messages will be saved in the database. This is because log messages are sent to a “topic” on the message bus. Messages placed on a topic are sent to all applications that are listening to the topic. This would mean all running EDDS Archivers would receive the same log messages and archive them in the database. Messages placed on a queue on the message bus on the other hand are guaranteed to be sent to only one of the listeners.

The EDDS Archiver also responds to requests for status information, historical logs, deletion requests and the number of ongoing requests and saving updates to request acknowledgements. These requests are placed on a queue. It is therefore possible to run more than one EDDS Archiver to balance the load of these requests, but first the Camel route to archive the log messages must be removed from the “archivercontext.xml” file for all but one of the Archivers.

Multiple EDDS Servers for the same mission and request sub types could also be run, as requests are placed on a queue, so only one of the EDDS Servers will receive the message and process the request. However, there is a check in the EDDS Server for any ACTIVE requests on start-up. If any are found in the database, it is assumed that it did not complete the request successfully and will process the request again. If another EDDS Server for the same mission is currently processing the same request, the two servers will compete with each other to process it. If it is necessary to run multiple EDDS Servers for the same mission, it would be necessary to ensure there are no ACTIVE requests when it is started up.

Some services, for example updating the Master LDAP for user management and quota updates and sending e-mails, are performed by the EDDS Server but are not mission specific. As these requests are placed on a queue, the first EDDS Server to receive the message will process the request, regardless of which mission it is managing.

If FARC Subscription requests are being taken from a topic, then the property “farc.jms.enabled” within edds.properties for all but one of the running EDDS Servers should be set to false.

6.1.5 EDDS Component Overview

This section provides a breakdown and general description of the components and sub-components that comprise the EDDS. A detailed description of each is given in section 6.2.

Top Level Components	***Sub-Components***	***Summary Description***
EDDS MMI Application	Standalone runtime Web Application	Depending on the privileges assigned to a user’s role, it allows: • requesting data: batch and streaming; • displaying delivered data: batch response files and streaming; • monitoring the status of requests; • administering user accounts; • administering mission data configuration. A web browser is used to download the EDDS MMI standalone client and the relevant documentation.
Generic EDDS Client		Any other client which requires access to the EDDS services (e.g. MUST, ARES)
EDDS Client APIs		Java APIs, which allow clients submitting batch and stream requests to the EDDS server and register for asynchronous notifications. The APIs allow managing the retrieval of responses from the EDDS server. The APIs act as a façade over the adopted communication protocol (SOAP)
EDDS Web Server		The JAX-WS based web server is in charge of

		authenticating any EDDS user and processes the EDDS services requests (SOAP requests) generated through the EDDS Client API. Also provides notifications services.
Secure File Transfer		Forwards the files for the Data Provider synchronisation when the Data Providers are deployed in the RELAY LAN and OFFICE LAN. Forwards the EDDS service responses and the data response files from the OPS LAN to the RELAY LAN when the data provider is located in the OPS LAN.
Request Submitter		Responsible for the validation and transformation of EDDS and GDDS requests into a format that can be issued to the EDDS Server for processing.
Stream Client		Standalone client for save stream data to file.
EDDS Archiver		Processes non-mission specific requests, for example new batch requests from the web server that need to be scheduled, added to the database and sent to an EDDS Server for processing. It also processes acknowledgement updates, responds to requests for status information in the database, the number of active requests and archives log messages sent by all other applications.
EDDS Server		Core EDDS server responsible for: • receiving data requests that need to be processed; • authorising the requests; • routing data requests to the corresponding data provider; • executing MCS report data requests; • retrieving streamed data from the data provider, filtering it, and providing the stream to the client.
	Request Manager	Receives the data request sent on the message bus and authorises the request according to the user privileges.
	Request Execution Manager	Distributes the service request to the appropriate data provider driver.
	EDDS Database	Stores the EDDS requests and their current status
	LDAP Interface	Stores the EDDS user information.

PARC data provider	Request Processor	Processes the specific service requests (batch and stream) and retrieves the required data from the PARC. In detail the following operations are performed: • Data Retrieval: the data request is performed through the usage of the PARC Manager. Wherever possible the filter options are added in the service request; • Filter: whenever the archive interface does not allow a filtering capability, it processes the retrieved data in order to filter as requested; • Formatter: Formats the retrieved data as requested; • Compression: compresses the formatted data as requested; • Encryption: encrypts the compressed data as requested.
FARC data provider	Request Processor	Processes the specific service requests (batch) and retrieves the required data from the FARC. In detail the following operations are performed: • Data Retrieval: data request is performed through the use of the FARC Java API. Wherever possible the filter options are added in the service request; • Filter: whenever the archive interface does not allow filtering capability, it processes the retrieved data in order to filter as requested; • Formatter: Formats the retrieved data as requested; • Compression: compresses the formatted data as requested; • Encryption: encrypts the compressed data as requested.

File System Data Provider	Request Processor	Processes the specific service requests (batch) and retrieves the required data from the EDDS File System Index. In detail the following operations are performed: • Data Retrieval: data request is performed through the use of the EDDS File System Index; • Formatter: Formats the retrieved data as requested; • Compression: compresses the formatted data as requested; • Encryption: encrypts the compressed data as requested.
DARC data provider	Request Processor	Processes the specific service requests (batch and stream) and retrieves the required data from the DARC: In detail the following operations are performed: • Data Retrieval: the data request is performed through the usage of the DARC Java API. Wherever is possible the filter options are added in the service request; • Filter: whenever the archive interface does not allow filtering capability, it processes the retrieved data in order to filter as requested; • Formatter: Formats the retrieved data as requested; • Compression: compresses the formatted data as requested; • Encryption: encrypts the compressed data as requested.

SMON data provider	Request Processor	Processes the specific service requests and retrieves the required data from the SMON service.
EDDS Report data provider	Request Processor	Processes the specific service requests and retrieves the required report data from the database. In detail the following operations are performed: • Data Retrieval: EDDS report data is stored on the DB; • Filtering: Filters are applied during the elaboration of the report, if part of the request itself; the MMI provides filtering capabilities as well, but in this case the filter is applied on the fly in the EUD view. • Formatting: EDDS report are always delivered in XML format; • Compression: compresses the formatted data as requested; • Encryption: encrypts the compressed data as requested. Delivery: EDDS Reports are available for FTP and server delivery.
Delivery Manager		The delivery manager polls a configurable directory for files created by the EDDS Server in response to requests that it has processed. It then delivers the file to the required destination (either made available to the web server by saving to the local FTP directories or upload to a remote FTP server). The default filename of the response includes the request ID that is used to retrieve the details about the file from the database.
		Notifies users via email and sends file responses through the FTP-based File Delivery mechanism and handles the stored data. (e.g. Automatically deletes data after a configurable time)

Table 1 EDDS Component Summary

6.1.6 EDDS Services

This section describes the EDDS services from the point of view of the EDDS architecture. The figure below illustrates the relations between the EDDS services files and the EDDS software components. The details of the services are defined in the EUICD [AD-3].

The EDDS services are the key element of the EDDS architecture.

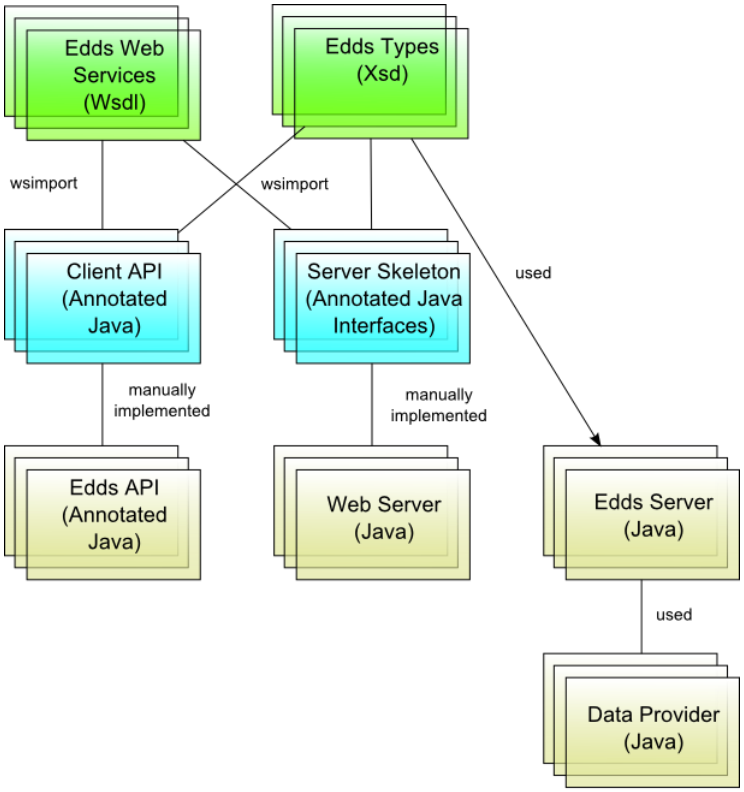

Figure 8 EDDS Services Architecture

The EDDS services are described as web services, i.e. through a set of WSDL files which list the possible operations (e.g. perform a batch request) and a set of EDDS data types described as XSD schema files. Such types are self-contained structured data used from the web services as arguments.

In detail, the following services are foreseen:

- Authentication services: which allow the user to log-in/log-out from the EDDS;
- Batch request services: which allow the user to submit batch requests;
- Stream request services: which allow the user to submit stream requests;
- User Management services: which allow the authorised user to create/modify/delete the user which access the EDDS services ;

The EDDS Web Services file and the Type files satisfy the JAX-WS 2.0 programming model, which supports the SOAP protocol and allows generating annotation-based models to develop Web Service server applications and clients. Using the tools provided by the JAX-WS library (wsimport) the Web Services and the XSD data types are translated into java code.

The result of the translation is:.

- Client Stub classes;
- Server Skeleton files;
- A set of bean classes which describe the types defines in the XSD;

The generated files constitute the baseline from which the following are built:

- The EDDS Clients APIs;
- The EDDS Web Server;
- The EDDS Server;
- The EDDS Data Providers services (PARC, DARC, FARC, Report).

6.1.7 Scenarios

This section presents a set of scenarios that give a representative subset of typical interactions between EDDS components. The scenarios are client-driven, and provide a summary of the functional aspects of the software within the scope of the given scenario.

6.1.7.1 Logging in to EDDS (user outside OPS LAN)

This scenario shows the interactions between components when a user logs in to EDDS.

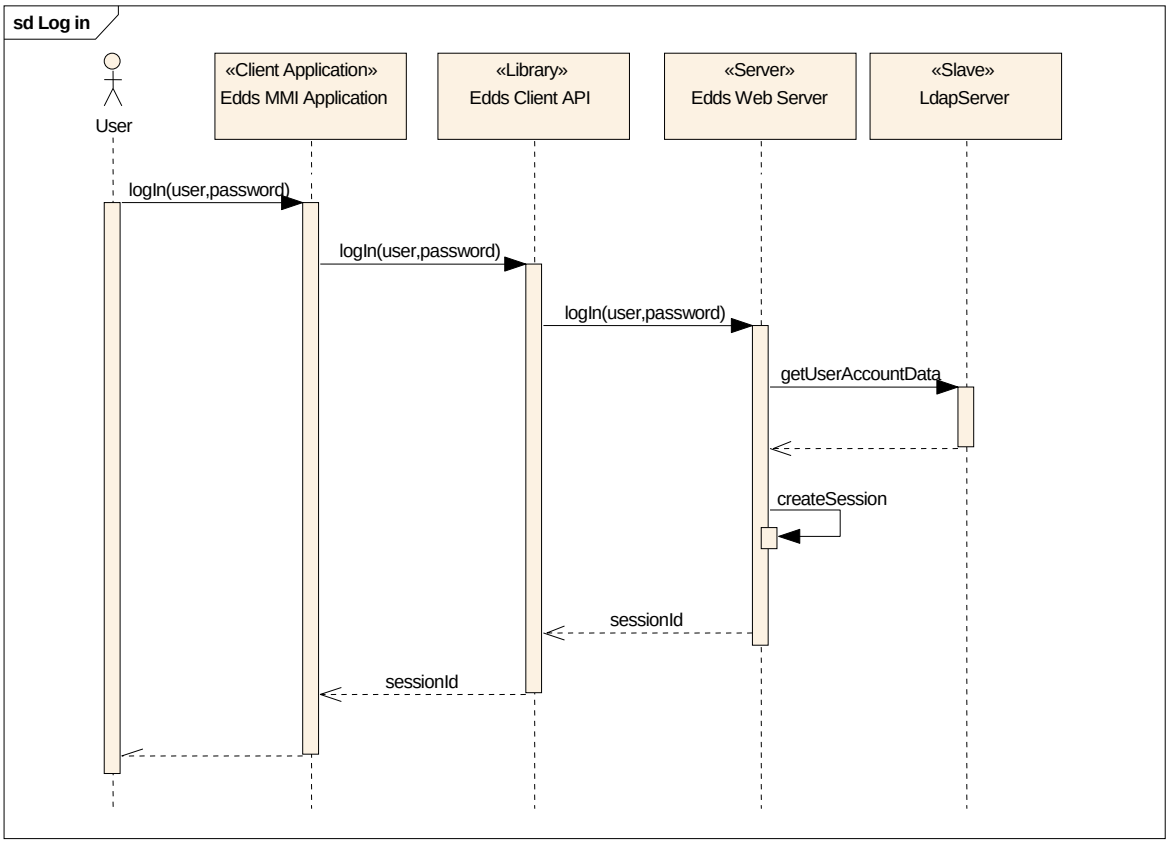

Figure 9 : Logging in to EDDS

The figure above shows the interactions from the client side as a user logs in to the EDDS system. The figure indicates that there are three stages for the login. During the first stage, the user enters his or her name and their password. The operation opens a session on the web server. The web server accesses the Slave LDAP server directly to get the user data and verify the entered data. If the user and password correspond to the one stored in the server the session is opened and the session ID is returned to the user. Only if the web session has been opened successfully can a batch request be sent to the EDDS Server. This operation is transparent to the EDDS client and helps to minimise requests over the LAN in case the data provided by the user are not correct (e.g. no session, session expired).

After the session has been opened, the operator can retrieve the list of their mission and the list of their roles (for each mission). For the EDDS Client MMI application, this is handled automatically.

During the last stage the user selects the desired mission and roles and is then ready to perform requests.

6.1.7.2 Logging out of EDDS

This scenario shows the interactions between components when a user logs out of the EDDS.

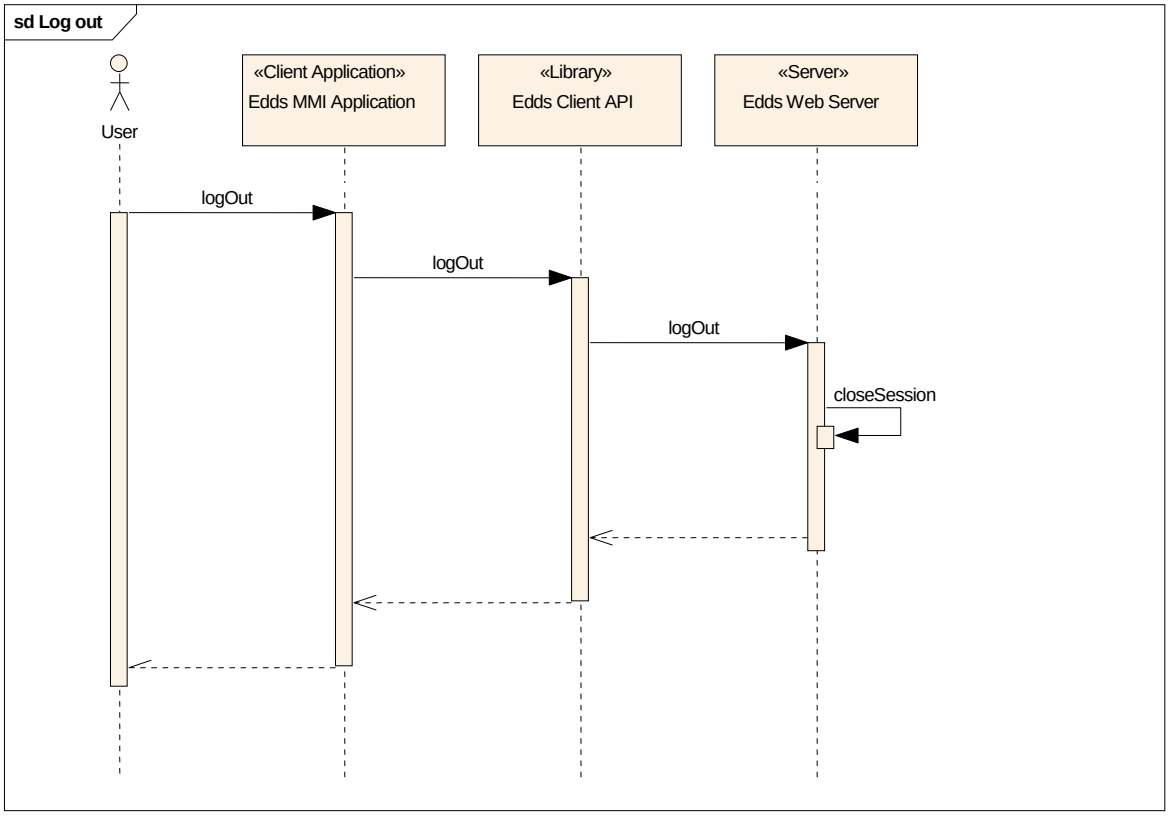

Figure 10 : Logging out of EDDS

The figure above shows that the current session that is open for the user needs to be closed.

The closure of a session does not involve any resource clean up. The client session and the resource management are two independent entities. The created response files can be deleted manually by a user or they will be deleted automatically after a configured period of time has elapsed. Refer to the Configuration and Installation Guide (CIG) [AD-9] for more information.

6.1.7.3 Making a batch request

This scenario shows the interactions between components when a batch request is made from the EDDS Client application. In this particular case, the request is for archived data packet and the delivery mechanism is done through the Delivery Manager, i.e. the response data is pushed to the client by the EDDS Delivery Manager. This scenario places the data provider on the same LAN as the Web Server.

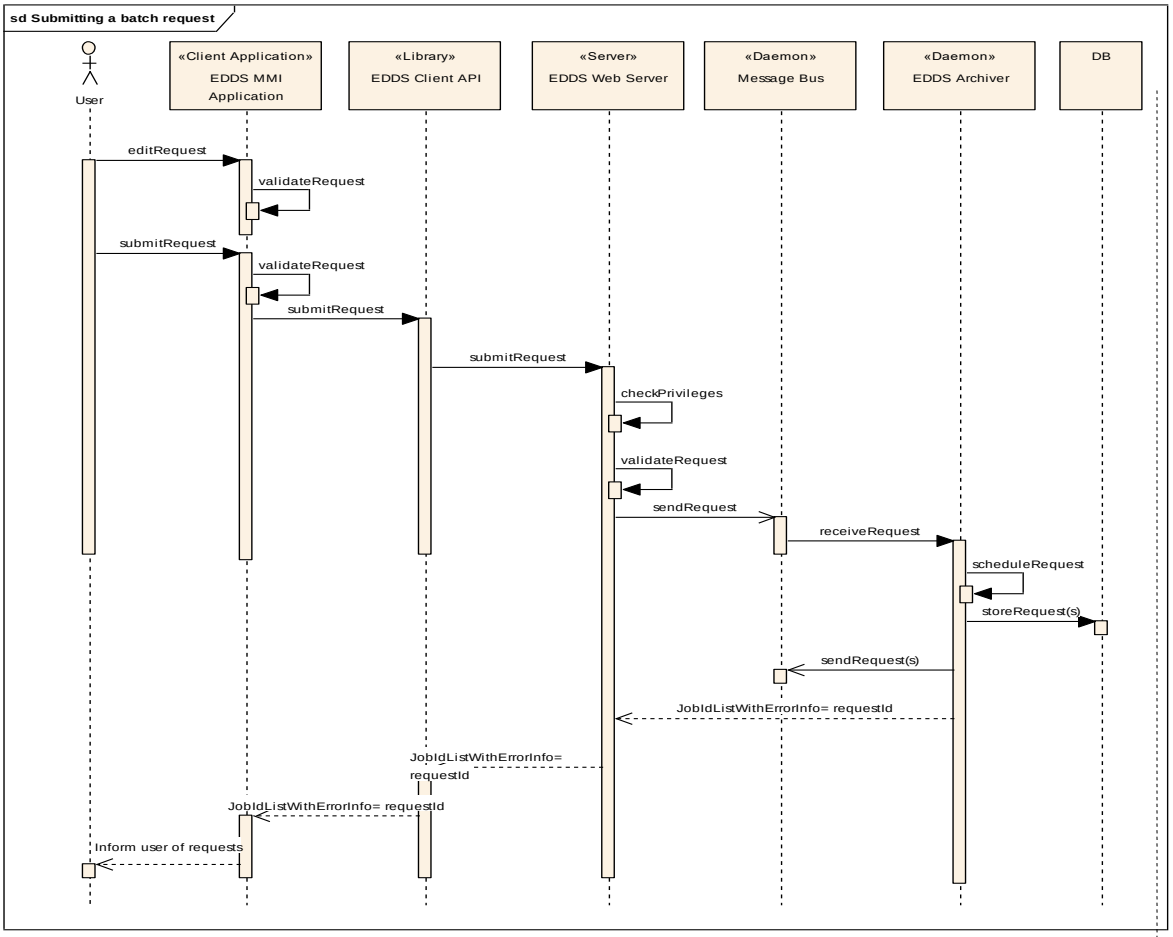

Figure 11 Submitting a Batch Request

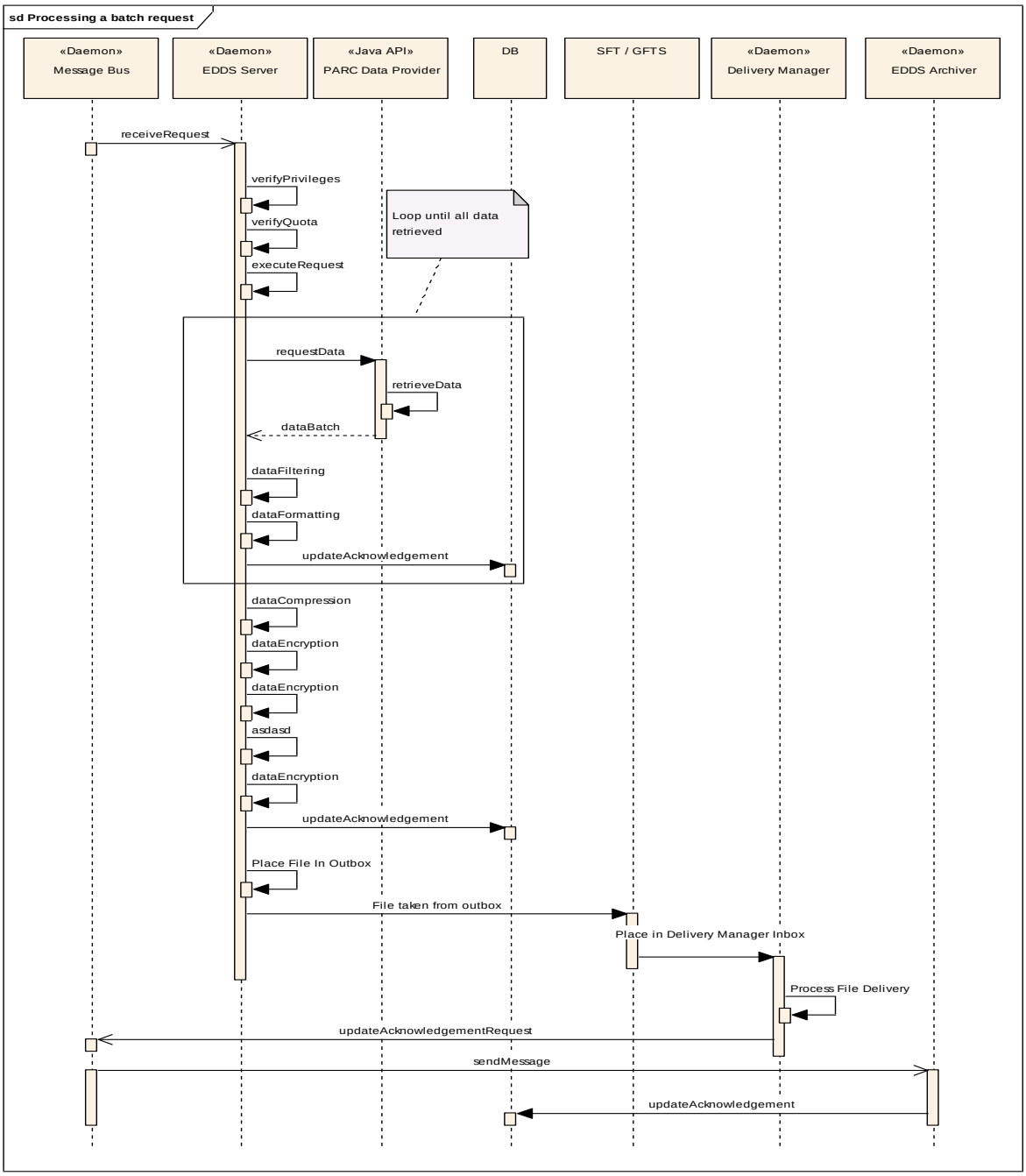

Figure 12 Processing a Batch Request

The User edits the batch request using the EDDS MMI application. The batch request is defined by an XML schema file. This allows validating the request at editing time. The batch request is submitted to the EDDS Web Server through the available EDDS client APIs. As the batch request is received by the Web Server, it is checked to see if the user is authenticated, i.e. if the user has been logged in and has opened a valid session.

