European Component Oriented Architecture (ECOA®) Collaboration Programme: Guidance Document: Data Servers

Date: June 2016
Revised: October 2017

Prepared by
BAE Systems (Operations) Limited
Electronic Systems (UK)

Contents

1	Scope	1
2	Introduction	1
3	Abbreviations	2
4	Definitions	2
5	References	3
6	Data Server Scenarios	4
6.1	Scenario 1: Mission Planning	5
6.2	Scenario 2: Sensor Data Fusion	7
6.3	Scenario 3: Mission Management System	12
6.4	Scenario 4: Demonstrating Large Data Sets	17
7	Design Considerations	29
7.1	Pushmi Pullyu	29
7.2	Operation Choice	31
7.3	Data Encoding	32
8	ECOA Data Server Designs	34
8.1	Query-based Data Servers	34
8.2	Rapid-Access (Indexed) Data Servers	44
8.3	File-based Data Servers	53
8.4	Web Servers	61
9	ECOA Data Server Demonstration	69
9.1	The Demonstration Mission System Oriented ASCs	69
9.2	Build and Execution	71
9.3	Warranty	71

Figures

Figure 1 Data Server Scenarios 4
Figure 2 Scenario 1: Mission Planning 5
Figure 3 JDL Data Fusion Model 8
Figure 4 JDL Data Fusion Management 9
Figure 5 Combat UAV Mission Data Management Architecture 13
Figure 6 Reconnaissance UAV Mission Data Management Architecture 14
Figure 7 Distributed Data Management (DDM) Architecture 15
Figure 8 Scenario 4: Demonstrating Large Data Sets – Demonstration Scenarios 18
Figure 9 Scenario 4.1: HUMS Data Processing 19
Figure 10 Scenario 4.2: HUMS Data Recording 20
Figure 11 Scenario 4.3: Weather Data 21
Figure 12 Scenario 4.4: Vehicle State 22
Figure 13 Digital Map Tiling 24
Figure 14 Scenario 4.5: Digital Map Tiles 25
Figure 15 Scenario 4.6: Target Data 26
Figure 16 Scenario 4.7: Tactical Data 27
Figure 17 Push Model Behaviour 29
Figure 18 Pull Model Behaviour 30
Figure 19 ECOA Versioned Data - Push/Pull Implementations 32
Figure 20 Example Base64 Encoded Value 32
Figure 21 Typical SQL Client-Data Server Configuration 35
Figure 22 ECOA SQL Client-Data Server Configuration 36
Figure 23 ECOA SQL Service Definition (as a UML Interface Class) 37
Figure 24 sqlServer ASC Design (as UML Class Diagram) 41
Figure 25 Typical Key-Value Data Server Configuration 44
Figure 26 ECOA Key-Value Data Server Configuration 45
Figure 27 ECOA Key-Value Data Server Configuration (Multiple Hosts) 45
Figure 28 ECOA Key-Value Service Definition (as a UML Interface Class) 46
Figure 29 dbmServer ASC Design (as UML Class Diagram) 50
Figure 30 Typical File-based Data Server Configuration 53
Figure 31 ECOA File-based Data Server Configuration 54
Figure 32 ECOA File-based Data Server Configuration (Multiple Hosts) 54
Figure 33 ECOA File IO Service Definition (as a UML Interface Class) 55
Figure 34 fileServer ASC Design (as UML Class Diagram) 58
Figure 35 Typical Web Service Data Server Configuration 62
Figure 36 ECOA-ized Web Service Data Server Configuration 63

Figure 37 ECOA (Weather) Web Server Access Service Definition (as a UML Interface Class) 64
Figure 38 metServer ASC Design (as UML Class Diagram) 66
Figure 39 ECOA Data Servers Demonstration Assembly 69

Tables

Table 1 Demonstrating Large Data Sets (Examples) 17

0 Executive Summary

'*Data server*' is the phrase used to describe computer software and hardware (a database platform) that delivers database services. Also called a '*database server*' it may also perform tasks such as data analysis, storage, data manipulation, archiving, and other tasks using a client-server architecture.