If the authentication check passes, the request is placed on the message bus for the EDDS Archiver to process. The EDDS Archiver schedules the request if needed and produces several requests, if a repeating schedule is used. All the requests are given a unique jobId, which are returned to the user. The jobId will allow the user to monitor and control the requested execution at any time. The EDDS Archiver at this point creates a Data Request and an Acknowledgement data structure, which describes the current status of the request. A new record on the database is created for the request. The structure of the Acknowledgement data structure is defined in the EUICD [AD-3]

In case of errors in the request processing, EDDS server will retry processing the request if configured to do so. If no retries should be made or the maximum number of retires have been reached an exception is raised to the EDDS client API, reporting the encountered error. The detail on the name convention used for the Data Request File and the Acknowledgement File can be found in section 6.2.6.10.

The Data Request is received by the EDDS Server when it is time for it to be executed. The EDDS Server connects to the message bus (ActiveMQ) located outside the firewall using the Java JMS API through the usage of a TCP/IP static port (typically 61616, although can be configured in the ActiveMQ configuration). The connection is initiated from the OPS LAN but is not permanent. It is re-established if the connection is lost.

The EDDS Server performs the following steps to execute the batch request:

1. Verifies the request authorisation, i.e. checks the user privileges;
2. Checks the user quota usage and limits
3. Queues the request;
4. Initiates the request execution.

Note that at the end of each step an update of the Acknowledgement File is performed.

The execution of the request consists of routing the request to the corresponding EDDS processor, e.g. PARC request processor, for the actual execution. Note that the mechanism is generic and independent from the data provider allowing the extension for the support of different data providers.

The EDDS Server performs the following steps to execute the batch request:

1. Extracts from the request data necessary to build the data request;
2. Performs the data request, e.g. Packet retrieval;
3. For all the received data, performs the filtering and the formatting;
4. Transforms the resulting files if XML transformation is used;
5. Encrypts and compresses the data if required;
6. Creates the Data Response file.

During the data retrieval, the status of the processing is periodically updated on the EDDS database. In case of errors (e.g. the archive becomes unavailable) the request execution is aborted. Depending on server configurations, it will either report the encountered error to the client or keep trying with given intervals for certain number of times, until the request is processed successfully or maximum number of retries has been reached. During the retry process, the request status stays ACTIVE. However, each time retry is applied, the request acknowledgment gets updated with the number of retries already done and the reason for a retry.

At the end of the data processing, the generated Data Response file is stored on the local file system. At this time the Delivery Manager pushes the Data response file to the client.

6.1.7.4 Making a batch request (data provider behind a firewall)

In case only the Data provider is located behind a firewall (e.g. OPS LAN) the scenario described in the section, 6.1.7.3 has to be extended as follows:

1) Inside the firewall an instance of the EDDS server shall be installed that retrieves the requests from the message bus located outside of the firewall;
2) The request is processed;
3) The response data file (i.e. the retrieved data) is moved from inside to outside the firewall using the SFT application.

6.1.7.5 Making a stream request

This scenario shows the interactions between components when a stream request is made from a generic EDDS Client application. In this particular case, the request is generic and does not refer to any specific data provider. The scenario foresees the data provider located on the same LAN as the Web Server.

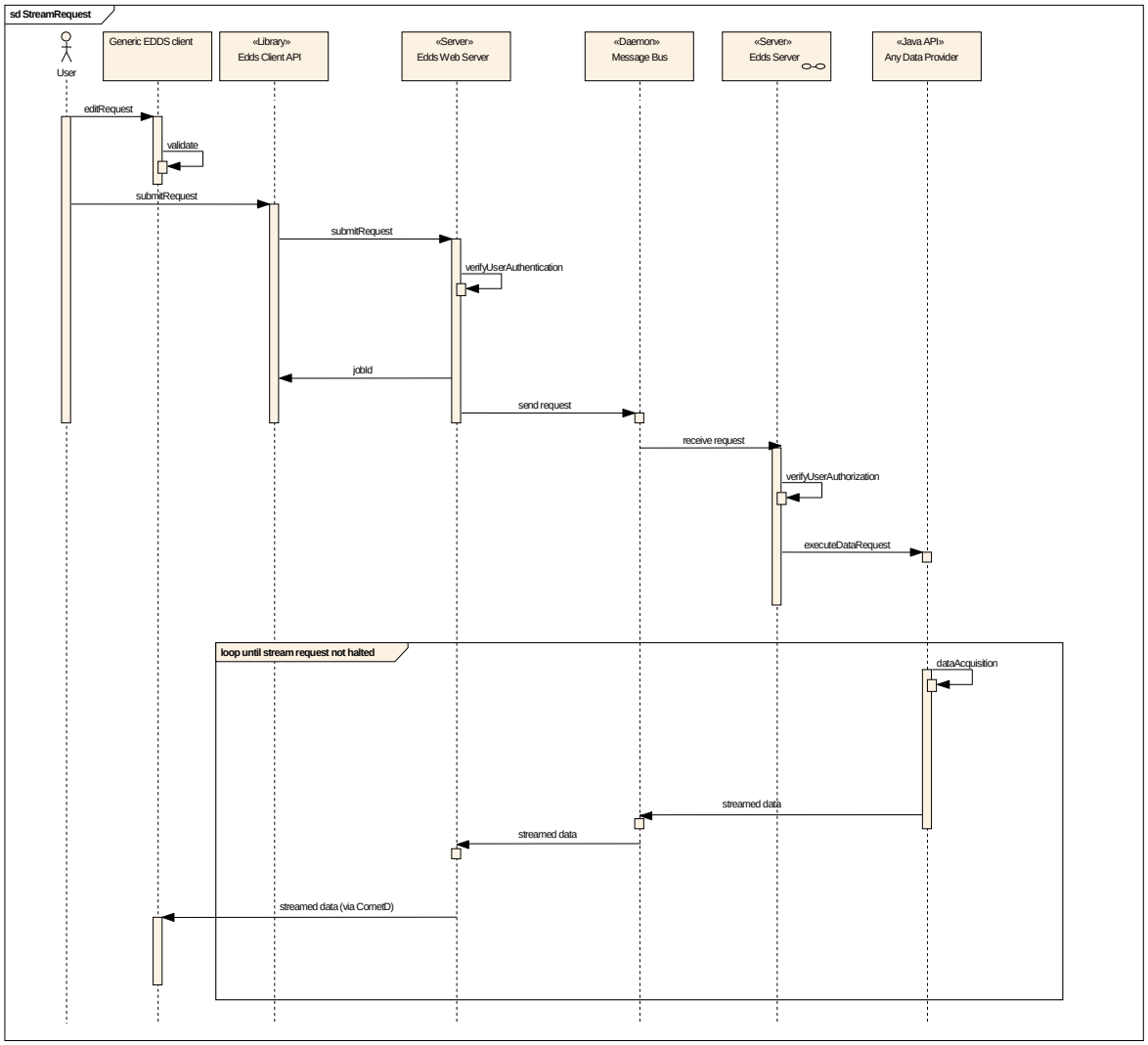

Figure 13 : Stream Request Example

The User edits the stream request. The stream request is described by an XML schema file. This allows validating the request at editing time. The stream request is submitted to the EDDS Web Server through the available EDDS client APIs. As the stream request is received by the Web Server, it is checked if the user is authenticated, i.e. if the user has been logged in and has open a valid session.

If the authentication check passes, it is created a unique jobId, which is returned to the user. The jobId will allow to control the stream request and to retrieve the stream data. The Web server at this point submits the request to the message broker for the EDDS Server to process (via the EDDS Archiver). The Archiver has been omitted from the diagram for simplicity. It is still involved in the process in a similar way to batch requests, saving the request in the database.

In case of errors in the request, processing an exception is raised to the EDDS client API, reporting the encountered error.

The Data Request is extracted and then processed by the EDDS server. The EDDS server performs the following steps to execute the stream request:

1. Verifies the request authorisation, i.e. checks the user privileges;
2. Checks the user quota usage and limits
3. Initiates the request execution.

The execution of the request consists of sending the streaming request to the corresponding Stream Data Provider, for the actual execution. Note that the mechanism is generic and independent from the data provider allowing the extension for the support of different data providers.

The Data provider performs the following steps to execute the stream request:

1. Extracts the relevant data necessary to build the data request;
2. Performs the data acquisition;
3. Streams the data onto the message bus

The Web Server is listening to the message bus for the streamed data and passes the stream to the client via CometD. The client is listening to the stream channel and receives the data as soon as streaming starts. The data is stored in the payload of the message using Google Protocol Buffers (See ICD [AD-3] for information on the data format).

The process of data retrieval continues until explicitly halted or the request expires. Note that the halt operation is not described in the above figure.

It is possible also to use EDDS Stream Client to listen to a stream and periodically dump the data into files. Note that stream client currently supports only TM stream request type. Please refer to the CIG for more information about EDDS Stream Client.

6.1.7.6 Managing Users

This scenario shows the interactions between components when an EDDS user wants to manage the EDDS user accounts.

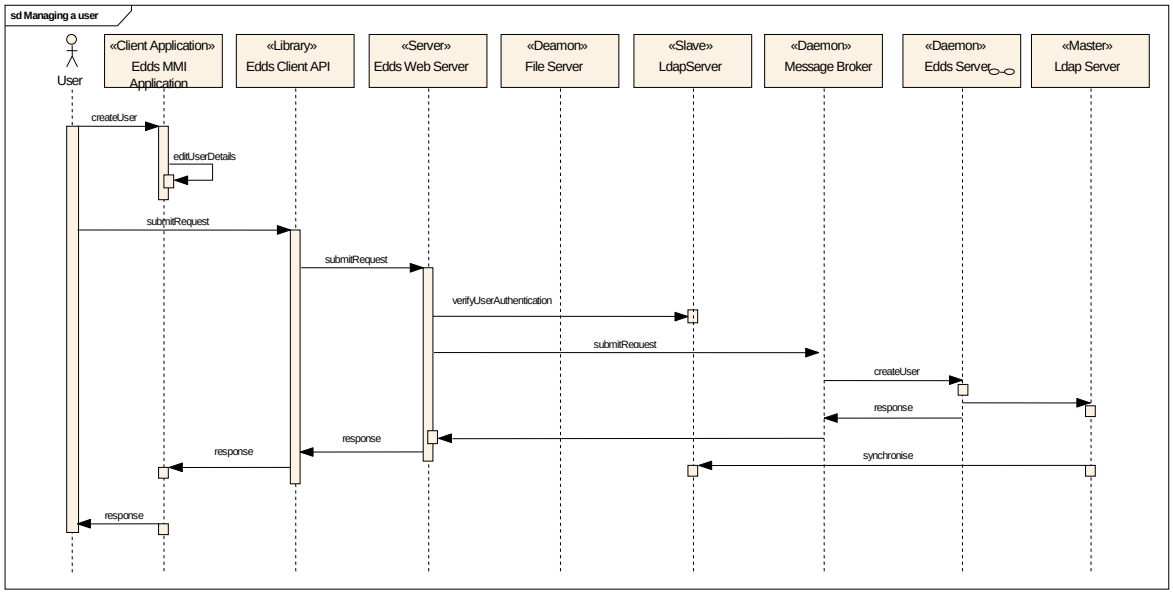

Figure 14 : User Management Example.

In the case of the creation of a new user, the user data is stored in the LDAP server (master) which is located in the OPS LAN. The content of the LDAP server (master) has to be synchronised with the slave located in the RELAY LAN. The synchronisation is initiated from the LDAP master. When the LDAP master notices that there are changes to propagate to the slave LDAP, it opens a connection with the slave and sends the changes, then closes the connection.

6.2 Software Product Components

This section describes the EDDS software components presented in the component overview:

- 6.2.1 EDDS Client Application;
- 6.2.2 EDDS Client API;
- 6.2.3 Secure File Transfer;
- 6.2.4 Request Submitter;
- 6.2.5EDDS Stream Client
- 6.2.6EDDS Web Server;
- 6.2.7 EDDS Archiver;
- 6.2.8 EDDS Server;
- 6.2.9 PARC and Data Provision Data Providers;
- 6.2.10EDDS FARC Interface
- 6.2.11 FARC Data Provider;
- 6.2.12 File System Data Provider;
- 6.2.13 DARC Data Provider;
- 6.2.14SMON Data provider;
- 6.2.15 Delivery Manager.

6.2.1 EDDS Client Application

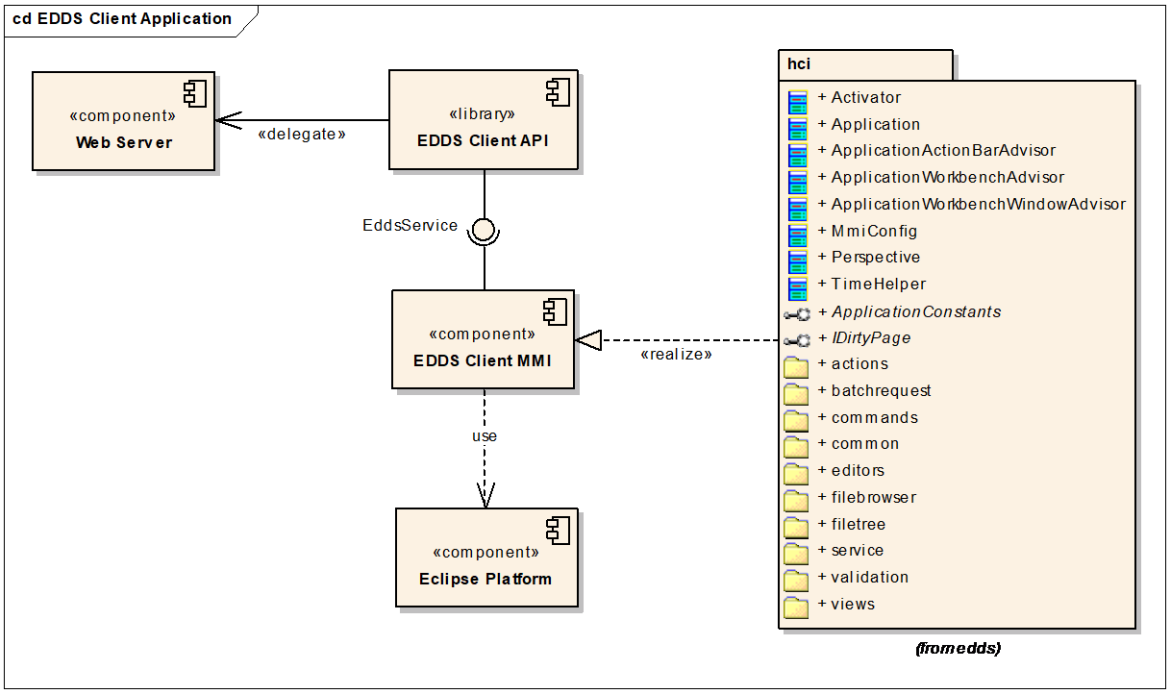

Figure 15 - EDDS Client Application

6.2.1.1 Type

EDDS MMI can be used as an executable Eclipse RCP based program. It is also provided as a Java Web Start Application on EDDS Web Server.

Or alternatively there is a web application of the same MMI which can be used directly in the browser.

6.2.1.2 Purpose

The EDDS Client Application allows end users to create, update and issue batch and stream requests to the EDDS Web Server for processing. It also allows the end user to download responses and perform user management tasks such as creating users and updating mission information. No knowledge of the technical details of the server or knowledge of XML are required.

6.2.1.3 Function

The EDDS Standalone Client Application is downloaded from the EDDS web site as a compressed zip file and, installed on the user's machine in a directory accessible only to that user. This allows several client installations to be held on the same machine, and (if the machine supports multi-user hosting), in principle, supports multiple running instances.

Or alternatively the EDDS MMI Web Application can be used directly in the browser. Then the web server instance can host multiple EDDS client sessions in parallel. The advantage is that there is no effort needed by the end user to install anything on their machine. But that means that there is another backend service to maintain and deploy by the system administrators. Also the server hosting the web application needs to have enough resources to be able to cope with multiple users and perform within user expectations.

The user, in order to use the EDDS Client and be able to submit service requests, has to login submitting his username and the password. The operation is performed through the usage of the EDDS client APIs and involves connection to the EDDS Web Server and the execution of the corresponding login service request. After the user authentication, a session is opened.

After a successful login, the user (depending on her privileges) will be able to:

- Perform/Monitor batch request;
- Perform user management requests;
- Get live notifications from the EDDS system;
- Perform/View stream requests, and save the data to disk;
- Download response files from processed batch requests.

When the user logs out, the EDDS client disconnects from the EDDS Web Server, releasing its session and disposing any used session resource. Information related to the user's requests will be kept by the EDDS Web Server, in order to be able to restore the previous status.

When the session is closed, the cyclic requests are not deleted or removed from the scheduler. The session management and the request management are two separated entities which do not share any resource.

6.2.1.4 Subordinates

The subordinates of this component are:

- EDDS Request Editor (for creating or opening XML request files in a form);
- EDDS Request/Response Tree View (for listing the previously saved XML files);
- EDDS Historical Log View (for retrieving log messages from the server);
- EDDS Parameters View (for displaying TM Parameter definitions);
- EDDS Quota View (for displaying the user's quota usage);
- EDDS Request Status View (for displaying the current status of a submitted request).

These low level components are not described further in the SDD and are described in the JavaDoc for the MMI.

6.2.1.5 Dependencies

This component depends upon:

- Eclipse RCP framework library for standalone installation;
- Eclipse RAP for web application deployment;
- EGOS User Desktop.
- EDDS Client APIs;

6.2.1.6 Interfaces

N/A

6.2.1.7 Resources

The EDDS Standalone Client Application needs network access to the EDDS Web Server.
For the EDDS MMI Web Application, the user needs to access the web server that is hosting the application.

6.2.1.8 References

N/A

6.2.1.9 Processing

N/A

6.2.2 EDDS Client API

6.2.2.1 Type

The component is implemented as a Java library and delivered as a standalone JAR to be included in the EDDS Client Application.

6.2.2.2 Purpose

The EDDS Client API allows third party applications (including the EDDS Client Application) to utilise EDDS's services. The EDDS Client Application, for example, only uses this API to communicate with the EDDS Web Server. It does not use any other "private" API only available to the Client Application.

6.2.2.3 Function

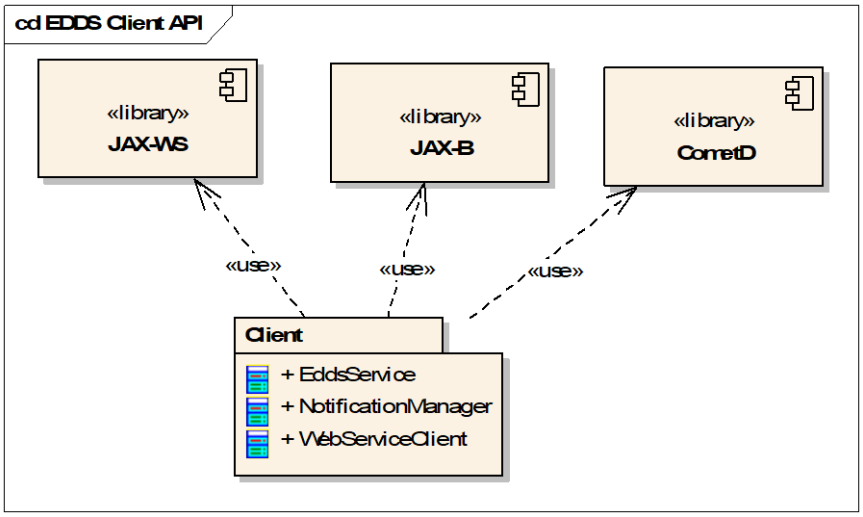

Figure 16 - EDDS Client API

The main function of the EDDS Client APIs is to provide to an EDDS client the means to access the EDDS services. The APIs is a façade to the EDDS Web Services providing the following functionality:

- Hides the complexity associated with the access to the EDDS web services;
- Supports the submission and validation of the EDDS request (batch, stream and management);
- Supports the retrieval of the batch responses;
- Supports the acquisition and buffering of the stream data;
- Provides notifications interface for client application

6.2.2.4 Subordinates

The subordinates of this component are:

- EDDS Request APIs (Batch, Stream, User Management);
- EDDS Response APIs (Batch, Stream, User Management).
- EDDS Notifications service

6.2.2.5 Dependencies

The EDDS Client APIs are wrappers to the models and interfaces generated from the WDSL service files (see Section 8.2.1 of the EUICD [AD-03] for more information). On the top of these classes, they provide ad-hoc behaviours in order to add specific functionalities (i.e. logging, SSL authentication …)

The package uses the JAX-WS library for the management of the SOAP requests and the JAXB library for the management of the XML.

Client APIs also depend on CometD Java client implementation of Bayeux protocol. This enables clients to receive asynchronous notifications initiated by the EDDS system or other clients.

6.2.2.6 Interfaces

The interfaces of this component are:

- WSDL service
- NotificationManager

6.2.2.7 Resources

The EDDS Client API needs network access to only the EDDS Web Server.

6.2.2.8 References

None

6.2.2.9 Processing

The library is a façade to the EDDS functionalities. The class EddsService represents the access point of the library and acts as a singleton in order to keep trace of the session management.
The session creation is requested by the library during the logIn operation and closed during the logOut operation. After this, the session handling is completely transparent to the user and it is needed on the Web Server layer to keep trace of authenticated users. In case a user access a service without being logged in the Web server, the service will return always a SessionFault message and the library will throw a SessionFault exception.
The library hides all the complexity of the XML and SOAP management, so the resulting code a user has to implement is minimal. Code examples on the usage of the library are described in the EUICD [AD-03] in the section “EDDS Client API – code examples”.

NotificationManager provides interface for clients to listen to asynchronous push notifications coming from the EDDS system. It uses the same session to authenticate with the web server as EddsService.

6.2.2.10 Data

Due to the relevance of the response file for the final user, the details of the response file can be found on the EUICD [AD-3].

6.2.3 Secure File Transfer

6.2.3.1 Type

This component is an executable program. It is not an EDDS component but this application or a similar application (e.g. GFTS) is required depending on the EDDS deployment.

6.2.3.2 Purpose

As defined in the SFT documentation [AD-8]

6.2.3.3 Function

The main function of the Secure File Transfer is to move data across the firewall, which separates the OPS LAN and the RELAY LAN respecting the ESOC security policy.

The SFT has to transfer the following data:

- From the OPS LAN to the RELAY LAN: The EDDS request data responses (Batch and Stream)
- From the OPS LAN to the RELAY LAN: The data file necessary to synchronise the RELAY LAN data providers with the OPS LAN data providers if required.

In order to optimise the data transfer multiple instances of the SFT can be instantiated to separate, for example, the transfer between the stream data responses and the batch data responses.

6.2.3.4 Subordinates

None

6.2.3.5 Dependencies

The SFT is an independent stand-alone application. See [AD-10]

6.2.3.6 Interfaces

N/A

6.2.3.7 Resources

For the Secure File Transfer (SFT) program to work, it requires a connection to another SFT instance over TCP/IP.

6.2.3.8 References

N/A

6.2.3.9 Processing

N/A

6.2.3.10 Data

N/A

6.2.4 Request Submitter

6.2.4.1 Type

This component is an executable program.

6.2.4.2 Purpose

Allows both EDDS and GDDS requests to be issued by placing the XML file inside a directory the Request Submitter is polling. It is also possible to place requests to resume, cancel and suspend previously submitted requests.

6.2.4.3 Function

Any valid request files in the polling directory will be processed and submitted to the EDDS Server. Responses are returned as defined in the request.

The processed request file will be moved to an 'Outbox' directory. Any failed requests will be moved to a 'Failed Request' directory.

6.2.4.4 Subordinates

None

6.2.4.5 Dependencies

This program depends upon:

- Apache Camel

6.2.4.6 Interfaces

Esa.egos.edds.requestsubmitter

6.2.4.7 Resources

N/A

6.2.4.8 References

N/A

6.2.4.9 Processing

The request file is picked up from the polling directory and validated against the EDDS request schema. Valid requests will be issued to the EDDS Server for processing.
Those requests that fail the EDDS schema validation (likely to be in GDDS format) are validated against the GDDS format schema. Valid files are then transformed into EDDS format using the XSLT file provided and then issued for processing.
All successful requests will be moved to an Outbox directory, all failed requests will be moved to a Failed request directory.

6.2.4.10 Data

N/A

6.2.5 EDDS Stream Client

6.2.5.1 Type

This component is an executable program.

6.2.5.1 Purpose

Stream Client is able to listen to an existing stream request and dump data into file periodically. So, the main purpose of this client is to save stream data to files in various formats.

6.2.5.2 Function

Stream client takes as input an ACTIVE stream request's id and starts to receive data for that stream. It is possible to define the output folder, output format and a time interval inside the config.properties file of the client. Stream client is not aware about the state of the request it is listening to, so if the request is cancelled or the request terminates then Stream client must be shut down manually. If no data has been received within a period of time, then no file will be saved for that time interval from the Stream Client.

6.2.5.1 Subordinates

None

6.2.5.1 Dependencies

EDDS Client Api

6.2.5.1 Interfaces

esa.egos.edds.streamclient

6.2.5.1 Resources

The EDDS Stream Client needs network access to the EDDS Web Server.

6.2.5.1 References

N/A

6.2.5.1 Processing

Stream Client authenticates to EDDS Web Server using username and password then registers itself as a listener of the stream request identified by the request id. While EDDS Stream Client is alive, it waits for new streaming data from web server and saves it into a temporary response file (in protobuf format). At regular time interval, converts data from the temporary file to the indicated format and save them to the output folder.

6.2.5.1 Data

It only supports TM type stream requests and xml ,Protobuf and GDDS Binary output format.

6.2.6 EDDS Web Server

6.2.6.1 Type

The component is a Java Web Application delivered as a WAR to be run in a JAX-WS compliant servlet engine.

6.2.6.2 Purpose

The EDDS Web Server provides access to the EDDS services by EDDS client applications.

6.2.6.3 Function

The EDDS Web Server acts as the portal to EDDS services. The services offered use Web Services technology and standard web page access. The Web Server can support secure communication with clients via the HTTPS protocol and digital certificates to authenticate clients.

The Web Server provides a mechanism to allow users to perform service requests (batch, stream and user management) and get the service request responses. The EDDS web server provides the following category of services:

- Authentication services;
- Batch Request services;
- Stream Request services;
- User Management services.
- Notifications Services

6.2.6.4 Subordinates

The subordinates of this component are:

- Bayeux server for asynchronous notifications. Bayeux is a protocol for transporting asynchronous messages (primarily over HTTP), with low latency between a web server and web clients.

6.2.6.5 Dependencies

EDDS Web Server uses the JAX-WS specification for the design and development of the Web Services.

EDDS Web Server uses the CometD implementation of Bayeux server (see http://cometd.org/).

EDDS Web Server uses the Spring JMS Template for sending and receiving messages on the message bus.

6.2.6.6 Interfaces

The interface of this component is:

esa.egos.edds.ws.server.EddsServiceImpl

The clients connect to EDDS Web Server through ports configured in tomcat configuration (typically port 8080 for HTTP and 8443 for HTTPS), so the ports have to be opened in the firewall or forwarded to EDDS Web Server for the clients to be able to send requests from the external network.

6.2.6.7 Resources

None

6.2.6.8 References

None

6.2.6.9 Processing

An off-the-shelf web server (Apache Tomcat) accepts HTTP(S) requests from users through the available EDDS Client APIs. Client certificates are assumed to have been issued by an ESA authority and connections will be rejected if the certificate supplied has not been signed by ESA.

The EDDS Web Server is considered to be non-critical operational software that is normally run on the RELAY LAN and OPS LAN. The EDDS Web Server communicates with EDDS Servers on the OPS LAN by writing the service requests into an XML data structure, which are then passed onto a message bus. Under normal circumstances, it is expected that all data transfer between clients and the EDDS Web Server uses the HTTPS protocol. However, the design allows a mission to use a pure HTTP protocol (without data encryption) by deploying and configuring a mission-specific Web Server.

EDDS Web Server can send asynchronous messages to client applications notifying them of events in the EDDS components.

6.2.6.10 Data

6.2.6.10.1 Response File Name Convention

Due to the relevance of the response file for the final user, the details of the response file can be found on the EUICD [AD-03].

6.2.7 EDDS Archiver

6.2.7.1 Purpose

The EDDS Archiver is a Java application that takes requests submitted by the user via the Web Server and schedules them, archives the request into the database and sends the request onto the message bus for processing by the EDDS Server.

6.2.7.2 Function

The EDDS Archiver provides services to other EDDS components that do not have access to the database (i.e. the web server and delivery manager) as well as archiving log messages into the database and saving initial requests from the web server into the database, before notifying the EDDS Server that the request can be processed.

6.2.7.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS Acknowledgement Manager;
- EDDS Database Manager;
- EDDS Scheduling Utility;

6.2.7.4 Dependencies

The EDDS Archiver depends on Apache Camel for processing the messages received.

6.2.7.5 Interfaces

The EDDS Archiver responds exclusively to messages sent on the message broker. The messages received are routed by Camel to be processed. In particular, the archiver responds to messages sent on the following topics:

- esa.egos.edds.logs.deliverymanager
- esa.egos.edds.logs.eddsserver
- esa.egos.edds.logs.webserver

These messages are log messages, and are saved into the database so that they can be retrieved if needed later.

Before a request from the user via the web server can be processed by the EDDS Server, it needs to be stored in the database. This is one of the EDDS Archiver's roles. The typical pattern is for the request to be placed on a ".tobeprocessed" queue, which the Archiver listens to. The request message is then filed in the database, and if necessary additional headers are added (for example, state updates) and the message is then placed on the appropriate queue or topic as necessary for the EDDS Server.

The EDDS Archiver responds to messages sent to the following queues:

- esa.egos.edds.request.usermanagement

  Responds to requests to update the Master LDAP with changes to User Management. The payload is a `AcountRequestMessagePart` object detailing the changes. The reply message contains a String with any error messages, or an empty String if there were none.

- esa.egos.edds.request.usermanagement.suspenduser

  Responds to requests to suspend a user account after the user has logged in incorrectly too many times. There should be a single String property called "UserName" containing the username of the user to suspend.

- esa.egos.edds.request.usermanagement.updateincorrectlogins

Responds to requests to update the number of incorrect logins for a user. There should be a single String header property with a key of “UserName” containing the username of the user to update and an Integer payload containing the number to update LDAP with.

- esa.egos.edds.request.usermanagement.updatelastlogin

  Responds to requests to update the date the user last logged in. There should be a single String header property with a key of “UserName” containing the username of the user to update and a Date payload containing the date to update LDAP with.

- esa.egos.edds.update.ack

  Response to requests to update acknowledgements in the database. The payload of the message should contain a `UpdateAckRequest` object. A single String header called “RequestId” should be set containing the ID of the request.

- esa.egos.edds.retrieve.acknowledgement

  Responds to requests for acknowledgement information from the database for a particular request. The payload of the message should contain a `RequestId` object. The `AcknowledgementPart` from the database is sent on the JMSReplyId queue in the original message.

- esa.egos.edds.retrieve.statuses

  Responds to requests for status information from the database. The payload of the message should contain a `StatusRequestRec` object, detailing the information needed – either a list of request IDs, or the list of missions and the username to get the requests of. If the mission list is empty, all the missions are retrieved. Two Long properties should be set in the message header – “FromTime” and “ToTime”: This specifies the time range for the request, and are optional. If not specified, all the requests are retrieved from either the first request (in the case of from time) to the last request in the database (in the case of to time). Two more String properties should be set – a unique ID of the request (any random string) and the name of the user issuing the request (UserName).

  The response is posted onto a separate queue (esa.egos.edds.response.statuses). In case an error occurred, a message is posted onto the queue esa.egos.edds.response.statuses.error. The message contains two header properties – UserName for the name of the user that issued the original request and RequestId – the unique ID provided in the original request. This allows CometD to ensure the messages are sent to the correct user. The payload of the message is a byte array that can be decoded by creating a new `RequestInfoRecs` object from edds-ws-common and passing the byte array as a parameter to the constructor. The individual status request entries can be obtained by calling the `getEntries()` method on this object.

- esa.egos.edds.retrieve.state

  Responds to requests for the current state of a specific request, unlike the previous queue which returns status information for a range of requests. The payload of the message should contain the request ID as a String. The reply to the message will be the current status as a String.

- esa.egos.edds.retrieve.jobids

  Same as esa.egos.edds.retrieve.statuses, however a set of request IDs as a String is returned instead.

- esa.egos.edds.retrieve.requestinfo

  Responds to requests for the information of a particular request. The payload of the message should be a `RequestId` object for the request to retrieve the details of. The response is a byte array encoded using Google Protocol Buffers for speed. The information can be decoded by creating a new `RequestInfoRec` object in edds-ws-common and passing the byte array as a parameter to the constructor.

- esa.egos.edds.retrieve.batch.request

Responds to requests for the `RequestMessagePart` of a request. The payload of the message should be a `RequestId` object for the request to retrieve the `RequestMessagePart` of. The response is the `RequestMessagePart` for the request.

- esa.egos.edds.retrieve.stream.request

  Responds to requests for the `StreamRequestMessagePart` of a request. The payload of the message should be a `RequestId` object for the request to retrieve the `StreamRequestMessagePart` of. The response is the `StreamRequestMessagePart` for the request.

- esa.egos.edds.retrieve.ongoingrequests

  Responds to requests for the number of ongoing (i.e. ACTIVE) requests for a particular mission / user / role combination. The payload should be simply an empty String, while three String properties should be set called MISSION_NAME, UserName and RoleName.

- esa.egos.edds.retrieve.historicallogs

  Responds to requests for the historical log messages. The payload for the message should be a List of mission names as a String (can be empty). Two Long properties should be set – the FromTime and the ToTime – the time range for the period to retrieve the log messages for. These should both be simply a Unix timestamp obtained from `Date.getTime()`.Two more String properties should be set – a unique ID of the request (RequestId) and the name of the user issuing the request (UserName).

  The historical log messages are posted onto a separate queue (esa.egos.edds.response.historicallogs) one by one. Each message contains two header properties – UserName for the name of the user that issued the original request and RequestId – the unique ID provided in the original request. This allows CometD to ensure the messages are sent to the correct user. The payload of the message is a byte array that can be decoded by creating a new `SystemLogRec` object from edds-ws-common and passing the byte array as a parameter to the constructor.

- esa.egos.edds.update.state

  Responds to requests to update the state of a request in the database. Two String header properties should be set – RequestId and State containing the ID of the request to update and the State to update the request to respectively. The payload of the message should be a String containing the exception stack trace to include, if applicable. If not applicable, include an empty String.

- esa.egos.edds.update.ack

  Responds to requests to update the acknowledgement in the database. The message must be an UpdateAckRequest object and have a header – RequestId. This object will be used to make an incremental update to the acknowledgement in the database. EDDS Archiver is the only application performing updates on acknowledgement to prevent loss of data when two or more applications try to update the same record in database.

- esa.egos.edds.request.db.delete

  Responds to requests to delete a request entry in the database. This is normally called by the Delivery Manager. The payload of the message is the request ID as a String of the request to delete in the database.

- esa.egos.edds.request.batch.unprocessed

  Responds to requests from the web server to add a new batch request. The payload of the message should be a `RequestMessagePart` containing the details of the batch request. The EDDS Archiver will then determine the priority of the message based on the role used, pass the request to the schedule utility which will then create a new request for each scheduled execution time (if necessary) and then add the requests to the database. The EDDS Archiver will then create a new message with the following header properties set:

  MISSION_NAME – the name of the mission that the request is for

RequestId – the unique ID of the request

RequestType – the type of request (i.e. BatchRequest)

RequestSubType – the sub-type of the request (e.g. PktTc)

If the request is scheduled, a job will be scheduled with the Quartz Scheduler so that the request is processed at the right time.

The message is then sent to the esa.egos.edds.request.batch.tobeprocessed queue which the EDDS Server(s) are listening to.

- esa.egos.edds.request.stream.unprocessed

  Responds to requests from the web server to add a new stream request. The payload of the message should be a `StreamRequestMessagePart` containing the details of the stream request. The EDDS Archiver will then determine the priority of the message based on the role used, pass the request to the schedule utility which will then add the request to the database. The EDDS Archiver will then create a new message with the following header properties set:

  MISSION_NAME – the name of the mission that the request is for

  RequestId – the unique ID of the request

  RequestType – the type of request (i.e. StreamRequest)

  RequestSubType – the sub-type of the request (e.g. ParamStream)

  If the request is scheduled, a job will be scheduled with the Quartz Scheduler so that the request is processed at the right time..

  The message is then sent to the esa.egos.edds.request.stream.tobeprocessed queue which the EDDS Server(s) are listening to.

- esa.egos.edds.request.cancel.unprocessed

  Responds to requests from the web server to cancel a request. The payload should be a `CancelPart` object, containing the details of the request to cancel, and two String properties – the TargetRequestId (the ID of the request to cancel) and the UserName (the user who issued the request). The EDDS Archiver will then check the current state of the request. If it is SUBMITTED, it will simply set the state to CANCELED. If it had been scheduled, it will remove the scheduled message. If the state is ACTIVE or QUEUED, the EDDS Archiver will create a new message with the following header properties set:

  MISSION_NAME – the name of the mission that the cancellation request is for

  TargetRequestId – the ID of the request to cancel

  UserName – the name of the user submitting the request.

  The payload is the passed `CancelPart` object. The message is then sent on either the esa.egos.edds.request.batch.cancel.tobeprocessed topic, or the esa.egos.edds.request.stream.cancel.tobeprocessed topic as appropriate which all the EDDS Server(s) are listening to and will receive the cancellation message.

- esa.egos.edds.request.suspend.unprocessed

  Responds to requests from the web server to suspend a request. The payload should be a `SuspendPart` object, containing the details of the request to suspend, and two String properties – the TargetRequestId (the ID of the request to suspend) and the UserName (the user who issued the request). The EDDS Archiver will then check the current state, type and sub type of the request. If the request is a suspendable batch request and is ACTIVE the EDDS Archiver will create a new message with the following header properties set:

  MISSION_NAME – the name of the mission that the suspension request is for

  TargetRequestId – the ID of the request to suspend

  UserName – the name of the user submitting the suspend request.

The payload is the passed `SuspendPart` object. The message is then sent to the esa.egos.edds.request.batch.suspend.tobeprocessed topic which all the EDDS Server(s) are listening to and will then receive the suspension message.

- esa.egos.edds.request.resume.unprocessed

  Responds to requests from the web server to resume a suspended or failed request. The payload should be a `ResumePart` object, containing the details of the request to resume, and two String properties – the TargetRequestId (the ID of the request to resume) and the UserName (the user who issued the request). The EDDS Archiver will then check the current state, type and sub type of the request. If the request is a resumable batch request and is SUSPENDED or in one of the resumable error states (errors that occur while EDDS server is processing request, that means that errors that have to do with invalid requests and etc. are not resumable) the EDDS Archiver will create a new message with the following header properties set:

  MISSION_NAME – the name of the mission that the resume request is for

  TargetRequestId – the ID of the request to resume

  UserName – the name of the user submitting the resume request.

  The payload is the passed `ResumePart` object. The message is then sent to the esa.egos.edds.request.batch.resume.tobeprocessed topic which all the EDDS Server(s) are listening to and will then receive the resume message.

- esa.egos.edds.request.batch.deleterequest.unprocessed

  Responds to requests from the web server to delete a request in the database. The payload of the message is a `JobIdPart` object, detailing the request to be deleted with two String properties – TargetRequestId (the ID of the request to be deleted) and UserName (the name of the user issuing the request). The EDDS Archiver inserts the request into the database, and sends a message on the esa.egos.edds.request.batch.deleterequest.tobeprocessed queue for the Delivery Manager to process with the following properties set:

  RequestId – the ID of the deletion request

  TargetRequestId – the ID of the request to delete

  The payload of the message is the passed `JobIdPart`.

- esa.egos.edds.request.batch.deletedata.unprocessed

  Responds to requests from the web server to delete the response data of a completed request. The payload of the message is a `JobIdPart` object, detailing the request to delete the response data of with two String properties – TargetRequestId (the ID of the request to have the response data deleted) and UserName (the name of the user issuing the request). The EDDS Archiver inserts the request into the database, and sends a message on the esa.egos.edds.request.batch.deletedata.tobeprocessed queue for the Delivery Manager to process with the following properties set:

  RequestId – the ID of the data deletion request

  TargetRequestId – the ID of the request to delete the response data of

  The payload of the message is the passed `JobIdPart`.

- esa.egos.edds.delivery.responsedeletion.scheduled

  Responds to messages from the Delivery Manager to register a response file for deletion. This is then scheduled for later execution by the Archiver using Quartz. When the time to delete the response files for a request arrives, the job is fired by Quartz sending a message to the Delivery Manager to delete the files on the queue esa.egos.edds.delivery.responsedeletion.scheduled.tobeprocessed. The payload of the message is not used, only the message headers. The headers expected are:

RequestId – the ID of the request the response file relates to (String)

AmountOfData – the amount of data in bytes the response file uses (Long)

UserName – the user who created the request

RoleName – the name of the role used for the request (can be null)

FileName – the name of the file to delete, including full path

DeletionTime – the time to delete the response file. Unix timestamp in milliseconds (Long)

- esa.egos.edds.delivery.responsedeletion.unschedule

  Responds to requests to cancel the scheduled deletion of the response files for a request. This is needed should the user decide to delete the response data or request manually, or if the database clean-up job is fired. Messages contain no payload and a single header property: RequestId - the ID of the request the scheduled job relates to

6.2.7.6 Resources

None

6.2.7.7 References

None

6.2.7.8 Processing

Messages are received on the queues and topics listed above. They are then routed via Camel to the Java classes, which processes the message, and if necessary returns information. This information is then sent by Camel onto the reply queue specified in the header of the initial message.

The EDDS Archiver needs access to the database and Master LDAP server. When processing new requests from the web server, the EDDS Archiver will schedule the request if needed according to the scheduling criteria defined in the EUICD [AD-3]. At the end of the scheduling step, if any scheduling activity has been performed, an update of the Acknowledgement file is produced.

6.2.7.8.1 Acknowledgment Processing

The EDDS Archiver has the responsibility of creating the acknowledgement messages when a new request is received from the client. Note that there is only ever one acknowledgement per request. The acknowledgement is updated over time with the latest information. Steps involved are:

(a) receiving the acknowledgement update messages generated by EDDS components (EDDS Server, Delivery Manager, EDDS Web Server),

(b) Transforming the update messages into an acknowledgement model object, that depicts the latest state of the processing of the request,,

(c) Storing into the EDDS DB (see 6.1.7.3) where they can be at any time accessible to be delivered to the User,

(d) Sending update notifications to the clients when the acknowledgment is updated.

6.2.8 EDDS Server

6.2.8.1 Purpose

The EDDS Server is the Java application that processes requests that need to be processed for the mission and request sub types it is currently configured for. It produces response files that the Delivery Manager delivers to the destination requested by the user.

6.2.8.2 Function

The main functions of the EDDS server are:

- Process the services requests (Batch and Stream);
- Validate the incoming service request against the user privileges and quotas;
- Monitor and Control the requests which have been scheduled and/or under execution;
- Update the acknowledgement of requests to reflect the current state;
- Maintain the status of the requests under execution;
- Initiate the execution of the service request. The activity consists of the distribution of the service request to the corresponding data provider;
- Execute all the EDDS report service requests.

6.2.8.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS Request Manager (Batch, Stream, User Management);
- EDDS Acknowledgement Manager;
- EDDS Request Handler (Batch, Stream, User Management);
- EDDS Database Manager

The figure below describes EDDS Server sub components and its dependency

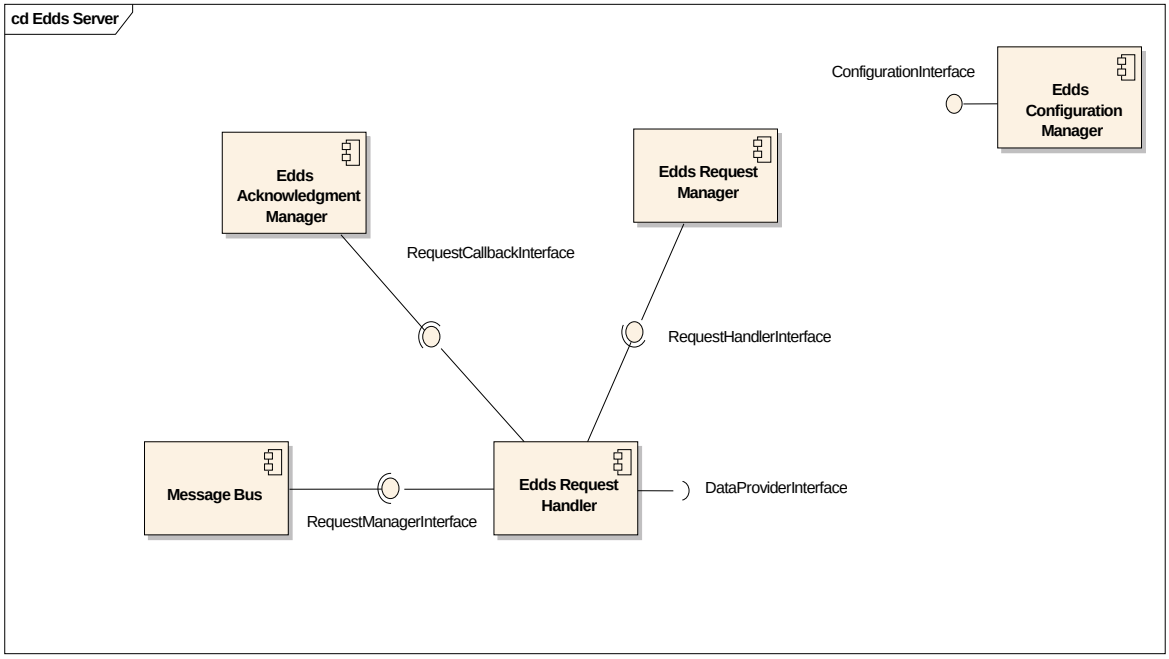

Figure 17 EDDS Server

6.2.8.4 Dependencies

This component depends upon:

- Third-party Java APIs for interfacing with the archives;
- EDDS Model classes, which are generated from the EDDS Web services schema files

6.2.8.5 Interfaces

The interfaces of this component are:

- esa.egos.edds.server.request.RequestHandlerInterface
- esa.egos.edds.server.request.DataProviderInterface
- esa.egos.edds.server.request.RequestCallbackInterface
- esa.egos.edds.server.RequestManagerInterface

6.2.8.6 Resources

N/A

6.2.8.7 References

N/A

6.2.8.8 Processing

The EDDS Server responds to messages sent to the following queues:

- esa.egos.edds.retrieve.transformations

  Responds to requests for the available transformation (XSLT) files for request sub type specified and returns a list of names. The payload of the message should be the request sub type as a String. A List of Strings is returned with the names of the available transformation files for the request sub type.

- esa.egos.edds.request.usermanagement

  Responds to requests to update the Master LDAP with changes to User Management. The payload is a `AcountRequestMessagePart` object detailing the changes. The reply message contains a String with any error messages, or an empty String if there were none.

- esa.egos.edds.request.usermanagement.suspenduser

  Responds to requests to suspend a user account after the user has logged in incorrectly too many times. There should be a single String property called “UserName” containing the username of the user to suspend.

- esa.egos.edds.request.usermanagement.updateincorrectlogins

  Responds to requests to update the number of incorrect logins for a user. There should be a single String header property with a key of “UserName” containing the username of the user to update and an Integer payload containing the number to update LDAP with.

- esa.egos.edds.request.usermanagement.updatelastlogin

  Responds to requests to update the date the user last logged in. There should be a single String header property with a key of “UserName” containing the username of the user to update and a Date payload containing the date to update LDAP with.

- esa.egos.edds.retrieve.dataspaces

  Responds to requests to get a list of dataspaces for the specified request type. There should be a String header property with a key of “MISSION_NAME” containing the name of the mission to

fetch the dataspace names for. The “DomainId” header has to be specified with the current domain if the request is sent for PARC requests. The payload returned is a `List <String>` of the dataspace names. They are ordered so that the first entry is the active (current in PARC) dataspace.

- esa.egos.edds.retrieve.consolidation

  Responds to requests to get the last consolidation time from DARC database for the given dataspace, or for the active one if none specified. There should be a String header property with a key of “MISSION_NAME” containing the name of the mission to fetch the dataspace names for. The last consolidation time is sent as Long, and depending on the configuration of *edds.darc.post.2.3.0,* it will either contain the microseconds or not.

- esa.egos.edds.request.quota

  Responds to requests to update the disk space usage in the Master LDAP. The payload is a `DiskSpaceUsageInfo` object containing the name of the mission, username, role and disk space change information.

- esa.egos.edds.retrieve.paramdef

  Responds to requests for parameter definitions from the DARC. The message should contain three String properties –RequestId – the unique ID of the request, UserName – the name of the user initiating the request, MISSION_NAME – the name of the mission that the definitions should come from.

  The parameter definitions are sent onto a queue (esa.egos.edds.response.paramdef) with three String header properties – MISSION_NAME, RequestId and UserName as passed in the original message and the payload is a byte array. The byte array is all of the parameter definitions encoded using Google Protocol Buffers for speed. The information can be decoded by creating a new `ParamDefRecs` object from edds-ws-common and passing the byte array as a parameter to the constructor. The individual parameter definition entries can be obtained by calling the `getEntries()` method on this object.

- esa.egos.edds.retrieve.farccat

  Responds to requests for catalogue information from the FARC. The message should contain three String properties –DomainId – the domain to retrieve the catalogue for, or “-1” for the “No Domain” within the FARC, MISSION_NAME – the name of the mission that the catalogue should come from and RequestId – a unique identifier for this request.

  The response is posted onto a separate queue (esa.egos.edds.response.farccat). In case an error occurred, a message is posted onto the queue esa.egos.edds.response.farccat.error. The message contains two header properties – UserName for the name of the user that issued the original request and RequestId – the unique ID provided in the original request. This allows CometD to ensure the messages are sent to the correct user. The payload of the message is a byte array that can be decoded by creating a new `FarcCatRecs` object from edds-ws-common and passing the byte array as a parameter to the constructor. The individual catalogue entries can be obtained by calling the `getEntries()` method on this object.

- esa.egos.edds.send.acknowledgements

  Responds to requests to send an e-mail. The payload of the message is a `MailMessage` object. If the property esa.egos.edds.acknowledgement.mail.transform is set to true in the EDDS properties file, then the mail message is transformed using the XSLT file called “AcknowledgementPart.xsl” placed in the directory specified in the EDDS property EDDS_MISSION_XSL_DIRECTORY. The mail message is then placed in the queue “esa.egos.edds.email.queue” so that it can be sent by an EDDS component that has access to a mail server.

- esa.egos.edds.request.batch.tobeprocessed

  Responds to requests for batch requests that have been pre-processed by the EDDS Archiver. The EDDS Server starts the process of retrieving the data requested. The processing is done as

soon as the message is received. For scheduled messages, the message will have had its delivery delayed until it is ready to be executed.