A '*Data server*' is a stand-alone computer or application in a local area network that holds and manages the database. It implies that database management functions, such as locating the actual record being requested, are performed in the server computer/application.

This document describes a number of ways how to interface an ECOA® software system to a data server. That is, how to create ECOA Application Software Components (ASCs) that provide ECOA Services that provide data server functionality.

It is not in any way a "normative", part of the ECOA, or even definitive. The discussions here are purely examples of how ECOA ASC interfaces can be designed and implemented for the different data server types addressed.

After describing example scenarios calling for use of data servers in an ECOA mission system context, example (basic) ECOA ASC interfaces are addressed for several types of data server.

The document concludes with a description of an implemented software system able to demonstrate the concepts and example solutions discussed.

1 Scope

This document is intended to provide an overview of implementing data server mechanisms within an ECOA software system, particularly in respect of the client-server model, and to provide some guidance on the design issues and choices that may arise. The document and its content are not intended to be normative nor in any way regulatory. The designs and implementations presented are purely illustrational examples.

Section 2 gives a brief introduction to the data server topic.

Section 3 expands abbreviations used in this report.

Section 4 provides definitions for the key terms used in this report.

Section 5 lists key documents referenced by this report.

Section 6 discusses a number of mission system scenarios requiring data servers where ECOA systems may be relevant. These scenarios set the context in which the subject of data servers is then explored.

Section 7 discusses a number of significant design considerations that must be addressed when designing and implementing data servers, particularly with the ECOA.

Section 8 describes example designs for ECOA ASC interfaces to a number of data server types.

Section 9 describes demonstration implementations of the described ASC designs.

Section 10 records the Intellectual Property ownership of the material in this document.

2 Introduction

'*Data server*' is the phrase used to describe computer software and hardware (a database platform) that delivers database services. Also called a '*database server*' it may also perform tasks such as data analysis, storage, data manipulation, archiving, and other tasks using a client-server architecture (ref. *[Howe]*).

A '*Data server*' is a stand-alone computer or application in a local area network that holds and manages the database. It implies that database management functions, such as locating the actual record being requested, are performed in the server computer/application.

It is not our purpose here to elaborate on the details of what a data server may or may not do. There are plenty of sources for that. Nor will we describe how to create or employ a data server. All we are interested in here is how to interface an ECOA software system to a data server. That is, how to create ECOA Application Software Components (ASCs) that provide ECOA Services that provide data server functionality.

Nor is this document intended to be in any way "normative", part of the ECOA, or even definitive or complete. The discussions here are purely examples of how ECOA ASC interfaces can be designed and implemented for the different data server types addressed, in the hope that they might provide some useful guidance. At best, we might hope that these examples might form the basis of fully designed ASCs for inclusion in the ECOA Vault.

We will address ECOA ASC interfaces for a number of different types of data server, starting with a SQL-driven data management system, then a Key-Value pair data server, a simple file access interface for flexible persistent data storage, and finally a web-service interface.

3 Abbreviations

API	Application Programming Interface
(A)MMS	(Advanced) Mission Management System
ASAAC	Allied Standards Avionics Architecture Council
ASC	Application Software Component
COTS	Commercial Off-The-Shelf
DAS	Defensive Aids System
DGA	Direction Générale de l'Armement
Dstl	Defence Science and Technology Laboratory
ECOA	European Component Oriented Architecture
ELINT	Electronically Obtained Intelligence
EO/IR	Electro-Optic / Infra-Red
HUMINT	Human (derived) Intelligence
HUMS	Health and Usage Monitoring (or Management) System
IP	Intellectual Property
MOD	Ministry of Defence
OS	Operating System
PC	Personal Computer
POSIX	Portable Operating System Interface
QoS	Quality of Service
SAR	Synthetic Aperture Radar
SitRep	Situation Report
SOA	Service-oriented Architecture
UAV	Unmanned Aerial vehicle
UML	Unified Modeling Language
WASM	Weapon Aiming and Stores Management
XML	eXtensible Markup Language
XSD	XML Schema Definition

4 Definitions

For the purpose of this document, the definitions given in the ECOA Architecture Specification *(ref. [AS])* Part 2 and those given below apply.