- esa.egos.edds.request.stream.tobeprocessed

  Responds to requests for stream requests that have been pre-processed by the EDDS Archiver. The EDDS Server starts the process of retrieving the streamed data requested. The processing is done as soon as the message is received. For scheduled messages, the message will have had its delivery delayed until it is ready to be executed.

- esa.egos.edds.send.subscriptionnotifications

  Responds to requests to send a FARC Subscription notification via e-mail. The payload of the message is a `MailMessage` object. If the property esa.egos.edds.subscription.mail.transform is set to true in the EDDS properties file, then the mail message is transformed using the XSLT file called "ArchiveSubscriptionNotification.xsl" placed in the directory specified in the EDDS property EDDS_MISSION_XSL_DIRECTORY.

- esa.egos.edds.farc.subscription.expiredrequests

  Responds to notifications that a FARC Subscription request has expired. The EDDS Server sets the relevant request to SERVER_COMPLETED and removes the subscription information from the database.

- esa.egos.edds.request.batch.resume.tobeprocessed

  Responds to requests that have been pre-processed by the EDDS Archiver to resume an active or queued batch request. When the message is received, a check is made to see if the request is suspended or in one of the resumable error states. If so, the request is restarted so, that it starts processing data that would have come after the last file that was completed when the request failed or was suspended. A minor overlap of data might occur. For raw requests and the OOL report the overlap can be larger. This is because the packets are not decoded, and the PARC provides the packets in batches, so the exact time stamps are not known.

The EDDS Server responds to messages sent to the following topics:

- esa.egos.edds.request.batch.cancel.tobeprocessed

  Responds to requests that have been pre-processed by the EDDS Archiver to cancel an active or queued batch request. This needs to be a topic, as it is not known which EDDS Server could be processing the batch request, so each EDDS Server needs to receive the message. If it were a queue, only one of the EDDS Servers would receive the message. When the message is received, a check is made to see if it is currently being processed, and if so, the request is stopped and set to CANCELED.

- esa.egos.edds.request.batch.suspend.tobeprocessed

  Responds to requests that have been pre-processed by the EDDS Archiver to suspend an active batch request. This needs to be a topic, as it is not known which EDDS Server could be processing the batch request, so each EDDS Server needs to receive the message. If it were a queue, only one of the EDDS Servers would receive the message. When the message is received, a check is made to see if it is currently being processed, and if so, the request is stopped and set to SUSPENDED. Before stopping, a reference of the last sample of the last finished file is marked into the acknowledgement. This gives EDDS server the ability to resume the request later on.

- esa.egos.edds.request.stream.cancel.tobeprocessed

  Responds to requests that have been pre-processed by the EDDS Archiver to cancel an active or queued stream request. This needs to be a topic, as it is not known which EDDS Server could be processing the stream request, so each EDDS Server needs to receive the message. If it were a queue, only one of the EDDS Servers would receive the message. When the message is received, a check is made to see if it is currently being processed, and if so, the request is stopped and set to CANCELED.

- FARC Notifications topic (specified in properties file)

Responds to notifications from the FARC that a file has been changed in the FARC. The EDDS Server takes the information in the message header and uses it to find active FARC subscriptions that are interested in this file. It then processes the subscription as per the request (either sending an e-mail notification, retrieving the file or ending the subscription).

- FARC Notifications topic (specified in properties file)

  Responds to notifications from the FARC that a file has been changed in the FARC. The EDDS Server takes the information in the message header and uses it to construct a `FarcCatRec` to notify the client, via the web server, that a change has been made to the FARC. The header property MISSION_NAME is set with the name of the mission the notification was processed on. The message is sent to the queue “esa.egos.edds.updates.farc”.

6.2.8.8.1 Server Initialization

The EDDS Server is a standalone Java application. The process of initialisation consists on the creation of:

- EDDS Request Manager;
- EDDS Acknowledgement Manager;
- EDDS Request Handler;
- EDDS Report Data Processor;
- EDDS Database Manager.

After the complete initialisation of the EDDS data structure the Request Manager waits to process the incoming EDDS batch and stream requests.

6.2.8.8.2 Request Processing

The EDDS server has the main functionality to support the execution of any EDDS request (Batch and Stream). The requests are received from the message broker by the Request Manager. It should be noted that the EDDS Server establishes the connection to the message broker itself, not the other way around. Once the connection is established, messages are sent to EDDS Server over the connection. As the messages are stored persistently, any missed messages are sent when the EDDS Server is restarted. That means that the requests are still going to be processed even if the EDDS server was not running or when the EDDS server has crashed before acknowledging the messages. In addition the server will recover requests which are in state SUBMITTED with execution time in the past. Time interval for this control is present among the configurable parameters of the server.

The batch requests are then passed to the Request Handler by a Camel route, which initiates their processing. In order to process the request the chain of responsibility pattern has been adopted. Each request has to pass the following steps:

- Authorisation: It is verified that the user performing the request has sufficient privileges to perform the request. At the end of the authorisation step, an update of the Acknowledgement file is produced.
- Quota Check: All the generic quota checks are made here and if any of them fail, the request will not be executed, acknowledgement with the explanation of failed check is produced.
- Execution: before requests are executed, they are queued according to the priority assigned to each user. This guarantees that only a limited number (configurable) of requests are processed at the same time, and the requests with higher priority are executed first. The request execution is delegated to the data provider. During the execution of the request, the data provider notifies the EDDS server with the execution status, which is stored for the elaboration of the EDDS statistics. During the execution Acknowledgement files are updated to allow the client to check the actual execution status of the request.

For stream requests, the requests are passed to the Stream Request Handler by a Camel route, which initiates their processing. The following steps are performed:

- Execution: before requests are executed, they are queued according to the priority assigned to each user. This guarantees that only a limited number (configurable) of requests are processed at the same time, and the requests with higher priority are executed first. The request execution is delegated to the stream provider. The stream provider takes the streamed data from the archive and makes it available on a topic that the web server is listening to, so that the streamed data can be passed to the client using CometD.

6.2.8.8.3 Acknowledgement Processing

During the processing of a request, the EDDS server produces updates for each request acknowledgement messages. Updating the acknowledgment is centralised and is done by the EDDS Archiver.

6.2.8.8.4 Logger and Configuration Processing

At server start-up, the Configuration Manager initialised by the Spring Framework. This allows the EDDS server and any of the other components to read the relevant configuration data. Logging is performed using the Log4j library, and the level of logging can be configured at runtime by editing the log4j configuration file found in the same directory as the EDDS Server runtime installation. The log messages are also logged to a database, via the EDDS Archiver, to allow the client application to retrieve historical log messages, should the user have permission to do so.

6.2.8.8.5 Interface with the Data Providers

Each request passes through the specific BatchRequestHandlerFactory that generates and combines all the handlers associated with the processing of this specific request. Each archive has specialized handlers for retrieving data, filtering and formatting the results to an output file.

6.2.8.8.6 Data

The request is processed by the EDDS request processor and a result file is generated. The result is subject to formatting, compression and encryption depending on the request and then moved into the INTRAY directory of the SFT component. The SFT component will take care of the transport of the result file over the firewall to the Delivery Manager component.

6.2.9 PARC and Data Provision data providers

6.2.9.1 Purpose

The PARC data provider retrieves the data from the Packet Archive from the user's request, formats the data and performs optional functions such as compressing and encrypting the data. The code is the same for the PARC Manager and Data Provision Service. The request contains information about the data source, allowing EDDS to fetch the data using the correct API.

6.2.9.2 Function

The main functions of the PARC data provider are:

- Processing the specific PARC service requests (Batch, Stream) which are retrieved by the EDDS server;
- Performing the PARC service request execution via the interface made available from the PARC;
- Filtering the retrieved packet data;
- Formatting the retrieved packet data;
- Transforming the XML files;
- Compressing and Encrypting the retrieved packet data;
- Updating the acknowledgement for the request with the status of the request.

6.2.9.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS PARC Request Processor chain (Data Manager, Filter, Formatter, Encryption, Data Compress);

6.2.9.4 Dependencies

This component depends on the PARC Java API and the Data Provision Services Java APIs

6.2.9.5 Interfaces

6.2.9.5.1 PARC data provider

In order to satisfy the performance requirements, and to minimise the dependency of the EDDS code from the S2K libraries, the data is retrieved from the PARC using the CORBA interface provided by the PARC Manager.

6.2.9.5.1.1 PARC CORBA Interface Current Limitations

The current version of the PARC Manager interface imposes the following limitations:

- 1) It is possible to perform queries based only on SPID and not on APID.
- 2) It is not possible to define filtering criteria (e.g. type and sub type)

Note: EDDS extends the filtering options provided by the PARC API with other fields (e.g. APID, TYPE, and SUBTYPE) which improves dramatically its retrieval capability. The retrieval by APID is achieved by loading the S2K Packet definition table and performing a mapping between the APID to the corresponding SPID.

6.2.9.5.2 Data Provision Services data provider

The data is retrieved from the PARC via the Data Provision Services using the CORBA interface provided by the Data Provision Services. This interface provides the data from the archive already

decoded, helping to decouple EDDS from the low level packet details. In addition, parameter information can be obtained for Telemetry packets using this interface.

6.2.9.6 Resources

Disk Space: the retrieved data after the operation of filtering and formatting are stored on the local file system.

6.2.9.7 References

N/A

6.2.9.8 Processing

6.2.9.8.1 Batch Request Processing

The service allows the retrieval of Telemetry, Commanding and Event Packets from the PARC. The execution of the services performs the following actions:

1) Instantiate the PARC data processor: The components of the data processor, i.e. the data provider handler, the filter handler, the formatter handler, the data compression handler and the data encryption handler are instantiated according to the content of the data request;
2) The batch request is executed, A PARC Batch request is executed as an activity, and this implies that each request is executed in a separated thread;
3) The service provides feedback on the completion status of the batch request which allows monitoring of the used resources;
4) At any time using the request execution can be aborted.

6.2.9.8.2 Stream Request Processing

Streaming packet data is only supported by the Data Provision Services. The service allows the streaming of live Telemetry, Commanding, Event Packets and out of limits data. The execution of the services performs the following actions:

1) Instantiate the Data Provision streaming processor: The components of the data processor, i.e. the data provider handler, the filter handler and the formatter handler are instantiated according to the content of the data request;
2) The stream request is executed, A stream request is executed as an activity, and this implies that each request is executed in a separated thread;
3) The service provides feedback on the status of the stream request which allows monitoring of the used resources;
4) At any time using the request execution can be aborted.
5) Clients can start the data stream by using the client API to start the stream of data from the ActiveMQ message bus to the CometD stream.

6.2.9.9 Data

N/A

6.2.10 EDDS FARC Interface

6.2.10.1 Purpose

The FARC interface Jar file contains an interface layer to the actual FARC implementation for EDDS. The implementation of these interfaces is provided in a separate Jar file which is included in the classpath at runtime. This enables the EDDS Server to be de-coupled from the actual FARC version being used. To change the supported version of the FARC being used, the correct EDDS FARC interface implementation is deployed along with the relevant FARC COTS.

6.2.11 FARC data provider

6.2.11.1 Purpose

The FARC data provider retrieves the data from the File Archive from the user's request, formats the data (only for the catalogue) and performs optional functions such as compressing and encrypting the data.

6.2.11.2 Function

The main functions of the FARC data provider are:

- Processing the specific FARC service requests (Batch) which are retrieved by the EDDS Server;
- Perform the FARC service request execution via the interface made available from the FARC;
- Formatting the retrieved file/catalogue data;
- Compressing and encrypting the retrieved file/catalogue data;
- Notifying the client with the execution status of each request.

6.2.11.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS FARC Interface

6.2.11.4 Dependencies

This component depends on the FARC Java API and the EDDS FARC interface implementation.

6.2.11.5 Interfaces

6.2.11.5.1 FARC data provider

The data is retrieved from the FARC using the Java APIs interface provided by the FARC.

6.2.11.6 Resources

Disk Space: the retrieved data after the operation of filtering and formatting are stored on the local file system.

6.2.11.7 References

N/A

6.2.11.8 Processing

6.2.11.8.1 Batch Request Processing

The service allows the retrieval of Files and Catalogues from the FARC and for subscribing to updates from the FARC (subscription). The execution of the services performs the following actions:

1) Instantiate the FARC data processor: The components of the data processor, i.e. the data provider handler, the filter handler, the formatter handler, the data compression handler and the data encryption handler are instantiated according to the content of the data request;
2) The batch request is executed, A FARC Batch request is executed as an activity, and this implies that each request is executed in a separated thread;
3) The service provides feedback on the completion status of the batch request which allows the monitoring of the used resources;
4) At any time using the request execution can be aborted.

For FARC Subscription requests, the execution proceeds as follows:

1) Call the FARC Subscription Manager with the FARC Subscription request. This will add an entry into the farc_subscription table within the database.
2) When a change occurs in the FARC, the FARC Server sends a message to a topic on a message bus that the EDDS Server is listening to.
3) A Camel route then passes the received message to the FARC Subscription Manager which then looks for matching subscription requests in the farc_subscription database.
4) The EDDS Server then submits an ArchiveFile batch request for the changed data or sends an e-mail notification, depending on what has been requested in the subscription request.
5) If the subscription request is cancelled, or if the subscription request is for a specific file in the FARC, and this file is deleted, the subscription is removed from the farc_subscription table in the database.

6.2.11.8.2 Stream Request Processing

Streaming does not make sense for the FARC, as it does not process streams of data, so streaming is not supported for FARC requests.

6.2.11.9 Data

N/A

6.2.12 File System data provider

6.2.12.1 Purpose

The File System data provider retrieves the data from the EDDS File System Index from the user's request, formats the data and performs optional functions such as compressing and encrypting the data.

6.2.12.2 Function

The main functions of the File System data provider are:

- Processing the specific File System service requests (Batch) which are retrieved by the EDDS Server;
- Perform the File System service request execution via the interface made available by the File System Index;
- Formatting the retrieved file/catalogue data;
- Compressing and encrypting the retrieved file/catalogue data;
- Notifying the client with the execution status of each request.

6.2.12.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS File System Index

6.2.12.4 Dependencies

This component depends on the EDDS File System Index implementation.

6.2.12.5 Interfaces

6.2.12.5.1 Index File Helper

The data is retrieved from the EDDS File System Index using the Index File Helper interface.

6.2.12.6 Resources

Disk Space: the retrieved data after the operation of filtering and formatting are stored on the local file system.

6.2.12.7 References

N/A

6.2.12.8 Processing

6.2.12.8.1 Batch Request Processing

The service allows the retrieval of the File Data and Catalogue listings. The execution of the service performs the following actions:

1) Instantiating the request processor: The components of the request processor, i.e. the data provider, the filter handler, the formatter handler, the data compression handler and the data encryption handler are instantiated according to the content of the data request;
2) The batch request is executed in a separate worker thread;
3) The user requested data is fetched from the File System Index, if additional info is needed, the information is retrieved from the File System Indexed directory on the local hard drive.
4) The data is written to the output file(s) in a requested format.

6.2.12.8.2 Stream Request Processing

Streaming does not make sense for the File System data provider, as it does not process streams of data, so streaming is not supported for File System requests.

6.2.12.9 Data

N/A

6.2.13 DARC data provider

6.2.13.1 Purpose

The DARC data provider retrieves the data from the Telemetry Parameter Archive from the user's request, formats the data and performs optional functions such as compressing and encrypting the data.

6.2.13.2 Function

The main functions of the DARC data provider are:

- Processing the specific DARC service requests (Batch, Stream) which are retrieved by the EDDS server;
- Performing the DARC service request execution via the interface made available from the DARC;
- Filtering the retrieved parameter data;
- Formatting the retrieved parameter data;
- Transforming the XML files;
- Compressing and encrypting the retrieved parameter data;
- Notifying the client with the execution status of each request.

6.2.13.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS DARC Request Processor chain (Data Manager, Filter, Formatter, Encryption, Data Compress);

6.2.13.4 Dependencies

This component depends on the DARC Java API.

6.2.13.5 Interfaces

6.2.13.5.1 DARC data provider

The data is retrieved from the DARC using the Java APIs interface provided by the DARC.

6.2.13.6 Resources

Disk Space: the retrieved data after the operation of filtering and formatting is stored on the local file system.

6.2.13.7 References

N/A

6.2.13.8 Processing

6.2.13.8.1 Batch Request Processing

The service allows the retrieval of the Parameter Data. The execution of the service performs the following actions:

5) Instantiating the DARC data processor: The components of the data processor, i.e. the data provider handler, the filter handler, the formatter handler, the data compression handler and the data encryption handler are instantiated according to the content of the data request;

6) The batch request is executed, A DARC Batch request is executed as an activity, and this implies that each request is executed in a separated thread;
7) The service provides feedback on the completion status of the batch request this allows the monitoring of the used resources;
8) At any time the request execution can be aborted.

6.2.13.8.2 Stream Request Processing

The service allows the retrieval of streamed Parameter Data. The execution of the service performs the following actions:

1) Creation of a new Camel route. The route takes messages from the DARC ActiveMQ broker based on the filter the user has provided, converts them for EDDS use (i.e. the payload becomes an encoded parameter record encoded using Google Protocol Buffers) and sends them to the parameter stream topic on the EDDS ActiveMQ broker.
2) At any time the Camel route can be aborted.

6.2.13.9 Data

N/A

6.2.14 SMON data provider

6.2.14.1 Purpose

The SMON data provider retrieves data from SCOS Monitoring service, formats the data and performs optional functions such as compressing and encrypting the data.

6.2.14.2 Function

The main functions of the SMON data provider are:

- Processing the specific SMON service requests which are retrieved by the EDDS server;
- Starting up (and restarting) the dedicated EDDS SMON service through S2K TKCOM interface;
- Performing the SMON service request execution via the SMON CORBA interface;
- Filtering the retrieved parameter data;
- Formatting the retrieved parameter data;
- Transforming the XML files;
- Compressing and encrypting the retrieved parameter data;
- Notifying the caller of the data provider with the execution status of each request

6.2.14.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS SMON Request Processor chain (Data Provider, Filter, Formatter, Encryption, Data Compress);

6.2.14.4 Dependencies

This component depends on the EUD4S2K provided jar for communicating with the CORBA services.

6.2.14.5 Interfaces

6.2.14.5.1 SMON data provider

The data is retrieved from the SMON using the Java APIs interface provided by the SMON.

6.2.14.5.2 TKMA interface

TKMA Java APIs are used to (re)start the SMON instance if necessary.

6.2.14.6 Resources

Disk Space: the retrieved data after the operation of filtering and formatting is stored on the local file system.

6.2.14.7 Processing

The SMONParameterProvider retrieves the Parameter samples through the SMON Java interface. In detail, the implementation of the request method performs the following operations:

1. Checks whether the configured SMON server is running. If it's not, starts it;
2. Gets all parameter definitions from SMON;
3. Checks whether the parameter names specified in the filter pass the quota restrictions set for the current user;
4. Subscribes to actual parameters matching the parameter filter specified in the request;
5. Checks the validity of the specified time window against quota restrictions;
6. Requests the parameter samples within the specified time frame and passes the data to the filter handler which passes it to the formatter handler to write the data to disk;

6.2.15 Delivery Manager

6.2.15.1 Type

The component is implemented as a standalone Java application.

6.2.15.2 Purpose

The Delivery Manager picks up the response files created by the EDDS Server and delivers them based on the user's request.

6.2.15.3 Function

The main function of the EDDS Delivery Manager is the delivery of response files to the user: it monitors the EDDS Server completion directory for any created batch request responses and manages the delivery of the file. The EDDS Delivery Manager also sends the acknowledgement e-mails to the users when requested. It also handles the processing of DeleteData and DeleteRequest requests, and deletes stored responses when they expire.
If EDDS Server emits more than one response file per request (e.g. when splitting the response data into several response files, adding the key file for encryption, etc.), then these files can be delivered independently of each other since the status of each of the result files is kept in the acknowledgement of the request.

6.2.15.4 Subordinates

The subordinates of this component are:

- Response File Poller (polls the directory where response files to be delivered are placed for new files to be processed);
- Response File List (a simple list of files that need to be processed);

- EDDS Acknowledgement Manager.

6.2.15.5 Dependencies

This component depends upon:

- The (S)FTP protocol.

6.2.15.6 Interfaces

The interfaces of this component are:

- esa.egos.edds.delivery.impl.AbstractDelivery
- esa.egos.edds.delivery.DeliveryManager
- esa.egos.edds.delivery.FileSystemMonitor

6.2.15.7 Resources

The following resources are required by this component:

- Disk space for storage of response data that is to be sent to the client in a 'Batch Response' Directory.

6.2.15.8 References

N/A

6.2.15.9 Processing

The Delivery Manager responds to messages sent on the following queues:

- esa.egos.edds.delivery.responsedeletion.scheduled.tobeprocessed

  The Delivery Manager itself sends messages on this queue for the scheduled deletion of response files. The message is sent with a delay as specified in the Delivery Manager properties file. When the message is finally delivered to the Delivery Manager, it starts the process of deleting the response file(s) and updating the user's disk space usage.

- esa.egos.edds.request.batch.deletedata.tobeprocessed

  Responds to requests pre-processed by the EDDS Archiver for manual deletion of response data stored on the server. The Delivery Manager will delete the response file(s), update the user's disk space usage and remove the scheduled deletion message from the queue as it is no longer needed.

- esa.egos.edds.request.batch.cancel.deletedata

  Responds to requests to delete the response files of requests that have been cancelled. The payload of the message is a HashMap<String, Long> mapping the files to be deleted (including path) to the size of the data (in bytes). The header properties expected are:

  RequestId – the ID of the request (String)

  RoleName – the role used for the request (can be null)

  UserName – the name of the user who submitted the request

  State – the current state of the request (String)

- esa.egos.edds.request.batch.deleterequest.scheduled

  Responds to messages sent by the EDDS Archiver to delete the files related to a request when the database clean-up job is fired, and the request itself. The payload of the message is a HashMap<String, Long> mapping the files to be deleted (including path) to the size of the data (in bytes). The header properties expected are:

TargetRequestId – the ID of the request (String)

RoleName – the role used for the request (can be null)

UserName – the name of the user who submitted the request

State – the current state of the request (String)

- esa.egos.edds.request.batch.deleterequest.tobeprocessed

  Responds to requests pre-processed by the EDDS Archiver for manual deletion of request entries in the database. The Delivery Manager will send a request to the EDDS Archiver to have the request specified removed from the database, remove any response file(s) associated with the request, update the user's disk space usage and remove the scheduled deletion message from the queue as it is no longer needed.

- esa.egos.edds.ftp.createprivateuserlocation

  Responds to requests to create a new user's private home folder. It accepts a single header property – UserName – the name of the new user.

- esa.egos.edds.ftp.removeprivateuserlocation

  Responds to requests to remove a user's private home folder. It accepts a single header property – UserName – the name of the user.

- esa.egos.edds.ftp.addusermissionlink

  Responds to requests to add the mission symbolic link to the user's home directory. It expects two header properties – UserName – the name of the user and MISSION_NAME – the name of the mission to create the link to.

- esa.egos.edds.ftp.removeusermissionlink

  Responds to requests to remove the mission symbolic link from the user's home directory. It expects two header properties – UserName – the name of the user and MISSION_NAME – the name of the mission to remove the link from.

- esa.egos.edds.ftp.adduserrolelink

  Responds to requests to create the symbolic link for the specified role for the specified user. It expects three header properties – UserName – the name of the user, RoleName – the name of the role and MISSION_NAME – the name of the mission.

- esa.egos.edds.ftp.removeuserrolelink

  Responds to requests to remove the symbolic link for the specified role for the specified user. It expects three header properties – UserName – the name of the user, RoleName – the name of the role and MISSION_NAME – the name of the mission.

- esa.egos.edds.ftp.createmissiondirectory

  Responds to requests to create the new mission directory, plus all the directories for each domain. It expects no header properties and a payload of a `esa.egos.edds.ldap.dto.MissionDto` object.

- esa.egos.edds.ftp.updatemissiondirectory

  Responds to requests to update the domain directories for the specified mission. It expects no header properties and a payload of a esa.egos.edds.ldap.dto.MissionDto object.

- esa.egos.edds.ftp.removemissiondirectory

  Responds to requests to remove the mission directory, plus all the directories for each domain. It expects no header properties and a payload of a esa.egos.edds.ldap.dto.MissionDto object.

- esa.egos.edds.ftp.createmissionrolelocation

  Responds to requests to create the role directory for the specified mission. It expects two header properties – RoleName – the name of the role to create the directory for, and MISSION_NAME – the name of the mission the role is for.

- esa.egos.edds.ftp.removemissionrolelocation

  Responds to requests to remove the role directory for the specified mission. It expects two header properties – RoleName – the name of the role to remove the directory for, and MISSION_NAME – the name of the mission the role is for.
- esa.egos.edds.email.queue

  The e-mail messages queue. Expects to find messages containing a `MailMessage` object. Messages placed on this queue will actually be sent to the mail server for delivery.

- esa.egos.edds.request.batch.cancel.deletedata

  Responds to requests to delete any delivered response files if a request has been cancelled. Expects a single String header property with a key of "RequestId" containing the ID of the request to delete the response files for.

Files that are processed by EDDS Server are uploaded by the SFT component in a configurable directory. The Response File Poller in Delivery Manager checks the directory periodically looking for new files. Any new file will be added to the Response File List. As soon as a new file is added, the Delivery Manager is notified by the Quartz Job Scheduler and a new Delivery is created and performed. The request ID is obtained from the filename and this is used to lookup the request in the database, in order to determine the delivery method.

Each new Delivery will be initialised with the information provided during the batch request in the acknowledgement part. For any Batch Request a related acknowledgement file is stored in the file system and is used by the Delivery Manager to initiate a new Delivery process. Being related to a specific batch request, acknowledgement files are removed as soon as the batch request response is removed from the file system.

The delivery mechanism provides three different delivery modes:

- Mail acknowledgement;
- File Server "push";
- Server Delivery (file is simply stored on EDDS local FTP server).

6.2.15.9.1 Mail acknowledgement

The acknowledgement file contains a mail recipient. The Delivery Manager notifies the Client about the completion of a batch request via email. The Client will, in a separate operation, retrieve the file directly through the EDDS Web Server (see EUICD [AD-03] BatchRequestService).

6.2.15.9.2 File Server - push file

The acknowledgement file contains the location of a remote file server (hostname and port). The Delivery Manager creates a new Delivery thread process and executes it. The delivery process tries to establish a connection with the remote FTP or SFTP file server and uses the "PUT" method to upload the file. In case of an error, the Delivery will try to send the file again (for a number of times, with configurable intervals between attempts as described in the mission's configuration file), after which the Delivery will be aborted.

The delivery manager attempts to deliver the file to the remote (S)FTP server. If the user selects 'Keep file after delivery' or the remote delivery fails, then the file will be stored on the local FTP for the maximum time allowed.

6.2.15.9.3 Client - download response file

The Client checks the status of its requests and retrieves the file directly through the EDDS Web Server (see EUICD [AD-03] BatchRequestService).

The whole request doesn't have to be completed for the response files to already be delivered to client.

The files can be delivered to the client as long as these are stored on the local EDDS file server after processing has been completed by the Delivery Manager. The mission administrator can specify the length of time a file is kept, once this time has elapsed, the file will be automatically deleted.

6.2.15.9.4 File Deletion

When a request is successfully completed and when the retrieved file is stored in a local directory on the EDDS server. To ensure that redundant data does not fill this directory, files are automatically deleted after a configurable amount of time. Mission administrators set this time period, within the mission's configuration file.

Before transferring the file to a remote (S)FTP, the file is stored on the EDDS. By default the file will be deleted from the EDDS server once the transfer has been successfully completed. The user has the option override this default setting when issuing the request. The file will be stored for the maximum amount of time allowed by the mission administrator.

Files are scheduled for deletion by sending a scheduled JMS message to a file deletion task queue. When the scheduled time arrives, the deletion task message is picked up by Delivery Manager and processed. Or if the Delivery Manager is not running then they are picked up whenever it is started again.

The file deletions are not rescheduled when the property in the mission specific configuration is changed. The new value is only applied to new files being delivered after the property change (and restart of the component).

The data can also be deleted from the server by issuing a DeleteData or DeleteRequest request specifying the request id for which response data should be deleted. After the response data is deleted from the server the status of the request changes to DELIVERY_RESP_DELETED.

6.2.15.9.5 Reattempting delivery after a system failure

Any files that fail to be delivered will be sent to Delivery Manager's failed directory. To reattempt delivery of these files, they have to be manually moved back to the Delivery Manager's inbox.

The delivery manager can also be stopped during the delivery process of a request, without loss of data. Upon restarting, a check is performed for any incomplete deliveries by checking the working directory for existing files. These requests will be resubmitted for delivery before new deliveries are processed. Note that the retrieval is not performed again, only the re-submission of the saved response data.

Users are advised not to touch the contents of the working directory.

6.2.15.10 Data

N/A

6.3 Software Product Sub Components

This section describes the EDDS software sub components for the components defined in the section 6.2:

6.3.1 Configuration Manager

6.3.1.1 Type

This component is a Java package (esa.egos.edds.configuration).

6.3.1.2 Purpose

The Configuration Manager exposes the configuration information loaded by Spring to non-Spring managed components of EDDS.

6.3.1.3 Function

It provides all of the EDDS server components with configuration information. The component provides a façade to the actual configuration data provider. If a configuration data is not found an exception is raised. It also supports the use of mission specific configuration information.

The implementation of the façade allows decoupling the implementation of the configuration interface from the EDDS server.

6.3.1.4 Subordinates

All of the EDDS server components use the Configuration Manager to read their specific configuration settings:

- EDDS Web Server
- EDDS Server
- EDDS Delivery Manager
- EDDS Archiver

6.3.1.5 Dependencies

Apache Log4J

6.3.1.6 Interfaces

The component provides:

- A configuration interface, which expose the methods to retrieve the EDDS configuration data;
- A configuration manager singleton, which provides the implementation of the configuration interface;

6.3.1.7 Resources

N/A

6.3.1.8 References

N/A

6.3.1.9 Processing

At start up of the EDDS server components, the actual configuration implementation is instantiated and passed to the configuration manager, which stores the configuration so that it is available for the EDDS server component. For speed, the configuration is not re-read every time a configuration value is required. Instead, the data is stored in memory. After changing a configuration variable, the component must be restarted so that the configuration is re-read.

If the configuration interface implementation is not instantiated, a specific exception is raised.

6.3.1.10 Data

N/A

6.3.2 Acknowledgement Manager

6.3.2.1 Type

The component is implemented as a Java Package (esa.egos.edds.acknowledgement).

6.3.2.2 Purpose

The Acknowledgement Manager creates acknowledgement data stored with the user's request in the database and prepares the notification email messages if needed.

6.3.2.3 Function

The purpose of the Acknowledgement Manager is to receive the execution statuses of any EDDS request processed by the server. It also puts the acknowledgements to be delivered as email messages to the message queue.

6.3.2.4 Subordinates

There are no subordinates of this component at the architectural level.

6.3.2.5 Dependencies

N/A

6.3.2.6 Interfaces

N/A

6.3.2.7 Resources

N/A

6.3.2.8 References

N/A

6.3.2.9 Processing

The Acknowledgement Manager is a utility package that is used by other components in order to:

- create a new acknowledgement (e.g. every time a request is received by the Web Server)
- notify a request change of status to the user (e.g. when a request is delivered)

All these functionalities are dependant from the actual request data, so during the processing of a request a business component will call the Acknowledgement Manager passing some parameter specific to the current request; the Acknowledgement Manager will manage the generation of the acknowledgement or the notification of the acknowledgement to the user, hiding all the complexity to the business component.

6.3.2.10 Data

N/A

6.3.3 EDDS Batch Request Manager

6.3.3.1 Type

The component is implemented as a Java Package (esa.egos.edds.server.request).

6.3.3.2 Purpose

The EDDS Batch Request Manager is a part of the EDDS Server and processes batch requests.

6.3.3.3 Function

The purpose of the EDDS Batch Request Manager is to receive the EDDS Batch Requests and initiate their execution and to check for previously uncompleted Batch requests during initialisation.

6.3.3.4 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS Batch Request Handler.

6.3.3.5 Dependencies

N/A

6.3.3.6 Interfaces

The component exposes an interface, which can be used to submit requests to the corresponding handlers for their execution.

6.3.3.7 Resources

N/A

6.3.3.8 References

N/A

6.3.3.9 Recovering

During the initialisation of the Request Manager all the Batch requests that are in an incompatible state (in status ACTIVE or QUEUED) because of the EDDS server crash are being restarted. If the request used a splittable format, the already completed parts are reserved and the request continues from where it left off, as if the request had been suspended and resumed. Other requests are to be processed from the start as if these were new requests.

Also the working directory of EDDS server is going to be emptied of all the half-completed files.

6.3.3.10 Processing

The EDDS Batch Request Manager has the main functionality to support the execution of any EDDS batch request. The EDDS Batch Request Manager receives the requests, which are passed to it from the message bus.

The requests are translated into Bean classes. Such Bean classes are automatically generated using JAXB from the schema files, which describe the EDDS requests. This operation allows validating the service request according to the schema and refusing their execution in case of validation errors.

The request processing is then delegated to the EDDS Batch Request Handler.

6.3.3.11 Data

N/A

6.3.4 EDDS Stream Request Manager

6.3.4.1 Type

The component is implemented as a Java Package (esa.egos.edds.server.stream).

6.3.4.2 Purpose

The EDDS Stream Request Manager is a part of the EDDS Server and processes stream requests.

6.3.4.3 Function

The purpose of the EDDS Stream Request Manager is to receive the EDDS Stream Requests and initiate their execution and to check for previously uncompleted Stream requests during initialisation.

6.3.4.4 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.4.5 Dependencies

N/A

6.3.4.6 Interfaces

The component exposes an interface, which can be used to submit requests to the corresponding handlers for their execution.

6.3.4.7 Resources

N/A

6.3.4.8 References

N/A

6.3.4.9 Recovering

During the initialisation of the Request Manager all the stream requests that are in an incompatible state (in status ACTIVE or QUEUED) because of the EDDS server crash are being restarted. Other requests are to be processed from the start as if these were new requests.

6.3.4.10 Processing

The EDDS Stream Request Manager has the main functionality to support the execution of any EDDS stream request. The EDDS Stream Request Manager receives the requests, which are passed to it from the message bus.

The requests are translated into Bean classes. Such Bean classes are automatically generated using JAXB from the schema files, which describe the EDDS requests. This operation allows validating the service request according to the schema and refusing their execution in case of validation errors.

The request processing is then delegated to the relevant stream processor

6.3.4.11 Data

N/A

6.3.5 EDDS Batch Request Handler

6.3.5.1 Type

The component is implemented as a Java Package (esa.egos.edds.server.batch).

6.3.5.2 Purpose

The EDDS Batch Request Handler is part of the EDDS Server and receives requests from the Request Handler and performs checks on the requests before executing them.

6.3.5.3 Function

The main function of the EDDS Batch Request Handler is to authorise and initiate the execution of the EDDS service requests.

6.3.5.4 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS Request Authorization Handler;
- EDDS Request Quota Check Handler;
- EDDS Batch Request Executor.

6.3.5.5 Dependencies

- Quartz Library;
- EDDS Model classes generated from the EDDS Web services schema files;

6.3.5.6 Interfaces

N/A

6.3.5.7 Resources

N/A

6.3.5.8 References

N/A

6.3.5.9 Processing

The EDDS Batch Request Handler implements the chain of responsibility pattern. Each request has to be authorised and checked against quota levels in order to be executed.

The chain of responsibility pattern decouples the sender of the request to the receiver. The only link between the sender and the receiver is the request that is sent. Based on the request data sent, the receiver is triggered. This approach is called “data-driven”: in most of the behavioural patterns, the data-driven concepts used have a loose coupling. The responsibility of handling the request data is given to any of the members of the “chain”. In the case of the EDDS Request Handler, the chain foresees three possible steps:

- The Authorisation step: The request is validated against the privileges possessed by the user performing the request. If the user has the necessary privileges, the request is sent to the next element of the chain, otherwise it is stopped;
- The Quota Check step: The request is validated against the user role quotas. If any of the generic quota limits is reached, the request processing is stopped.
- The Execution Step: The request is sent to the corresponding data provider for execution.

The Data provider identifies the processor, which executes the request.

At the end of each step of the chain, an update of the Acknowledgement message is produced.

6.3.5.10 Data

N/A

6.3.6 EDDS Quota Check Handler

6.3.6.1 Type

The component is implemented as a Java Class.

6.3.6.2 Purpose

The EDDS Quota Check Handler checks if the user's request can be processed within the generic quota restrictions imposed on the user.

6.3.6.3 Function

The main function of the EDDS Quota Check Handler is to check the generic quota limits.

6.3.6.4 Subordinates

None

6.3.6.5 Dependencies

- Acknowledgement Manager

6.3.6.6 Interfaces

N/A

6.3.6.7 Resources

N/A

6.3.6.8 References

N/A

6.3.6.9 Processing

The handler checks for 4 types of quota limits. The limits are defined by quota sets and these are assigned to user roles. If the user of the request is the EDDS administrator or a mission administrator, then the quota check will always be waived.

The types of limits are:

- Number of requests per period – user cannot issue more than limited number of requests per time period.
- Number of ongoing requests – user cannot have more than limited number of active or queued requests processed at once.
- Amount of data per period – user cannot issue any more requests in current time period after the amount of allowed data per period is reached. The amount of data is added towards the quota after the response file is produced by the EDDS Server and saved locally.
- Amount of disk space used in EDDS server – user cannot issue requests after the amount of disk space used on EDDS server exceeds the limit.

Only the request independent quota checks are done in this component. The quotas that are request-specific are checked in their respective processors.

- For PARC see: 6.3.9.9.2.2 (PktTm), 6.3.9.9.3.2 (PktTmRaw), 6.3.9.9.6.2 (PktTmStatistics)
- For FARC see: 6.3.10.9.1.2 (Catalogue), 6.3.10.9.2.2 (File), 6.3.10.9.3.2 (Subscription)
- For DARC see: 6.3.12.8.5.2 (Param), 6.3.12.8.6.2 (ParamStatistics), 6.3.12.8.7.2 (ParamDefinition), 6.3.12.8.8.2 (ParamPreview)

- For SMON see: 6.2.14.7 (Param)

6.3.6.10 Data

The acknowledgement for the request is updated. If any of the checks fail, the reason will be stated in the error message of the acknowledgement.

6.3.7 Generic Stream Request Processor

6.3.7.1 Type

The component is implemented as a Java Package.

6.3.7.2 Function

The main function of the Stream Request Processor is to provide a common infrastructure used to create the different Stream Request Processors needed to execute the EDDS Stream Requests.

6.3.7.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.7.4 Dependencies

None

6.3.7.5 Interfaces

Each stream request processor must implement the StreamRequestThreadInterface as shown in the figure below:

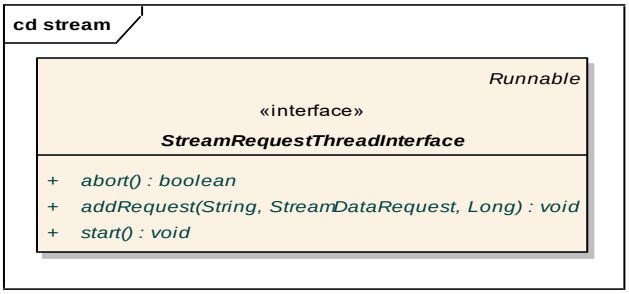

Figure 18 Stream Request Interface

The Stream Request Handler is similar to the Batch Request Execution Manager in that it executes requests for stream data and will create a new stream request processor implementing this interface for each new stream request. The constructor will pass the reference to the Stream Request Handler, so that the processor can inform the handler when it is complete, and the Camel Context to use for creating any required Camel route for processing the stream request. After the handler has created the processor, it will call the addRequest method to pass the details of the stream request that needs processing, and when the request should be stopped. Finally, the handler will call the start method to start the processor's thread.

6.3.8 Generic Batch Request Handler Factory (Data Manager, Filter, Formatter, XML Transformer, Encryption, Data Compress)

6.3.8.1 Type

The component is implemented as a Java Class.

6.3.8.2 Function

The main function of the Request Processor is to provide a common infrastructure used to create the different Request Handlers needed to execute the EDDS Services Requests.

The request processor is based on of the chain of responsibility pattern See Appendix A.

6.3.8.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.8.4 Dependencies

None

6.3.8.5 Interfaces

Each element of the chain (handler) implements the DataHandlerInterface as shown in the figure below:

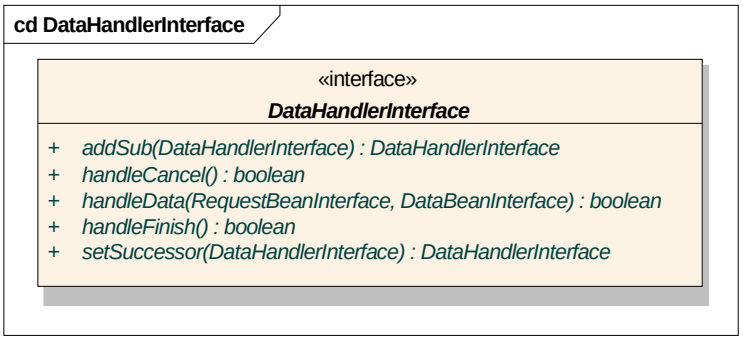

Figure 19 Data Handler Interface

The DataHandlerInteface relies on other two interfaces RequestBeanInterface and DataBeanInterface as reported in the figure below:

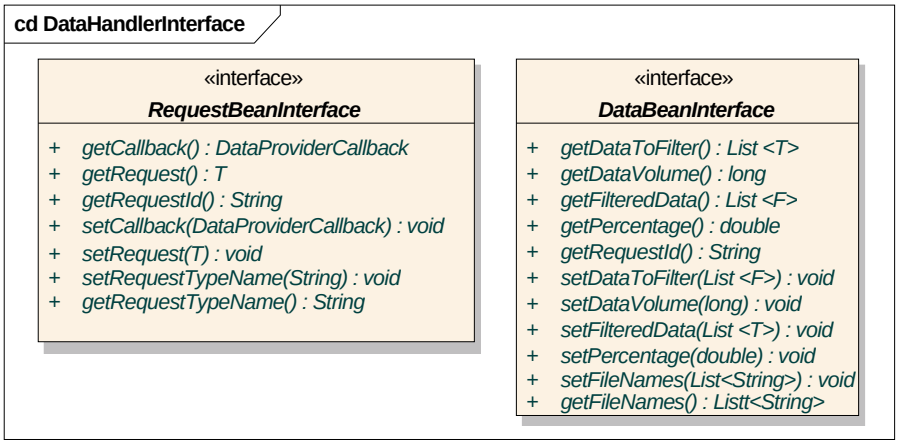

Figure 20 Request and Data Bean Interfaces

The RequestBeanInterface allows accessing to the original request and the notifications callback object.

The DataBeanInterface allows the handler access to the retrieved data and set the request statistics such as the execution percentage and the retrieved data volume.

6.3.8.5.1 Filter Interface

The Generic request processor provides a Filter interface, which shall be implemented for the filtering of the retrieved data. The interface provides a single method called performFilter which takes the record to be filtered and returns a Boolean value. True means the record fulfils the filter options while false means it does not fulfil at least one of the filter options.

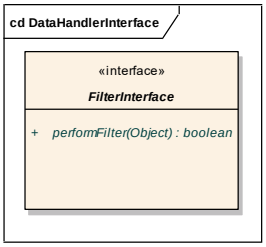

Figure 21 Filter Interface

6.3.8.5.2 Formatter Interface

The Generic request processor provides a Formatter interface, which shall be implemented by the different type of formatters. The most important method of the interface is the format one. It takes as input the record to be formatted and returns an object of type StringBuffer or byte array. The returned object describes the formatted record that should be written into the file, e.g. in case of the raw formatter it will describe the record as a sequence of hexadecimal values. The setCancelled() method may be optionally implemented by some long running formatters.

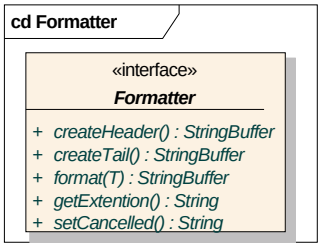

Figure 22 Formatter interface

6.3.8.5.3 FilesProcessor Interface

All the file based operations, after the data has been formatted and saved as a file, are implementing the FileProcessor interface. The available processors are:

- XML Transform processor – provides XSL Transformations for XML files
- AES Ecryption processor
- TAR, TARGZ and ZIP compression providers
- Delivery files processor – moves finished files to the EDDS Server outbox directory

The file processors are called from a FilesProcessorSubHandler in the chain of command processing flow.

6.3.8.5.4 Encryption Processor

EDDS provides a user the possibility to **encrypt** the result of a request. In the request itself the user has to explicitly select this option and this can be done on the MMI in the *Post Process* section of the *Delivery Options* tab.

Only one type of encryption is supported: AES. The AES handler will generate a random one-time key and then take the mission RSA public key and encrypt it. The encrypted key is saved with the response files. Each file is then encrypted with the AES key, and processing is then passed to the next handler.

The RSA location of the public key must be specified using the configuration variable ENCRYPTION_RSA_PUBLIC_KEY_LOCATION and the key must be stored in a secure location that the EDDS Server can access, but only authorised users can access to prevent unauthorised modification of the public key file. The encryption level is 128 bit by default. This can be changed to 192 bit or 256 bit by modifying the configuration variable ENCRYPTION_AES_KEY_STRENGTH. Before increasing the level beyond 128 bit, Oracle's Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy must be installed (see Java ™ Cryptography Architecture (JCA) Reference Guide for Java™ Platform Standard Edition 6).

The AES encryption handler only specifies "AES" as the encryption algorithm. No mode or padding are specified. The default values for AES mode and padding scheme are used. This can be modified by changing the Java security properties file.

It is possible to disabled encryption at a mission level by setting the ENCRYPTION_AES_ENABLED configuration variable to "false". Should a user set AES encryption in their request, the request will be rejected and no files will be delivered.

6.3.8.5.5 Compression Files Processors

EDDS provides a user the possibility to **compress** the result of a request. In the request itself the user has to explicitly select this option and this can be done on the MMI in the *Post Process* section of the *Filter* tab.

When any type of compression is used, the split files are not delivered individually even if it is enabled for the given mission, instead, all the response files are compressed into the same archive.

Three types of compression are supported: TAR, GNU Zip compressed TAR and Zip. TAR by itself is not a compression format, merely a way of storing many files in one single file, complete with the file attributes and directory structure. The compression handler takes each file from the list and adds it to the created archive. The original files are removed after being added to the archive.

The compression level used for Zip can be configured at a mission level by changing the configuration variable COMPRESSION_LEVEL. Valid options are: DEFAULT_COMPRESSION, BEST_COMPRESSION, BEST_SPEED and NO_COMPRESSION.

6.3.8.5.6 EDDSDataProvider and ProviderCallback interfaces

For easier extendablity EDDS has interfaces and generic implementation for the *DataHandlerInterface* that applies to data providers. So the different data providers don't need to handle the chain-of-command pattern, but only implement the data provision logic.

The generic implementation is the *ProviderDataHandler* and it can be used with any data provider that implements the *EDDSDataProvider* interface. The only method that it needs to implement is the *request* method. The data provider's responsibility then is:

- to prepare the filtering options for the back-end service. In this step only the filters that are supported by the back-end are used. Further filtering will be done on EDDS by Filter Data Handler.
- retrieve the data from the back-end service.
- notify the given *ProviderCallback* object with retrieved data by batches.
- optionally, provide an implementation for *SampleReferenceProvider* interface, to enable suspend/resume functionality.

For further documentation see the Javadoc of the mentioned interfaces.

6.3.8.6 Resources

N/A

6.3.8.7 References

N/A

6.3.8.8 Processing

As for the EDDS Request Handler, the Request Processor implements the chain of responsibility pattern. Each time a request has to be processed, it will be instantiated the chain of handlers, which will execute the request (batch).

6.3.8.8.1 Generic Batch Chain Processing

For typical batch request, the following handlers are foreseen:

- The Provider Handler: The handler is in charge of performing the data retrieval (also see section 6.3.8.5.6). The way the data is retrieved depends on the Data Provider (Packets, Parameters, etc.). The Provider Handler passes each bunch of retrieved data to a sub set of handlers, which process the data in order to perform the filtering, the formatting and the storage.
    - The Filter Data Handler: The handler applies to each of the retrieved data element the filter options defined in the batch request. The filter data handler processes the filter options which cannot be passed to the Data Handler when it is performed the data retrieval;
    - The Format Data Handler; the handler is responsible for the format of the retrieved data as requested in the batch request and store them on the file system;
- The Files Processor Data Handler is responsible to pass all the finished files through the required files processors:
    - The XML Transformation files processor transforms the XML files to a custom format.
    - The Encryption Processor: The handler is responsible for the encryption of the retrieved data as defined in the batch request;
    - The Compression Processor: The handler is responsible for the compression of the generated data file as defined in the batch request;
    - Delivery files processor – moves finished files to the EDDS Server outbox directory

6.3.8.8.2 Creation of the Batch Request processor

Most of the request processors are implemented extending the abstract class GenericBatchRequestHandlerFactory.

The GenericBatchRequestHandlerFactory provides the protected method createProcessor, which takes as input the batch request and returns the first of the chain of handlers in charge to execute the batch

request. The required handlers are instantiated through a dedicated implementation of this generic factory.

For the chain of command are created the following handlers:

- Data Provider Handler which contains two sub handlers for the filtering and the formatting;
- Files Processor Handler, which contains files processors for XML Transformation, Data Encryption, Compression;
    - The last files processor is always DeliveryFilesProcessor, that moves the completed files to the directory to be picked up by Delivery Manager

6.3.8.8.3 Generic Filter Handler

The Generic Request Processor provides a base implementation of the filter Handler.

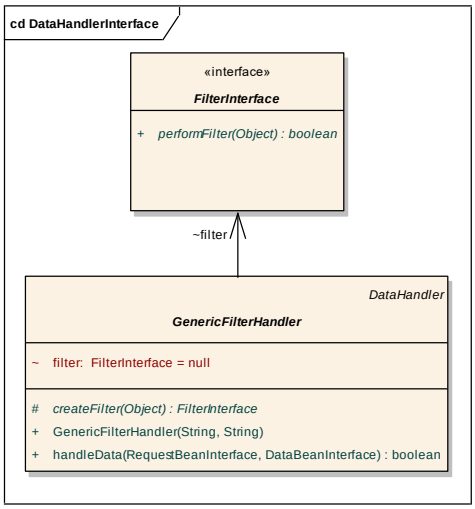

Figure 23 Generic Filter Handler

The GenericFilterHandler implements the DataHandlerInterface. The method handleData gets from the DataBeanInterface argument the list of records to be filtered and applies the filter for each element of the list. The outcome of the filter operation is send to next element of the chain pattern for processing.

Each specific FilterHandler extends the GenericFilterHandler class.

6.3.8.8.3.1 Generic Filtering Processing

The Generic Request processor provides a base implementation of the Filter interface.