Term	***Definition***
(currently none)	

5 References

AS	European Component Oriented Architecture (ECOA) Collaboration Programme: Architecture Specification (Parts 1 to 11) "ECOA" is a registered trade mark.
Howe	Dictionary.com, "database server," in *The Free On-line Dictionary of Computing*. Source location: *Denis Howe. http://dictionary.reference.com/browse/database server.*
SQL	Structured Query Language. *ISO/IEC 9075*
ODBC	Open Database Connectivity. *SQL Access Group, 1992*
JDL	Functional Description of the Data Fusion Process, technical report for the Office of Naval Technology Data Fusion Development Strategy O. Kessler, et. al. Naval Air Development Center, Nov. 1991
DFUS	Data Fuser for an Advanced Mission Management System (AMMS) *FUTURE TECHNOLOGIES FOR COMBAT AIRCRAFT:* *AVIONICS TECHNOLOGIES: Integrated Modular Avionics* *Identify Characteristics of Applications to be Modelled* BAE Systems & THALES Airborne Systems, 2001
MIFS	Mission Information-Flow Analysis and Computing Resource Allocation study for: *System Concepts for Mission Management* Smiths Industries Aerospace & Marconi Electronic Systems, 1999
CODE	The official ECOA® website: *http://www.ecoa.technology/* "ECOA" is a registered trade mark
dbm	dbm (the UNIX dbm Library) *https://en.wikipedia.org/wiki/Dbm*
	GDBM (GNU dbm) *http://www.gnu.org.ua/software/gdbm/*
SOAP	SOAP Version 1.2 W2C® Recommendation (Second Edition) 2007 *http://www.w3.org/TR/soap/*
KLV	Data Encoding Protocol Using Key-Length Value SMPTE 336M-2007
BER	Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) ITU-T X.690, 07/2002
B64	The Base16, Base32, and Base64 Data Encodings. RFC 4648, IETF. October 2006.
XML	Extensible Markup Language (XML) 1.0 W3C® Recommendation 26 November 2008 *http://www.w3.org/TR/2008/REC-xml-20081126/*