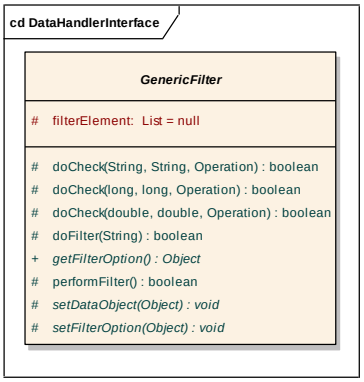

Figure 24 Generic Filter Class

The abstract class implements a basic filtering mechanism based on the assumption that the required filter is passed as a list of filter options. The method doFilter performs the filter operation. It loops for each

filter option and calls using the reflection a method called “check”+<OptionName>. As soon as a “check” method returns false the loop is broken and the record is discarded.

The mechanism allows implementing filters simply extending the GenericFilter class and implementing for each possible filter option a corresponding “check” method.

6.3.8.8.4 Format Handler

The FormatterHandler provides a base framework for formatting the data. It implements the DataSubHandlerInterface.

Most of the formats only need to implement the Formatter interface to format the specific data record into the specified format. Outputting formatted data into files, splitting the files when needed and sending update notifications to database is implemented generically and each of the formats don’t need to re-implement this.

Typically used subclasses of FormatterHander are StringFormatterHandler for writing Strings and BinaryFormatterHandler for writing byte arrays to files.

FormatterHandler also ensures that, for resumable requests, after every split, a reference to the last sample of the last file is marked up. This is needed so that if an error occurs or the request is suspended the reference can be saved to the acknowledgement.

6.3.8.9 Data

N/A

6.3.9 PARC & Data Provision Request Processor (Data Manager, Filter, Formatter, Encryption, Data Compress)

6.3.9.1 Type

The component is implemented as a Java Package.

6.3.9.2 Function

The main function of the PARC Request Processor is to specialise the Request Processor to execute the PARC EDDS Services Requests (batch and stream). In details the following type of requests are supported:

- Telemetry:
  - Telemetry Packet Batch Request (EDDS Raw format) (PktTm);
  - Telemetry Packet Batch Request (PARC Raw format) (PktTmRaw)*;
  - Telemetry Packet Report Batch Request (PktTmReport);
  - Telemetry Packet Gap Report Batch Request (PktTmGapReport)*;Telemetry Packet Statistics Batch Requests (PktTmStatistics);
- Command:
  - Telecommand Packet Batch Request (EDDS Raw format) (PktTc);
  - Telecommand Packet Batch Request (PARC Raw format) (PktTcRaw)*;
  - Telecommand Packet Report Batch Request (PktTcReport);
  - Telecommand Packet Statistics Batch Requests (PktTcStatistics);
- Event:
  - Event Packet Batch Request (EDDS Raw format) (PktEv);
  - Event Packet Batch Request (PARC Raw format) (PktEvRaw)*;
  - Event Packet Report Batch Request (EventRecordReport);
  - Event Packet Statistics Batch Requests (PktEvStatistics);

Request types indicated with a * are only supported by the PARC Manager interface.

For each type of request is implemented a separated Processor. The percentage completion within the acknowledgement message is updated on a regular basis as the packets are retrieved. This is calculated as being the last packet retrieved time, less the start time divided by the requested end time less the start time.

6.3.9.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.9.4 Dependencies

This component depends upon:

- CORBA PARC Manager;
- CORBA Data Provision Services.

6.3.9.5 Interfaces

None.

6.3.9.6 Resources

The output of the executed request is stored on the file system.

6.3.9.7 References

N/A

6.3.9.8 Data

The EUICD [AD-03] details all the possible filtering and formatting capability supported by the PARC Request Processor.

See the java doc documentation for the full code details.

6.3.9.9 Processing

6.3.9.9.1 Data Provision Services

When retrieving data from the Data Provision Services within SCOS to obtain data from the PARC, the following CORBA methods are used:

HFC.StreamingService.retrieveRange(for retrieval of historic data)

HFC.StreamingService.subscribe (for live data)

The first function takes an initial marker (obtained from the method *createMarkerFromTime*) that is the start time of the retrieval, whether to exclude the first packet obtained, the final marker to indicate the end time of the retrieval, the filter to use and the maximum processing period before timing out (set to 60 seconds). The method then returns a response, which contains a batch of zero or more messages, a flag as to whether the end of the history has been reached and the next marker for the next call to the method from which to start retrieval from. The batches of messages are processed by EDDS and passed to the processing chain as described in the following sections. The retrieval ends when either the end of history marker is true, or the next marker time stamp is greater than the end of the filter period specified.

The second function takes a CORBA object that is called with the live data, and a filter string. When live data is received, the callback function is called with the live data.

The same function is used for all packet request types described in the following sections. The service provider is injected into each request service bean, having been obtained via Spring from the CORBA name service:

TC – Command Provision Service

TM – Packet Provision Service

EV – Event Provision Service

OOL – Out of Limits (Behaviour) Provision Service

6.3.9.9.2 Telemetry Packet Batch Request (EDDS Raw format) (PktTm)

6.3.9.9.2.1 Processor Creation

The processors class TmPacketBatchRequestProcessor is in charge of executing the Telemetry Packet Batch Request. The class specialises the GenericBatchRequestProcessor class.

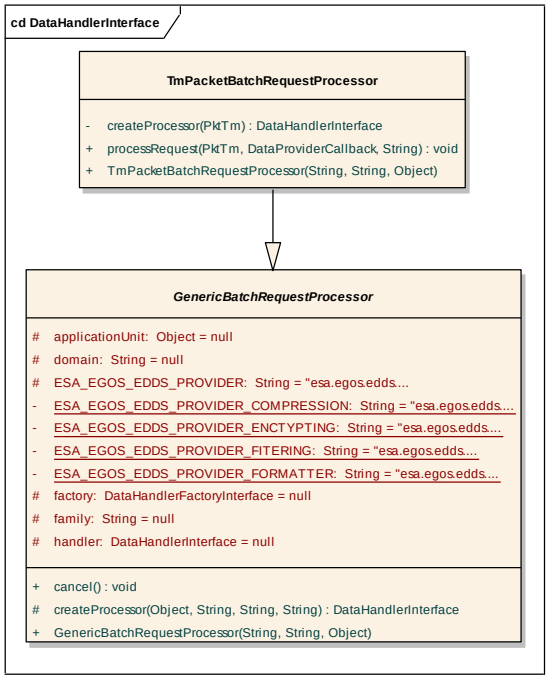

Figure 25 TmPacketBatchRequestProcessor

In order to instantiate the TmPacketBatchRequestProcessor the following parameters are required

- The domain and the family associated with the request;
- The reference to the PARCmanager or Data Provision Service CORBA object;

6.3.9.9.2.2 Request Processing

The Telemetry Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- PktTmManagerDataHandler which retrieves the data from the PARC
- PktTmFilterHandler: which performs the filter on the retrieved data
- Formater: The instantiated class depends on the selected format:
  - PktTmEddsRawFormatter for the EDDS raw formatter;
  - PktTmXFDUFormatterHandler for the XFDU formatter;
  - PktTmGDDS_BINARYFormatter for the backward compatibility with the GDDS;
  - PktTmSFDUFormatterHandler for the SFDU format;

The PktTmManagerDataHandler retrieves the Telemetry Packet through the PARC Manager CORBA interface. In detail, it uses the method viewSelectMode of the viewOperation interface.

The methods arguments are extracted by the request. In detail the following information are taken:

- The list of packets: Since the interface accept only the list of SPID an utility class ApidToSpid converter has been developed to translate the user requests in term of APID, TYPE, SUBTYPE, PI1 and PI2 into a list of SPID. The utility class relays on the SCOS-2000 “pid.dat” table to perform the translation. The location of the pid.dat is defined in the EDDS configuration file. The SPIDs are then matched against the APID and SPID quotas that are set for the current user, if none of the SPIDs pass the quota check then the request will be cancelled but if some of the SPIDs do pass the check a partial delivery will be made.

- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktTmFilterHandler that applies the filter options as defined in the request. The outcome of the filtering operation is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.3 Telemetry Packet Batch Request (PARC Raw format) (PktTmRaw)

6.3.9.9.3.1 Processor Creation

The processors class TmPacketRawBatchRequestProcessor is in charge of executing the Telemetry Packet Batch Request. The class specialises the GenericBatchRequestProcessor class.

In order to instantiate the TmPacketRawBatchRequestProcessor the following parameters are required

- The domain associated with the request;
- The reference to the PARCmanager CORBA object;

6.3.9.9.3.2 Request Processing

The Telemetry Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- PktTmRawManagerDataHandler which retrieves the data from the PARC;
- NoFilterHandler: no filtering is performed, as the data has not been decoded. Instead, the data is simply passed to the next processor unfiltered;
- PktRawXMLFormatter for the XML formatter;
- PktRawBinaryFormatter for the binary formatter;

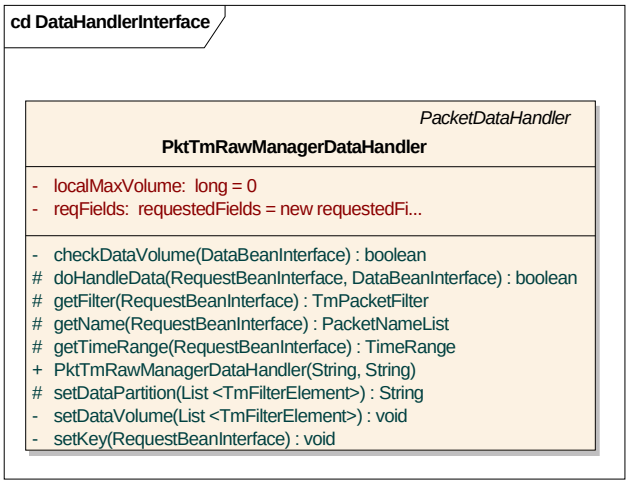

Figure 26 TmPacket Handlers

The PktTmRawManagerDataHandler retrieves the Telemetry Packet through the PARC Manager CORBA interface. It uses one of the following methods of the viewOperation interface, depending on the EDDS configuration:

Method	When used
viewFullPktHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is false
viewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is true
dataspaceViewFullPktHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is false
dataspaceViewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is true

The methods arguments are extracted by the request. In detail the following information are taken:

- The dataspace (if edds.parc.dataspaces.enabled is set to true) from the request or the default if not specified.
- The list of packets: Since the interface accept only the list of SPID an utility class ApidToSpid converter has been developed to translate the user requests in term of APID, TYPE, SUBTYPE, PI1 and PI2 into a list of SPID. The utility class relays on the SCOS-2000 “pid.dat” table to perform the translation. The location of the pid.dat is defined in the EDDS configuration file. The SPIDs are then matched against the APID and SPID quotas that are set for the current user, if none of the SPIDs pass the quota check then the request will be cancelled but if some of the SPIDs do pass the check a partial delivery will be made.
- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktTmRawFilterHandler. The data is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.4 Telemetry Packet Report Batch Request (PktTmReport)

6.3.9.9.4.1 Processor Creation

The processor is created in exactly the same way as for the PktTm processor.

6.3.9.9.4.2 Request Processing

The Telemetry Packet Report batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- PktTmReportManagerDataHandler which retrieves the data from the PARC
- PktTmReportFilterHandler: which performs the filter on the retrieved data
- Formatter: The instantiated class depends on the selected format:
    - PktTmReportXMLFormatter for the XML formatter;
    - PktTmReportBinaryFormatter for the binary formatter;

6.3.9.9.5 Telemetry Packet Gap Report Batch Request (PktTmGapReport)

6.3.9.9.5.1 Processor Creation

The processor is created in exactly the same way as for the PktTm processor.

6.3.9.9.5.2 Request Processing

The Telemetry Packet Gap Report batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- PktTmGapReportManagerDataHandler which retrieves the gap data from the PARC
- PktTmGapReportFilterHandler: which performs the filter on the retrieved data
- Formatter: The instantiated class depends on the selected format:
    - PktTmGapReportXMLFormatter for the XML formatter;
    - PktTmGapReportBinaryFormatter for the binary formatter;

6.3.9.9.6 Telemetry Packet Statistics Batch Requests (PktTmStatistics)

6.3.9.9.6.1 Processor Creation

The processors class TmPacketStatisticsBatchRequestProcessor is in charge of executing the Telemetry Statistics Packet Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.6.2 Request Processing

The Telemetry Statistics Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- PktTmStatisticsManagerDataHandler which retrieves the data from the PARC
- PktTmStatisticsFilterHandler: which performs the filter on the retrieved data
- Format: The instantiated class depends on the selected format:
    - PktTmStatisticsXMLFormatter for the XML formatter;

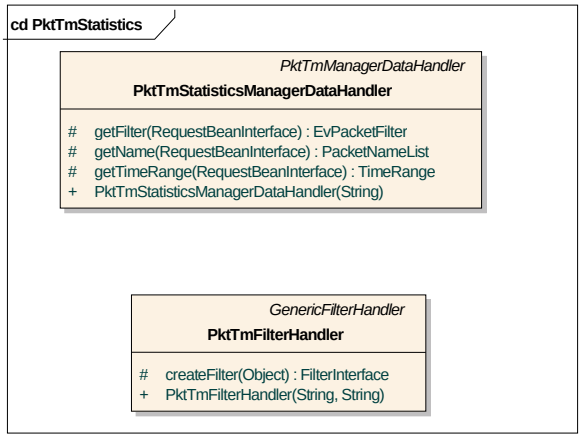

Figure 27 TM Packet Statistics Handlers

The PktTmStatisticsManagerDataHandler retrieves the Telemetry Packet through the PARC Manager or Data Provision Service CORBA interface. The class is derived by the PktTmManagerDataHandler used for the execution of the PktTmPacket Batch Request.

In detail, it uses the method viewSelectMode of the viewOperation interface for the PARC, and the retrieveRange method of the HFC StreamingService for the Data Provision Service.

The methods arguments are extracted by the request. In detail the following information are taken:

- The list of packets: Since the interface accept only the list of SPID an utility class ApidToSpid converter has been developed to translate the user requests in term of APID, TYPE, SUBTYPE, PI1 and PI2 into a list of SPID. The utility class relays on the SCOS-2000 “pid.dat” table to perform the translation. The location of the pid.dat is defined in the EDDS configuration file. The SPIDs are then matched against the APID and SPID quotas that are set for the current user, if none of the SPIDs pass the quota check then the request will be cancelled but if some of the SPIDs do pass the check a partial delivery will be made.
- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktTmStatisticsFilterHandler that applies the filter options as defined in the request. The outcome of the filtering operation is then formatted and stored in the file system. The filter class is derived by the PktTmFilterHandler. The two type of batch request shares the same filtering capabilities.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.7 Telecommand Packet Batch Requests (EDDS Raw format) (PktTc)

6.3.9.9.7.1 Processor Creation

The processors class TcPacketBatchRequestProcessor is in charge of executing the Telecommand Packet Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.7.2 Request Processing

The Command Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request, the following handlers are instantiated once:

- PktTcManagerDataHandler which retrieves the data from the PARC
- PktTcFilterHandler: which performs the filter on the retrieved data
- Format Handler: The instantiated class depends on the selected format:
    - PktTcEddsRawFormatter for the EDDS raw formatter;
    - PktTcXFDUFormatterHandler for the XFDU formatter;
    - PktTcGDDS_BINARYFormatter for the backward compatibility with the GDDS;
    - PktTcSFDUFormatterHandler for the backward compatibility with the GDDS;

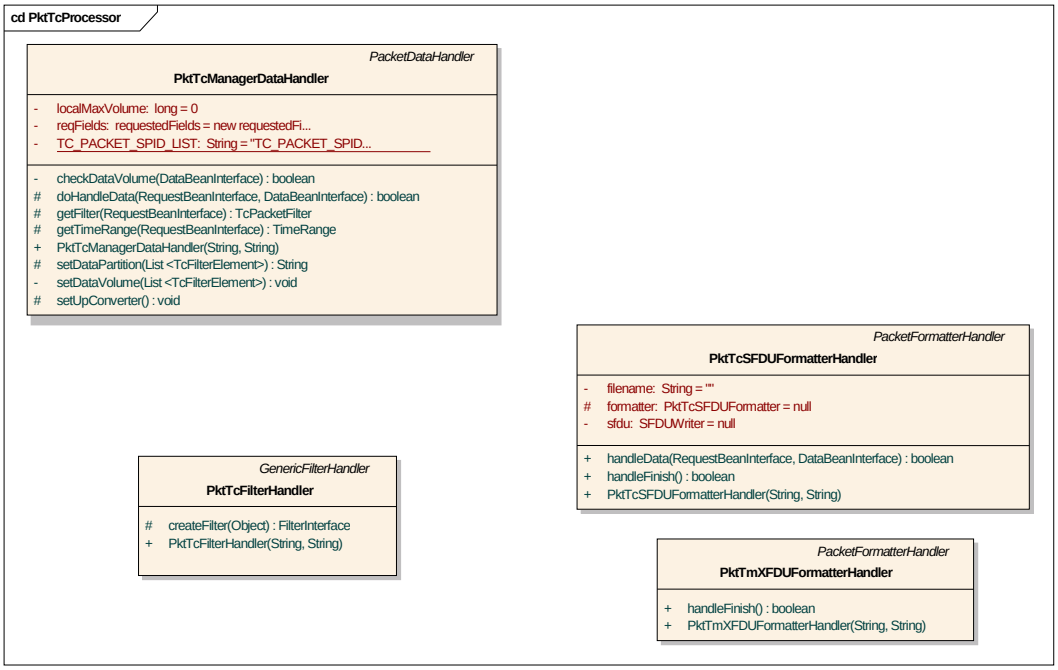

Figure 28 Tc Packet Handlers

The PktTcManagerDataHandler retrieves the Command Packet through the PARC Manager CORBA interface. In detail, it uses the method viewSelectMode of the viewOperation interface.

The methods arguments are extracted by the request. In detail the following information are taken:

- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktTcFilterHandler that applies the filter options as defined in the request. The outcome of the filtering operation is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.8 Telecommand Packet Batch Requests (PARC Raw format) (PktTcRaw)

6.3.9.9.8.1 Processor Creation

The processors class TcPacketRawBatchRequestProcessor is in charge of executing the Telecommand Packet Batch Request (PARC Raw). The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.8.2 Request Processing

The Command Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request, the following handlers are instantiated once:

- PktTcRawManagerDataHandler which retrieves the data from the PARC
- PktTcRawFilterHandler: no filtering is performed, as the data has not been decoded. Instead, the data is simply passed to the next processor unfiltered
- PktTcRawXMLFormatter for the XML formatter;

- PktRawBinaryFormatter for the binary formatter;
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

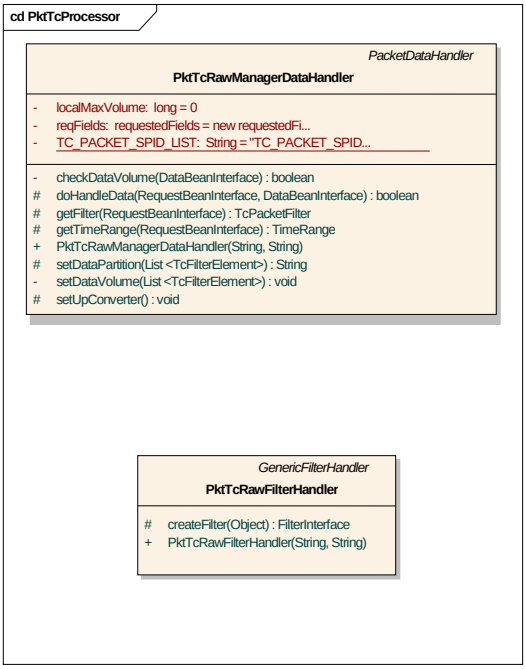

Figure 29 Tc Packet Raw Handlers

The PktTcRawManagerDataHandler retrieves the Command Packet through the PARC Manager CORBA interface. It uses one of the following methods of the viewOperation interface, depending on the EDDS configuration:

Method	**When used**
viewFullPktHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is false
viewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is true
dataspaceViewFullPktHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is false
dataspaceViewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is true

The methods arguments are extracted by the request. In detail the following information are taken:

- The dataspace (if edds.parc.dataspaces.enabled is set to true) from the request or the default if not specified.
- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktTcRawFilterHandler. The data is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.9 Telecommand Packet Report Batch Requests (PktTcReport)

6.3.9.9.9.1 Processor Creation

The PktTcReport processor is essentially the same as for the PktTc processor. The decoding of the TC packet is performed in the PktTcFull class. If the BriefReportFlag is set to false then the TC parameters are also decoded.

6.3.9.9.9.2 Request Processing

The filter options are the same as for PktTc. The available formatters are XML, binary and ASCII. The ASCII report is formatted as per the GDDS ICD [RD-4].

6.3.9.9.10 Telecommand Packet Statistics Batch Requests (PktTcStatistics)

6.3.9.9.10.1 Processor Creation

The processors class TcPacketStatisticsBatchRequestProcessor is in charge of executing the Telecommand Packet Statistics Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.10.2 Request Processing

The Telecommand Statistics Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- PktTcStatisticsManagerDataHandler which retrieves the data from the PARC
- PktTcStatisticsFilterHandler: which performs the filter on the retrieved data
- Format Handler: The instantiated class depends on the selected format:
    - PktTcStatisticsXMLFormatter for the XML formatter;

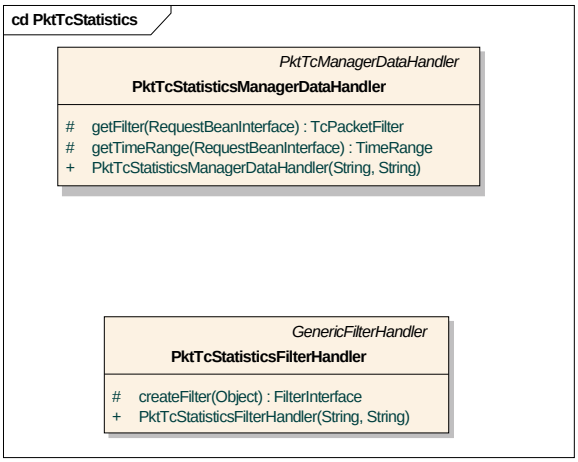

Figure 30 Tc Packet Statistics Handlers

The PktTcStatisticsManagerDataHandler retrieves the TC Packet through the PARC Manager CORBA interface. The class is derived by the PktTcManagerDataHandler used for the execution of the Pkt Tc Packet Batch Request.

In detail, it uses the method viewSelectMode of the viewOperation interface.

The methods arguments are extracted by the request. In detail the following information are taken:

- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktTcStatisticsFilterHandler that applies the filter options as defined in the request. The outcome of the filtering operation is then formatted and stored in the file system. The filter class is derived by the PktTcFilterHandler. The two type of batch request shares the same filtering capabilities.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.11 Event Packet Batch Requests (EDDS Raw format) (PktEv)

6.3.9.9.11.1 Processor Creation

The processor class EvPacketBatchRequestProcessor is in charge of executing the Event Packet Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.11.2 Request Processing

The Event Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request, the following handlers are instantiated once:

- PktEvManagerDataHandler which retrieves the data from the PARC
- PktEvFilterHandler: which performs the filter on the retrieved data
- Formatter: The instantiated class depends on the selected format:
    - PktEvEddsRawFormatter for the EDDS raw formatter;
    - PktEvXFDUFormatterHandler for the XFDU formatter;

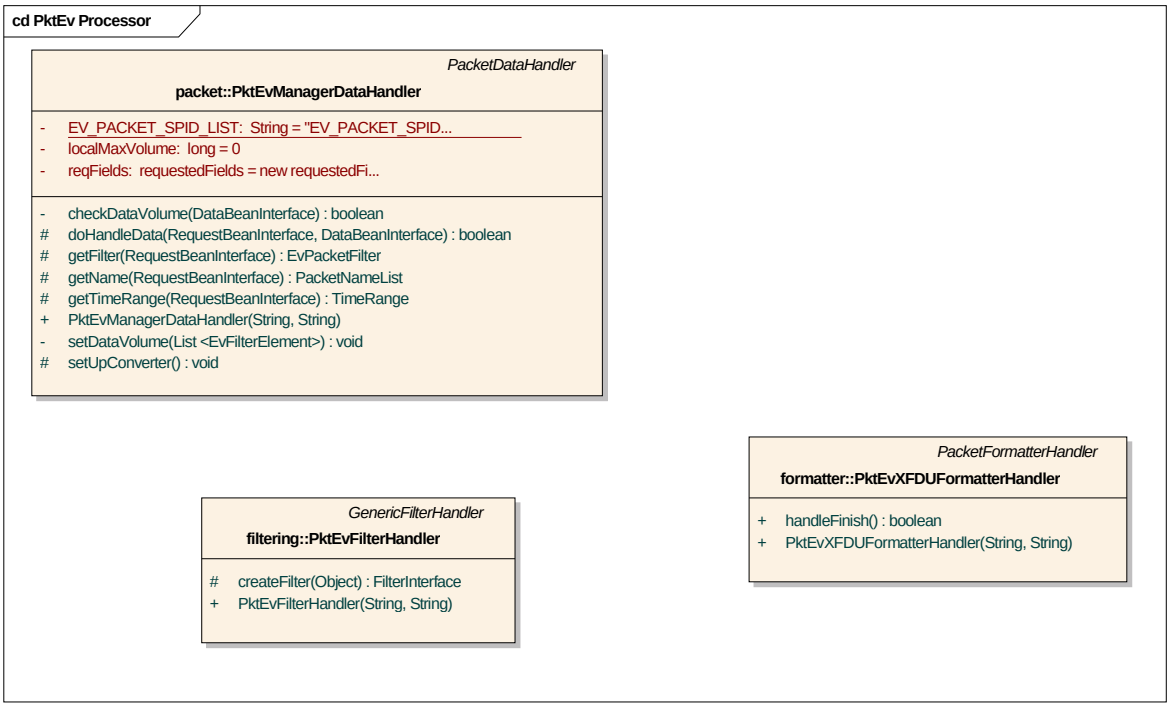

Figure 31 Ev Packet Handlers

The PktEvManagerDataHandler retrieves the Events Packet through the PARC Manager CORBA interface. In detail, it uses the method viewSelectMode of the viewOperation interface.

The methods arguments are extracted by the request. In detail the following information are taken:

- The list of packets: The Event Command request accepts only the SPID as Packet identifier.

- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktEvFilterHandler that applies the filter options as defined in the request. The outcome of the filtering operation is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.12 Event Packet Batch Requests (PARC Raw format) (PktEvRaw)

6.3.9.9.12.1 Processor Creation

The processor class EvPacketRawBatchRequestProcessor is in charge of executing the Event Packet Batch Request (PARC raw format). The class specialises the PacketRawBatchRequestProcessor class.

6.3.9.9.12.2 Request Processing

The Event Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request, the following handlers are instantiated once:

- PktEvRawManagerDataHandler which retrieves the data from the PARC
- PktEvRawFilterHandler: no filtering is performed, as the data has not been decoded. Instead, the data is simply passed to the next processor unfiltered
- PktRawXMLFormatter for the XML formatter;
- PktRawBinaryFormatter for the binary formatter;

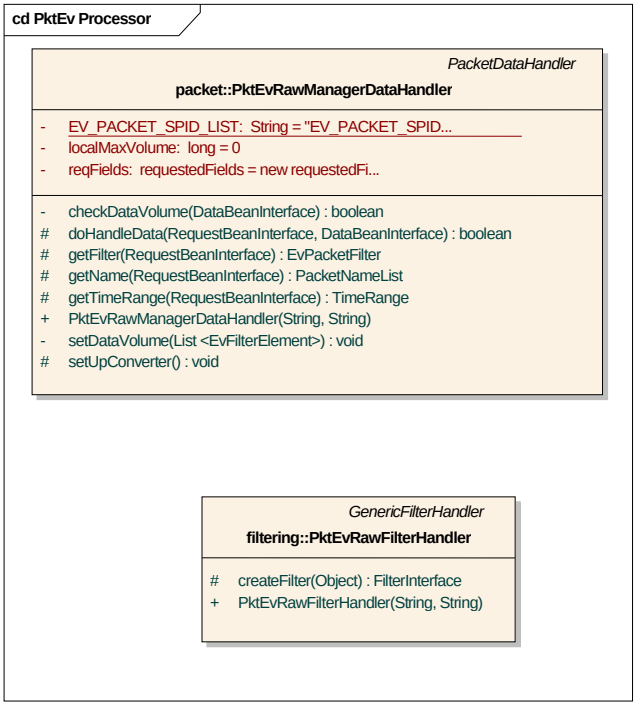

Figure 32 Ev Packet Handlers

The PktEvRawManagerDataHandler retrieves the Events Packet through the PARC Manager CORBA interface. It uses one of the following methods of the viewOperation interface, depending on the EDDS configuration:

Method	When used
viewFullPktHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is false
viewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is true
dataspaceViewFullPktHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is false
dataspaceViewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is true

The methods arguments are extracted by the request. In detail the following information are taken:

- The dataspace (if edds.parc.dataspaces.enabled is set to true) from the request or the default if not specified.
- The list of packets: The Event Command request accepts only the SPID as Packet identifier.
- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktEvRawFilterHandler. The data is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.13 Event Packet Report Batch Request (EventRecordReport)

6.3.9.9.13.1 Processor Creation

The processor is created in exactly the same way as for the PktEv processor.

6.3.9.9.13.2 Request Processing

The Event Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request, the following handlers are instantiated once:

- EventRecordReportManagerDataHandler which retrieves the data from the PARC
- EventRecordReportFilterHandler: which performs the filter on the retrieved data
- Formatter: The instantiated class depends on the selected format:
    - EventRecordReportASCIIFormatter for the ASCII formatter;
    - EventRecordReportXMLFormatter for the XML formatter;
    - EventRecordReportBinaryFormatter for the binary formatter;

6.3.9.9.14 Event Packet Statistics Batch Requests (PktEvStatistics)

6.3.9.9.14.1 Processor Creation

The processors class EvPacketStatisticsBatchRequestProcessor is in charge of executing the Event Packet Statistics Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.14.2 Request Processing

The Event Statistics Packet batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- PktEvStatisticsManagerDataHandler which retrieves the data from the PARC
- PktEvStatisticsFilterHandler: which performs the filter on the retrieved data
- Formatter: The instantiated class depends on the selected format:
    - PktEvStatisticsXMLFormatter for the XML formatter;

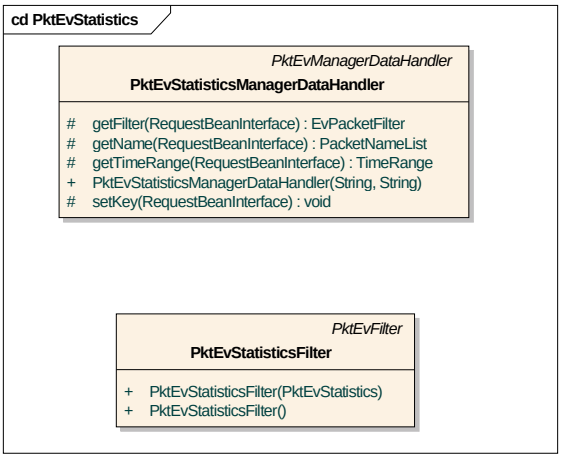

Figure 33 Ev Packet Statistics Handlers

The PktEvStatisticsManagerDataHandler retrieves the event packets through the PARC Manager CORBA interface. The class is derived from the PktEvManagerDataHandler and used for the execution of the PktEvStatistics Batch Request.

In detail, it uses the method viewSelectMode of the viewOperation interface.

The methods arguments are extracted by the request. In detail the following information are taken:

- The list of packets: The Event Command request accepts only the SPID as Packet identifier.
- The time range (start and stop) of the request, based on the PARC Primary Key and PARC Secondary Key;
- The Packet Data Stream;

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor PktEvStatisticsFilterHandler that applies the filter options as defined in the request. The outcome of the filtering operation is then formatted and stored in the file system. The filter class is derived by the PktEvFilterHandler. The two type of batch request shares the same filtering capabilities.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.15 Out Of Limit Report Batch Requests (OolRecordReport)

6.3.9.9.15.1 Processor Creation

The processors class OolRecordReportBatchRequestProcessor is in charge of executing the Event Packet Statistics Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

6.3.9.9.15.2 Request Processing

The Out of Limit Report batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- OolRecordReportManagerDataHandler which retrieves the data from the PARC or Data Provision Services
- OolRecordReportFilterHandler: which performs the filter on the retrieved data
- Formatter: The instantiated class depends on the selected format:
    - OolRecordReportASCIIFormatter for the plaintext formatter;
    - OolRecordReportXMLFormatter for the XML formatter;
    - OolRecordReportBinaryFormatter for the binary formatter;

The OolRecordReportManagerDataHandler retrieves the OOL Packets through the PARC Manager CORBA interface. It uses one of the following methods of the viewOperation interface, depending on the EDDS configuration:

Method	**When used**
viewFullPktHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is false
viewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is false and edds.parc.bigendian.enabled is true
dataspaceViewFullPktHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is false
dataspaceViewFullPktBigEndianHexDump	edds.parc.dataspaces.enabled is true and edds.parc.bigendian.enabled is true

Should the SCOS Data Provision Services have been requested, then EDDS will use the Out of Limits (BEHV) streaming service.

The packets are retrieved in array of fixed size. The retrieved packets are then passed to the chain successor OolRecordReportFilterHandler. The data is then formatted and stored in the file system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.9.9.16 Packet Raw Formatter

The raw packet formatter is an implementation of the FormatterInterface, which implements the basic mechanism used for the formatting of the TM/TC/EV packets according to the raw format described in the EUICD [AD-03]. It should be noted that the term “raw” here means as stored by the PARC, which is a hexadecimal encoded representation of the raw data.

The class implements a basic formatting mechanism based on the assumption that the required packet fields are passed through a configurable XML data structure (packet header). The method format performs the formatting operation. It loops for each required packet field and calls using the reflection a method called “get”+<Packet Field>.

The mechanism allows implementing the formatters by extending the PacketRawFormatter class and implementing for each possible packet field a corresponding “get” method.

It should be noted that the data retrieved from the PARC Manager is in a hex encoded format. The EDDS will decode this information before formatting the data into the EDDS raw format. This is necessary as some of the EDDS header fields that are permitted are stored within the packet body.

6.3.9.9.17 XFDU Formatter

The XFDU formatter creates a new XFDUWrapper object, which is used to create the XFDU data file. The XFDUWrapper class is a wrapper for the XFDU library. It performs some additional operations that are missing in the XFDU library, such as computing a checksum for the included files. Once the data files to be included in the XFDU file have been created, a new XFDU file is created, and the data files are added to the file in a Base64 encoded way, complete with data about the data files and the checksum.

6.3.9.9.18 Packet Statistics XML Formatter

The XML packet formatter is an implementation of the FormatterInterface, which implements the mechanism used for the formatting of the TM/TC/EV statistics packets according to the XML format described in the EUICD [AD-03].

6.3.10 FARC Request Processor (Data Manager, Filter, Formatter, Encryption, Data Compress)

6.3.10.1 Type

The component is implemented as a Java Package.

6.3.10.2 Function

The main function of the FARC Request Processor is to specialise the Request Processor to execute the FARC EDDS Services Requests (batch and stream). In details the following type of requests are supported:

- Catalogue Batch Request (ArchiveCatalogue);
- File Batch Request (ArchiveFile);
- FARC Subscription Request (ArchiveSubscription)

For each type of request is implemented a separated Processor.

6.3.10.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.10.4 Dependencies

This component depends upon:

- FARC Java API

6.3.10.5 Interfaces

None.

6.3.10.6 Resources

The output of the executed request is stored on the file system.

6.3.10.7 References

N/A

6.3.10.8 Data

The EUICD [AD-03] details all the possible filtering and formatting capability supported by the FARC Request Processor.

See the java doc documentation for the full code details.

6.3.10.9 Processing

6.3.10.9.1 Catalogue Batch Request (Archive Catalogue)

6.3.10.9.1.1 Processor Creation

The processors class ArchiveCatalogueRequestProcessor is in charge of executing the Archive Catalogue Batch Request. The class specialises the ArchiveProcessor class.

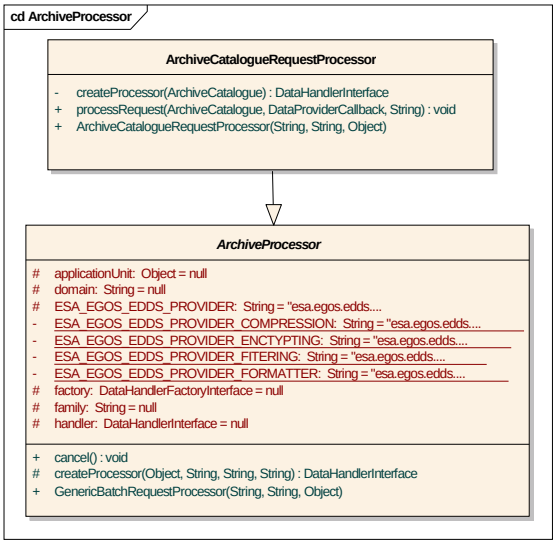

Figure 34 ArchiveCatalogueRequestProcessor

In order to instantiate the ArchiveCatalogueRequestProcessor the following parameters are required:

- The domain and the family associated with the request;
- The reference to the FARC Server CORBA object.

6.3.10.9.1.2 Request Processing

The Archive Catalogue batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- ArchiveCatalogueManagerDataHandler which retrieves the data from the FARC and passes the filter information to the FARC.
- Formatter: The instantiated class depends on the selected format:
  - ArchiveCatalogueXMLFormatter for the XML formatter;
  - ArchiveCatalogueASCIIFormatter for the ASCII formatter.

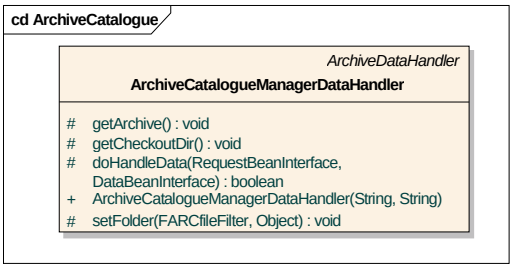

Figure 35 Archive Catalogue Manager Data Handler

The ArchiveCatalogueManagerDataHandler retrieves the catalogue XML data through the FARC Server CORBA interface. The class is derived by the ArchiveCatalogueDataHandler used for the execution of the ArchiveCatalogue Batch Request.

In detail, it uses the method FARCoperGetFileCatalogue of the FARC interface.

The method's arguments are extracted from the request. In detail the following information is taken to populate the FARC filter:

- The folder the file(s) are in (optional);

- The type of file (optional);
- The name of the file (optional);
- The creation time of the file(s) (optional);
- The commit time of the file(s) (optional);
- The version of the file(s) (optional);

In addition, the FARC archive to perform the request on is taken from the EDDS mission configuration (ARCHIVE_FAMILY) and should be the text name of the archive (e.g. OPERATIONAL, PRIME).

The quota checks are then done on the retrieved catalogue data. If no data passes the checks the request is cancelled.

The catalogue data that passes the quota checks is passed to the chain successor for formatting the data as either ASCII or XML. No filtering handler is required as the FARC has a built-in filter handler. The formatted data is the stored on the filing system.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.10.9.2 File Batch Request (ArchiveFile)

6.3.10.9.2.1 Processor Creation

The processors class ArchiveFileRequestProcessor is in charge of executing the Archive File Batch Request. The class specialises the ArchiveProcessor class.

6.3.10.9.2.2 Request Processing

The Archive File batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- ArchiveFileManagerDataHandler which retrieves the data from the FARC and passes the filter information to the FARC.
- Format Handler: The instantiated class depends on the selected format:
  - ArchiveFileBinaryFormatterHandler for the binary formatter;
  - ArchiveFileXFDUFormatterHandler for the XFDU formatter.
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

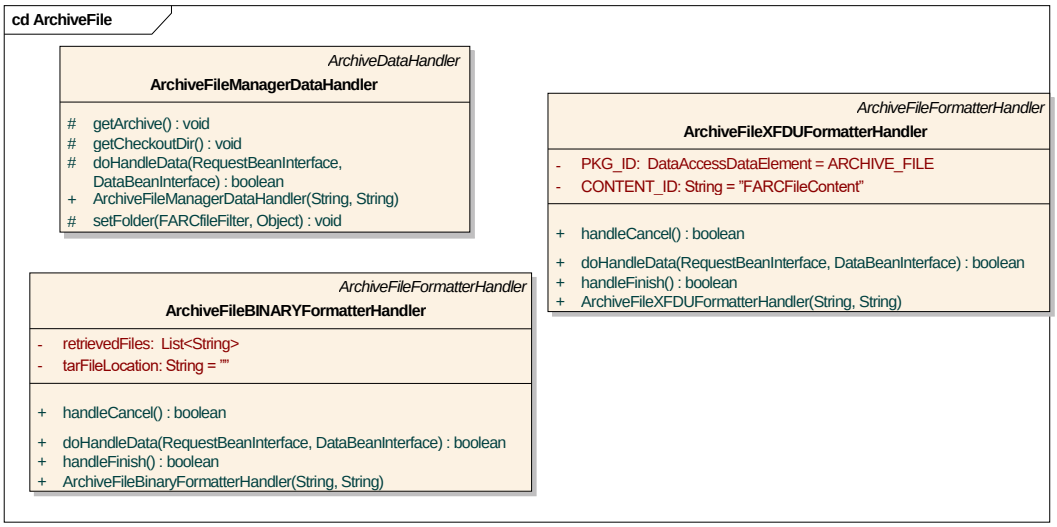

Figure 36 Archive File Handlers

First, a catalogue request is created using the user defined filters and the FarcCatalogueRetriever. The resulting catalogue data (list of files) is combined with the original filters to create new filters and has to then pass the quota checks. The list of combined filters that do pass the quota checks will then be used to create new FARC file request.

The ArchiveFileManagerDataHandler retrieves the request data through the FARC Server CORBA interface. The class is derived by the ArchiveFIleDataHandler used for the execution of the ArchiveFile Batch Request.

In detail, it uses the class FARCHandler, which in turn uses the registerAsyncTask of the FARC interface. This enables EDDS to timeout the operation if the FARC does not finish delivering the requested file(s) so as to avoid waiting indefinitely.

The method's arguments are extracted from the request. In detail the following information is taken to populate the FARC filter:

- The folder the file(s) are in (optional);
- The type of file (optional);
- The name of the file (optional);
- The creation time of the file(s) (optional);
- The commit time of the file(s) (optional);
- The version of the file(s) (optional);
- The description of the file(s) (optional);
- The comment of the file(s) (optional).

All the parameters are optional. In case the user will not provide any filter, the FARC (and so EDDS) will return:

- The complete catalogue, in case of *file catalogue request*;
- No file, in case of *file request*;

In addition, the FARC archive to perform the request on is taken from the EDDS mission configuration (ARCHIVE_FAMILY) and should be the text name of the archive (e.g. OPERATIONAL, PRIME).

The retrieved data files are then passed to the chain successor for formatting the data as either raw binary data (as stored in the FARC) or XFDU. No filtering handler is required as the FARC has a built-in filter handler. The formatted data is the stored on the filing system. The XDFU handler will encode the requested data files as Base64 and add the data to an XFDU package file.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.10.9.3 Subscription Batch Request (Archive Subscription)

6.3.10.9.3.1 Processor Creation

The processors class ArchiveSubscriptionRequestProcessor is in charge of executing the Archive Subscription Batch Request.

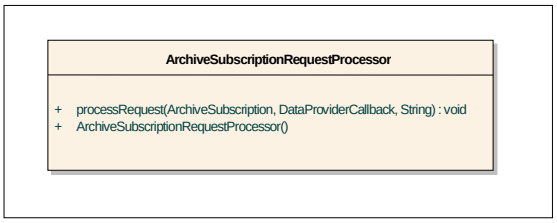

Figure 37 ArchiveSubscriptionRequestProcessor

6.3.10.9.3.2 Request Processing

The Archive Subscription batch request is executed through the method processRequest. This method checks that the expiry time specified is valid (i.e. in the future) and then simply notifies the FARC Subscription Manager. The FARC Subscription Manager then stores the details of the subscription in the database, so that the subscription is persisted between server restarts, and schedules a job with the Quartz Scheduler so that the subscription expires at the correct time. When notifications are received from the FARC; the FARC Subscription Manager checks if the subscribed user has enough privileges defined in the quota sets to get notified of the change, then creates a new ArchiveFile request to retrieve the new data from the FARC and/or sends an e-mail notification. When the expiry message is eventually delivered, the FARC Subscription Manager deletes the subscription from the database and sets the request to SERVER_COMPLETED. Cancellation notifications are also passed to the FARC Subscription Manager via the CancelExecuteHandler. The FARC Subscription Manager then deletes the subscription from the database and sets the request to CANCELED.

6.3.11 EDDS File System Request Processor (Data Provider, Filter, Formatter)

6.3.11.1 Type

This component is implemented as a Java project.

6.3.11.2 Function

The component provides the user with data from the EDDS File System Index.

The main function of the File System Request Processor is to specialise the Request Processor to execute the File System EDDS Services Requests. In details the following type of requests are supported:

- File System File Catalogue Batch Request (FileSystemFileCatalogue);
- File System Folder Catalogue Batch Request (FileSystemFolderCatalogue);
- File System File Batch Request (FileSystemFile);
- File System Subscription Request (FileSystemSubscription);

For each type of request a separate Processor is implemented.

6.3.11.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;
- EDDS File System Index;

6.3.11.4 Dependencies

This component depends upon:

- The Index File Helper interface of the EDDS File System Index

6.3.11.5 Interfaces

None.

6.3.11.6 Resources

The output of the executed request is stored on the file system.

6.3.11.7 References

N/A

6.3.11.8 Processing

6.3.11.8.1 File System File Catalogue Batch Request (FileSystemFileCatalogue)

6.3.11.8.1.1 Processor Creation

The processors class FileSystemFileCatalogueRequestHandlerFactory is in charge of creating executors for these requests. The class specialises the FileSystemCatalogueRequestHandlerFactory.

6.3.11.8.1.2 Request Processing

This request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- FileSystemFileCatalogueDataProvider which retrieves the data from the Index and handles the quota restrictions;
- ResumableProviderDataHandler which uses the FileSystemFileCatalogueDataProvider to fetch data;
- Format Handler: The instantiated class depends on the selected format:
  - FileSystemFileCatalogueASCIIFormatter for the plain text formatter;
  - FileSystemFileCatalogueXMLFormatter for the XML formatter;
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

6.3.11.8.2 File System Folder Catalogue Batch Request (FileSystemFolderCatalogue)

6.3.11.8.2.1 Processor Creation

The processors class FileSystemFolderCatalogueRequestHandlerFactory is in charge of creating executors for these requests. The class specialises the FileSystemCatalogueRequestHandlerFactory.

6.3.11.8.2.2 Request Processing

This request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- FileSystemFolderCatalogueDataProvider which retrieves the data from the Index and handles the quota restrictions;
- ResumableProviderDataHandler which uses the FileSystemFolderCatalogueDataProvider to fetch data;
- Format Handler: The instantiated class depends on the selected format:
  - FileSystemFolderCatalogueASCIIFormatter for the plain text formatter;
  - FileSystemFolderCatalogueXMLFormatter for the XML formatter;
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

6.3.11.8.3 File System File Batch Request (FileSystemFile)

6.3.11.8.3.1 Processor Creation

The processors class FileSystemFileRequestHandlerFactory is in charge of creating executors for these requests. The class specialises the GenericBatchRequestHandlerFactory.

6.3.11.8.3.2 Request Processing

This request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- FileSystemFileDataProvider which retrieves the data from the Index and handles the quota restrictions;
- ResumableProviderDataHandler which uses the FileSystemFileDataProvider to fetch data;
- FileTARFormatter to compose an archive of the resulting files;
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

6.3.11.8.4 File System File Catalogue Batch Request (FileSystemSubscription)

6.3.11.8.4.1 Processor Creation

The processors class FileSystemSubscriptionRequestHandlerFactory is in charge of creating executors for these requests. The class implements the BatchRequestHandlerFactory.

6.3.11.8.4.2 Request Processing

On the execution of the subscription request, an entry is stored in the EDDS database and the state of the request will stay ACTIVE until it expires.

On every File System update notification, the subscription table will be checked for matching subscriptions and if any matches are found, the subscriptions will be notified based on the options selected on the request (File System File request for the matching file or just acknowledgement update is created and sent via email).

When the subscription request expires or is cancelled, the subscription entry will be removed from the database. Also, when the subscription is an exact match to a file that was deleted, the subscription is then marked as finished.

6.3.12 DARC Request Processor (Data Manager, Filter, Formatter, Encryption, Data Compression)

6.3.12.1 Type

The component is implemented as a Java project.

6.3.12.2 Function

The main function of the DARC Request Processor is to specialise the Request Processor to execute the DARC EDDS Services Requests (batch and stream). In details the following type of requests are supported:

- Telemetry Parameter Batch Request (Param);
- Telemetry Parameter Statistics Batch Request (ParamStatistics);
- Telemetry Parameter Definition Batch Request (ParamDefinition);
- Telemetry Parameter Preview Batch Request (ParamPreview);
- Telemetry Parameter Report Batch Request (ParamReport);

For each type of request is implemented a separated Processor

6.3.12.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.12.4 Dependencies

This component depends upon:

- DARC Java APIs. The following interface is used:
  - IDataRetrieval

6.3.12.5 Interfaces

None.

6.3.12.6 Resources

The output of the executed request is stored on the file system.

6.3.12.7 References

N/A

6.3.12.8 Processing

6.3.12.8.1 Parameter XML Formatter

The Parameter XML Formatter takes the data retrieved from the DARC and populates an XML structure. The data is then streamed to disk as it is retrieved, thus ensuring the (potentially large) data retrieved from the DARC is not stored in memory.

6.3.12.8.2 Parameter TDRS Spreadsheet Formatter

The Parameter TDRS Formatter takes the data retrieved from the DARC and populates a TDRS like spreadsheet format. This is a text format with data separated by tabs and is based on the TDRS ICD [RD-11]. The data is received from the DARC one parameter at a time, so all the data received needs to be combined. This is achieved by storing the data for each parameter into a temporary binary

file on disk, and then reading in each file a line at a time to combine the data. This ensures that all the data is not stored in memory.

The TDRS Spreadsheet output is backwards compatible with the TDRS spreadsheet format with the following limitations:

- The Raw value and Raw validity are not supported because the DARC contains only the Engineering values of the telemetry parameters;
- The parameter sample out of limit status is not supported because the DARC currently does not provide them.

6.3.12.8.3 Parameter Spread Sheet Formatter (via XSLT)

Note: The Spreadsheet output is backward compatible with the TDRS spreadsheet format with the limitations as described in the previous section plus the following additional limitation:

- The processing time with XSL Transformation is much longer than without transformation, because in order to calculate the statistics for the parameters (this is not provided by DARC) all the parameter data needs to be read into the memory and processed.

6.3.12.8.4 Parameter Binary Formatter

Parameter Binary Formatter puts the data into an efficient format for further machine processing. Please see [AD-3] for more info on Parameter Binary format.

6.3.12.8.5 Telemetry Parameter Batch Request (Param)

6.3.12.8.5.1 Processor Creation

The processors class ParameterBatchRequestProcessor is in charge of executing the Telemetry Parameter Batch Request. The class specialises the GenericBatchRequestProcessor class.

6.3.12.8.5.2 Request Processing

The Telemetry Parameter batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- ParamManagerDataHandler which retrieves the data from the DARC;
- ParamFilterHandler: which performs the filter on the retrieved data;
- Format Handler: The instantiated class depends on the selected format:
    - ParamXMLFormatterHandler for the XML formatter;
    - ParamXFDUFormatterHandler for the XFDU formatter;
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

The ParamManagerDataHandler retrieves the Parameter samples through the DARC Java interface IDataRetrieval. In detail, the implementation of the method handleData performs the following operations:

1. Checks whether the parameter names specified in the filter pass the quota restrictions set for the current user.
2. Checks the validity of the specified time window against quota restrictions.
3. Retrieves the data from the backend in one of the two ways:
    1. When all the parameters are requested by user (the * wildcard is used), there are no applicable quota restrictions for retrievable parameters and TDRS formatting is not used
        - Builds the DARC Sample request for retrieving all the parameters within the requested time range;

- Requests samples for all the types and passes the data on to the next processing step

2. Else if the conditions above are not satisfied
    - Builds the DARC Sample request retrieving the list of parameters and the time range from the EDDS batch request;
    - Requests the parameter samples for each parameter and passes the data to the formatter handler to write the data to disk;

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.12.8.6 Telemetry Parameter Statistics Batch Request (ParamStatistics)

6.3.12.8.6.1 Processor Creation

The processors class ParamStatisticsRequestProcessor is in charge of executing the Telemetry Parameter Statistics Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

In order to instantiate the ParamStatisticsRequestProcessor the following parameters are required:

- The reference to the DARC DataProvisioningFactory. The factory is used by the processor to instantiate the IDataRetrieval interface needed for the Parameter information retrieval. The getParamSamples method is used in IDataRetrieval.

6.3.12.8.6.2 Request Processing

The Telemetry Parameter Statistics batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- ParamStatisticsManagerDataHandler which retrieves the data from the DARC;
- ParamStatisticsFilterHandler: which performs the filter on the retrieved data;
- Formatter: The instantiated class depends on the selected format:
    - ParamStatisticsXMLFormatter for the XML formatter;
    - ParamStatisticsASCIIFormatter for the ASCII formatter;

The ParamStatisticsManagerDataHandler retrieves the Parameter statistics through the DARC Java interface IDataRetrieval. In detail, the implementation of the method handleData performs the following operations:

1. Checks whether the parameter names specified in the filter pass the quota restrictions set for the current user.
2. Checks the validity of the specified time window against quota restrictions.
3. Builds the DARC Sample request retrieving the list of parameters and the time range from the EDDS batch request;
4. Issues the parameter samples request
5. Waits until the data has been retrieved;

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.12.8.7 Telemetry Parameter Definition Batch Request (ParamDefinition)

6.3.12.8.7.1 Processor Creation

The processors class ParamDefinitionRequestProcessor is in charge of executing the Telemetry Parameter Definition Batch Request. The class specialises the ParamHandlerFactory class.

In order to instantiate the ParamDefinitionRequestProcessor the following parameters are required:

- The reference to the DARC DataProvisioningFactory. The factory is used by the processor to instantiate the IDataRetrieval interface needed for the Parameter information retrieval. The getStaticParameterList method is used within IDataRetrieval.

6.3.12.8.7.2 Request Processing

The Telemetry Parameter Definition batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- ParamDefinitionManagerDataHandler which retrieves the data from the DARC;
- ParamDefinitionFilterHandler: which performs the filter on the retrieved data;
- Format Handler: The instantiated class depends on the selected format:
  - ParamDefinitionXMLFormatter for the XML formatter;

The ParamDefinitionManagerDataHandler retrieves the Parameter definitions through the DARC Java interface IDataRetreival. In detail, the implementation of the method handleData performs the following operations:

1. Uses DARC IDataRetrieval interface to query all the parameter definitions from DARC
2. For each retrieved parameter checks whether the parameter name passes the quota restrictions set for the current user.
3. Waits until the data has been retrieved;

When the retrieval operation is completed, the filtering, encryption and compression handlers are invoked sequentially (if part of the request).

6.3.12.8.8 Telemetry Parameter Preview Batch Request (ParamPreview)

6.3.12.8.8.1 Processor Creation

The processors class ParamPreviewRequestProcessor is in charge of executing the Telemetry Parameter Preview Batch Request. The class specialises the GenericBatchRequestHandlerFactory class.

In order to instantiate the ParamPreviewRequestProcessor the following parameters are required:

- The reference to the DARC DataProvisioningFactory. The factory is used by the processor to instantiate the IDataRetrieval interface needed for the Parameter information retrieval.

6.3.12.8.8.2 Request Processing

The Telemetry Parameter Preview batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once :

- ParamPreviewManagerDataHandler which retrieves the data from the DARC;
- ParamPreviewFilterHandler: which performs the filter on the retrieved data;
- Formatter: The instantiated class depends on the selected format:
  - ParamPreviewXMLFormatter for the XML formatter;
  - ParamPreviewASCIIFormatter for the ASCII formatter;

The ParamPreviewManagerDataHandler retrieves the Parameter statistics through the DARC Java interface IDataRetrieval. In detail, the implementation of the method handleData performs the following operations:

1. Checks whether the parameter names specified in the filter pass the quota restrictions set for the current user.
2. Checks the validity of the specified time window against quota restrictions.
3. Builds the DARC Sample request retrieving the list of parameters using IDataRetrieval interface and the time range from the EDDS batch request;
4. Requests the parameter samples for each parameter and passes the data to the formatter handler to write the data to disk.

When the retrieval operation is completed, the encryption and compression handlers are invoked sequentially (if part of the request).

6.3.13 EDDS SMON Request Processor (Data Provider, Filter, Formatter)

6.3.13.1 Type

This component is implemented as a Java project.

6.3.13.2 Function

The component provides the user with parameter data from SMON service.

The main function of the SMON Request Processor is to specialise the Request Processor to execute the SMON EDDS Services Requests. In details the following type of requests are supported:

- Telemetry Parameter Batch Request (SMONParam);

6.3.13.3 Subordinates

The subordinates of this component are:

- EDDS Configuration Manager;

6.3.13.4 Dependencies

This component depends upon:

- SMON Java APIs
- TKMA Java APIs

6.3.13.5 Interfaces

None.

6.3.13.6 Resources

The output of the executed request is stored on the file system.

6.3.13.7 References

N/A

6.3.13.8 Processing

6.3.13.8.1 Parameter XML Formatter

The SMON Parameter XML Formatter takes the data retrieved from SMON and populates an XML structure. The data is then streamed to disk as it is retrieved, thus ensuring the (potentially large) data retrieved from the SMON is not stored in memory.

6.3.13.8.2 Parameter TDRS Spreadsheet Formatter

The Parameter TDRS Formatter takes the data retrieved from the SMON and populates a TDRS like spreadsheet format. This is a text format with data separated by tabs and is based on the TDRS ICD [RD-11].

6.3.13.8.3 Parameter Spread Sheet Formatter (via XSLT)

Note: The Spreadsheet output is backward compatible with the TDRS spreadsheet format with the limitations as described in the previous section plus the following additional limitation:

- The processing time with XSL Transformation is much longer than without transformation, because in order to calculate the statistics for the parameters all the parameter data needs to be read into the memory and processed.

6.3.13.8.4 Parameter Binary Formatter

SMON Parameter Binary Formatter puts the data into an efficient format for further machine processing. Please see [AD-3] for more info on SMON Parameter Binary format.

6.3.13.8.5 Telemetry Parameter Batch Request (SMONParam)

6.3.13.8.5.1 Processor Creation

The processors class SMONParamBatchRequestProcessor is in charge of the executing of the SMON Telemetry Parameter Batch Requests. The class specialises the GenericBatchRequestProcessor.

6.3.13.8.5.2 Request Processing

The SMON Telemetry Parameter batch request is executed through the method processRequest to which are passed: (a) the batch request, (b) the reference to the call-back used to get notified about the request execution and (c) the request Id.

For each request the following handlers are instantiated once:

- SMONParameterProvider which retrieves the data from SMON;
- ResumableProviderDataHandler which uses the SMONParameterProvider to fetch data from SMON;
- SMONParamFilterHandler: which performs the filter on the retrieved data;
- Format Handler: The instantiated class depends on the selected format:
    - SMONParamXMLFormatterHandler for the XML formatter;
    - XFDUFormatterHandler for the XFDU formatter which uses the XML formatter;
    - SMONParamBinaryFormatter for the binary formatter;
    - SMONParamDARCXMLFormatter for the XML formatter of the DARC format;
    - SMONParamDARCBinaryFormatter for the binary formatter of the DARC format;
- Encryption handler as defined in the request;
- Compression Handler as defined in the request;

All the formats, except the DARC ones, have the option to select a preferred representation (raw, engineered or all). Selecting to have only the "raw" or "engineering" representations in the response file will result in only those representations being present in the output, even if there are other representations. However, if a parameter sample is found that doesn't have the selected representation at all, it will still be included in the output, but will clearly show what representation it is.

6.3.14 EDDS Report Request Processor

6.3.14.1 Type

This component is part of the EDDS Server.

6.3.14.2 Function

The function of this component is to provide EDDS reports to the user. The following reports are provided:

- EDDS Usage Report

6.3.14.3 Subordinates

None

6.3.14.4 Dependencies

The dependencies of this component are:

- Database Manager – used to retrieve data from the database;

6.3.14.5 Resources

N/A

6.3.14.6 References

N/A

6.3.14.7 Processing

The EDDS Server processes the request as a batch request. This follows the same chain of command as all other batch requests, with the data retrieval going straight to the database rather than via an external archive such as the PARC, DARC or FARC. The report format is XML, but the user can use the XML Transform option to have the XML transformed to another format using a stylesheet.

6.3.14.8 Data

N/A

6.3.15 EDDS User Request Processor

6.3.15.1 Type

The component is implemented as a Java package, and is performed on the EDDS Server.

6.3.15.2 Function

In detail the following type of requests are supported:

- Data Access Set management (DataAccessSet);
- User Group management (Group);
- Operation Set management (OperationSet);
- Quota Set management (QuotaSet);
- Role management (Role);
- User Account management (UserAccount).

6.3.15.3 Subordinates

The subordinates of this component are:

- LdapManagementHelper – used to connect to the LDAP server and set / update / retrieve data;
- DataAccessHelper – user to perform operations on data access sets;
- GroupHelper – used to perform operations on groups;
- OperationDataSetHelper - used to perform operations on operation data sets;
- ServiceDataSetHelper - used to perform operations on service data sets;
- UserAccountHelper - used to perform operations on user accounts;
- UserQuotaSetHelper - used to perform operations on quota sets;
- UserRoleHelper - used to perform operations on roles.

6.3.15.4 Dependencies

This component has no dependencies: LDAP support is built-in to the standard Java library.

6.3.15.5 Interfaces

None

6.3.15.6 Resources

The output of the executed request is stored in LDAP

6.3.15.7 References

N/A

6.3.15.8 Processing

6.3.15.8.1 Request Processing

The User Management Request Handler is executed through the method checkPrivilegesAndExecute to which are passed: (a) the account request, (b) the request Id.

For each request the following handlers are instantiated once:

- UserManagementAuthorisationHandler which checks if the user performing the request has sufficient privileges to perform the operation;
- LdapManagementHelper: which interfaces the calls to update LDAP.

The EDDS server uses helper classes to perform the actual LDAP operations. In detail, the implementation of the method checkPrivilegesAndExecute performs the following operations:

1. Check the type of the request (create, delete, update) and calls the LdapManagementHelper class to perform the relevant operation;
2. The LdapManagementHelper attempts to connect to the LDAP server, then calls the relevant helper to perform the LDAP operation;
3. The helper then performs the operation to make the changes to LDAP.

6.3.15.8.1.1 User

6.3.15.8.1.1.1 Login

Before being able to submit any requests, the user has to open a session by logging in to EDDS. During the login process, it is checked that

1. the user is enabled (is not suspended)
2. the given password matches with the one stored in LDAP
3. the user password is not expired

If the password does not match, the incorrect logins counter is increased for that user and user is suspended if it reaches the maximum incorrect logins limit, unless it is the last enabled EDDS admin.

6.3.15.9 Data

See the Javadoc documentation for the full code details.

6.3.16 EDDS Database Manager

6.3.16.1 Type

The component is implemented as a Java Package.

6.3.16.2 Function

The Database Manager is a façade to the EDDS Database. The DB Manager is a component in charge of handling all the persistence mechanisms between the application and the DB: in particular any Create, Read, Update, Delete (CRUD) operation and the management of concurrent transactions.

The DB resides with the EDDS Archiver and EDDS Server and contains information related to any request. The usage of the DB allows the framework to manage the acknowledgement, to produce reports and to store logs. The usage of a DB has been foreseen in order to facilitate the persistence management in the application and more over in order to simplify the generation of EDDS reports.

6.3.16.3 Subordinates

None

6.3.16.4 Dependencies

This component depends upon:

- JDBC connector API for MariaDB;
- MyBatis Core Framework;
- MyBatis Spring library;
- Java Logging standard library;

6.3.16.5 Interfaces

The interface to the library is the façade DaoAccess.

6.3.16.6 Resources

The usage of resources on the file system can be configured during the installation of MariaDB. This is part of the normal configuration as described in the “my.cnf” configuration file of MariaDB.

6.3.16.7 References

N/A

6.3.16.8 Processing

As soon as a data has to be stored or retrieved in the DB, the process invokes a DAO object in order to perform CRUD operations against the DB. The DAO objects are provided by MyBatis, depending on the given configuration. After the client has performed his operations MyBatis will take care of the transaction management (i.e. commit or rollback), so that the data in the DB will be consistent.

6.3.16.9 Data

MariaDB has been chosen as the DBMS provider. In order to provide a transactional context the EDDS installation of MariaDB will use InnoDB.

The logical design of the EDDS database contains 9 tables and 1 view:

- ACKNOWLEDGEMENT – this table contains the acknowledgement of a specific request (detailed information as to the outcome of a request)
- REQUEST – this table contains data concerning a specific request;
- STATE – this table contains all the possible states of a request;

- REQUESTS_WITH_JOB_ID – this view shows the request table and the .scheduled_job_ids table as one table;
- RESPONSE_FILE – the table contains the response files related to a request.
- SCHEDULED_JOB_IDS – this table contains a list of unique IDs for each scheduled request. Requests with the same scheduled job ID were created from the same original request;
- SCHEDULED_JOB_MAPPINGS – this table contains the mapping of a request to a scheduled job;
- FARC_SUBSCRIPTION – this table contains the currently active FARC subscriptions;
- EDDS_WEB_SERVER_LOGS – this table contains all the Log4j log messages generated by the Web Server;
- EDDS_SERVER_LOGS – this table contains all the Log4j log messages generated by the EDDS Server;
- DELIVERY_MANAGER_LOGS – this table contains all the Log4j log messages generated by the EDDS Delivery Manager.

In addition, there are 11 tables used by the Quartz Scheduler for scheduling jobs.

The following pictures represent the database logical view:

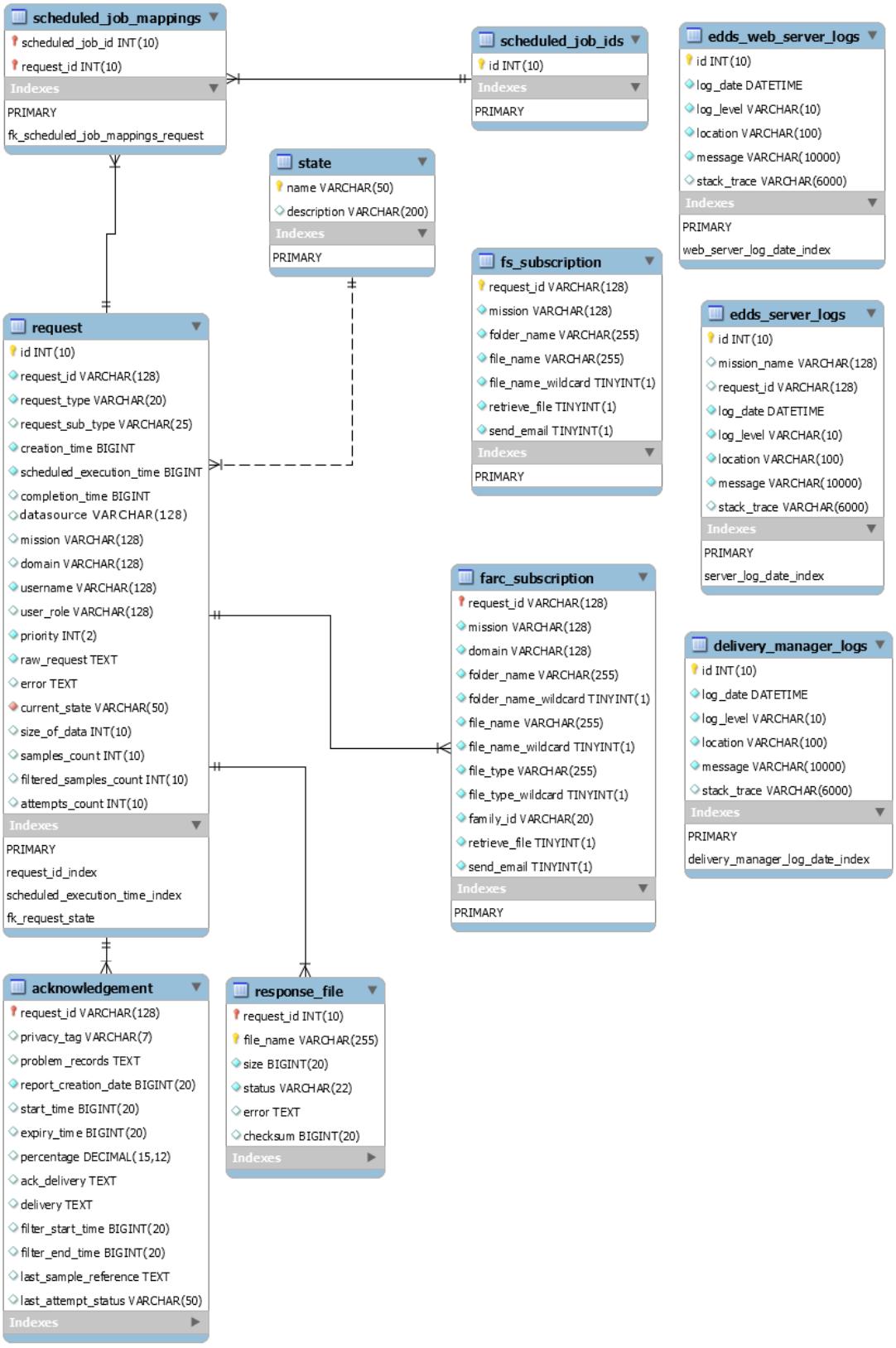

Figure 38 - database ER diagram

The EDDS DB is configured as a separated independent and self-contained database.

6.3.17 EDDS File System Index

6.3.17.1 Type

The component is implemented as a Java Package (esa.egos.edds.provider.fs.index).

6.3.17.2 Purpose

The File System Index provides an always up to date representation of a specific folder and its subfolders. The Index can be used to search for files or folders in the specified folder faster than would be possible by going to the actual file system for each query.

6.3.17.3 Function

The Index is rebuilt on each EDDS Server startup (because otherwise changes to the file system made during the downtime of EDDS Server would be lost). The configured folder and all of its subfolders will have a Java Watcher Service registered to them. The Watcher Service notifications will be used to keep the index up to date and will also be broadcasted to anyone registered as a listener (e.g. useful for subscriptions).

6.3.17.4 Subordinates

There are no subordinates of this component at the architectural level.

6.3.17.5 Dependencies

This component depends upon:

- Apache Lucene;

6.3.17.6 Interfaces

This component provides the following interfaces:

- Index File Helper – an interface for querying the index for files and/or folders
- File System Updates – an interface for subscribing to notifications for changes in the file system

6.3.17.7 Resources

Disk space – the index is stored on the local file system

RAM, CPU – a daemon thread that keeps the index up to date will be running for the entire lifetime of EDDS Server

6.3.17.8 References

N/A

6.3.17.9 Processing

N/A

6.3.17.10 Data

N/A

7. Software Components Detailed Design

7.1 Database Design

The implementation of the Database Manager is done through the ORM framework MyBatis. This tool maps the tables into objects and takes care of the storage of the Java object into the DB tables. This way there is a 1-1 relationship between the DB tables and the corresponding Java models.

Other helper interfaces are generated by MyBatis and can be used to generate *criteria-like* objects during the operation against the DB. The helper interfaces for the log, request and FARC Subscription tables. are created by the MyBatis Spring library at runtime, when the mapper beans are created. These beans can then be used to read/store data into the DB. Also here we will have a 1-1 relationship between mapper interfaces and database tables.
In order to facilitate the access to the DB and also to manage eventual database transactions a façade has been introduced: the DaoAccess. This façade provides the user functional methods. Its initialization is done via another utility class that read the provided configuration and return an instance of the façade.

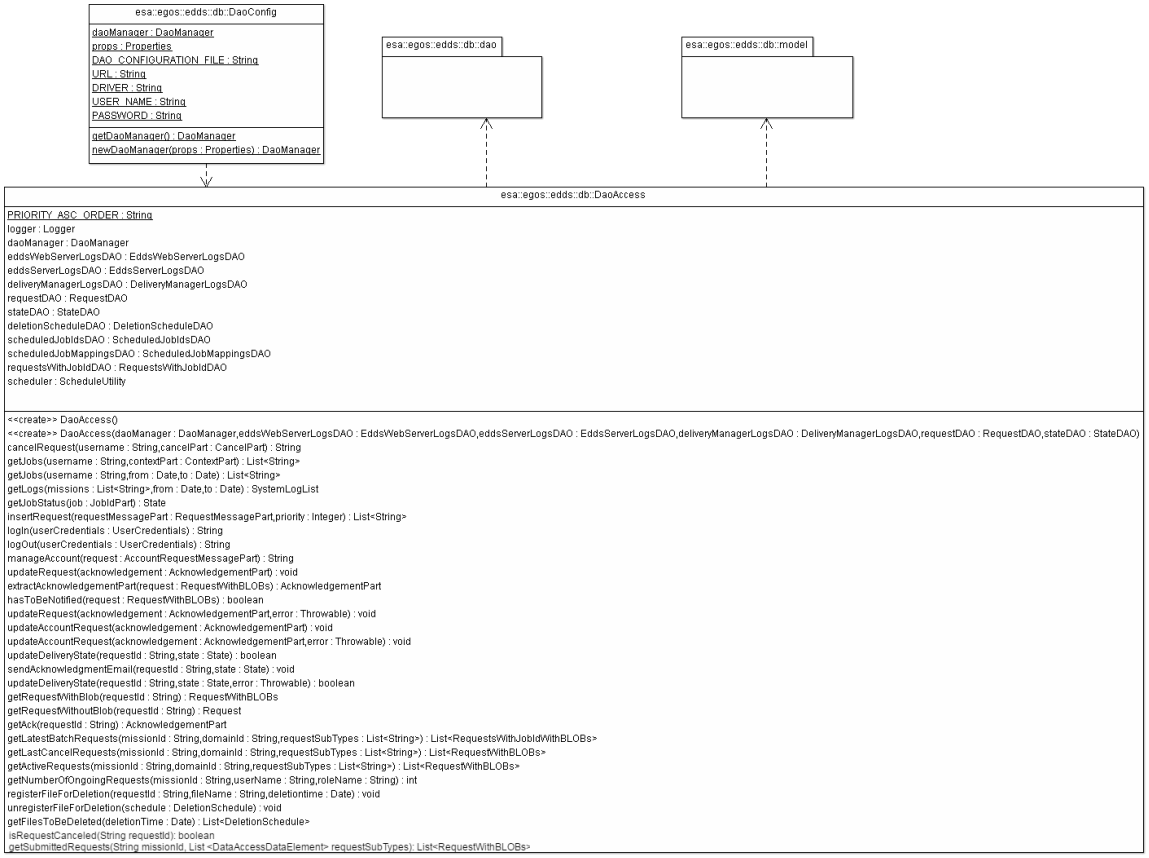

Figure 39 - database facade

7.2 Software Detailed Design

The detailed design for the software can be obtained from the JavaDoc. To create the JavaDoc from the source code, change into the directory where you have extracted the source code from the installation and then change into the “edds” directory where the file “edds-build.properties” exists. Execute the following command:

`mvn clean install javadoc:aggregate-jar -Dadditionalparam=-Xdoclint:none -DskipTest=true`If the command fails, you need to execute a `mvn install` first and then execute the above command again. The JavaDoc can then be viewed in a web browser by opening the file “index.html” in the generated JavaDoc directory.

Appendix A Chain of Responsibility Pattern

The Chain of Responsibility pattern uses a chain of objects to handle a request, which is typically an event. Objects in the chain forward the request along the chain until one of the objects handles the event. Processing stops after an event is handled.

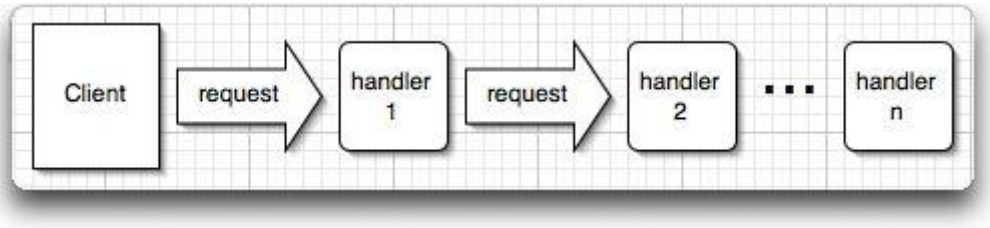

Figure 40 CoR Pattern