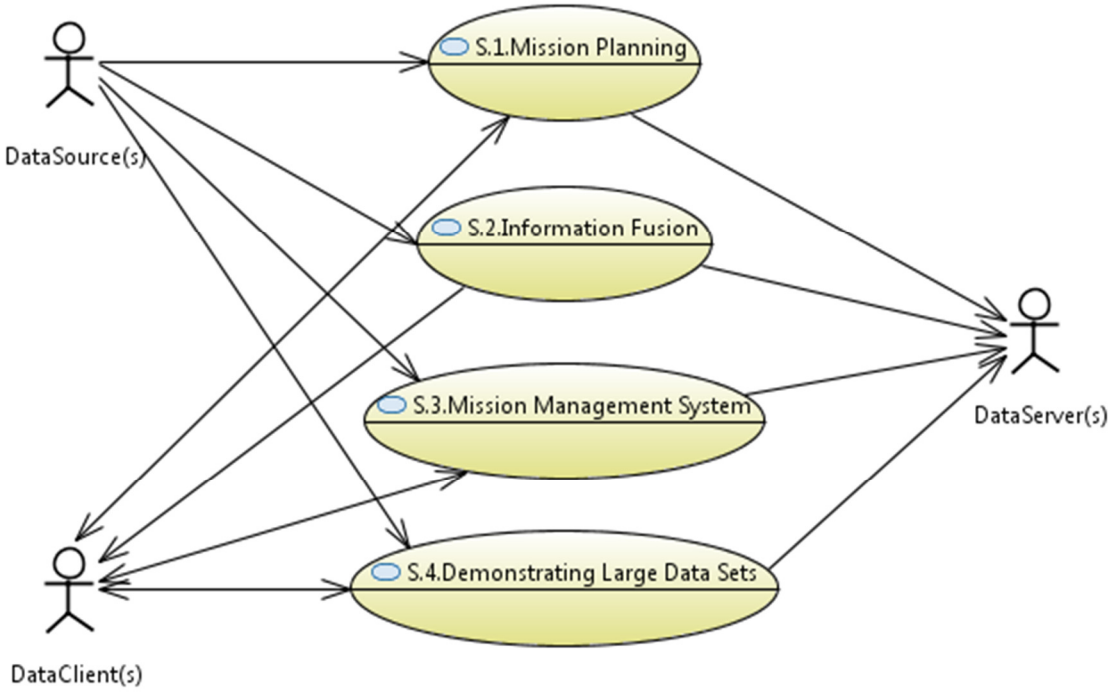
6 Data Server Scenarios

A number of scenarios have been identified for Data Servers within an ECOA system context, three specifically applying to Mission Systems. The intention here is not to *fully* describe each scenario and the data and processes therein, but rather to present them in overview as examples of where and why data server principles are likely to be invoked.

The fourth scenario is one applicable to demonstrating some of the concepts likely to be required in any attempt to fulfil the first three.

These scenarios are illustrated (in UML) and described below. Each scenario is associated with one or more *DataSources* and/or *DataClients*, and also with one or more *DataServers*. It is these *DataServers,* and the technology they require, that we seek to identify for each scenario.

Figure 1 Data Server Scenarios



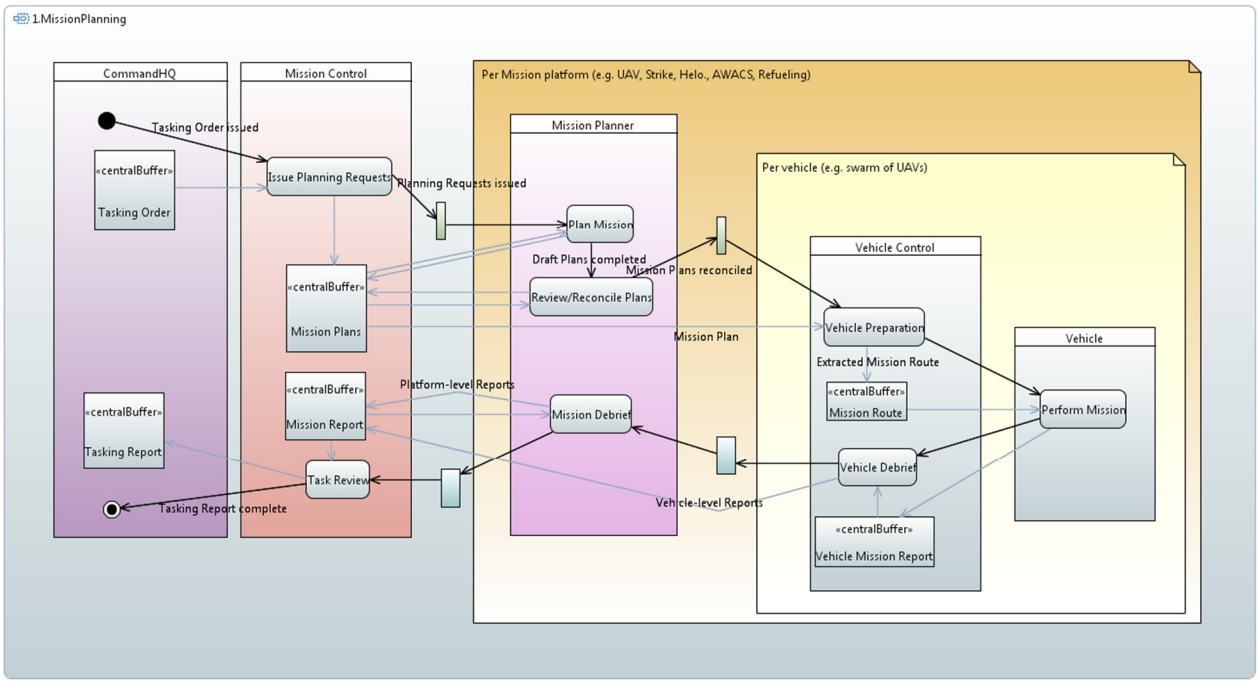
6.1 Scenario 1: Mission Planning

6.1.1 Description:

The planning of complex multi-agency missions involves information and data garnered from and distributed by many independent providers and users. Sources would include such agencies as weather bureaux, cartographic publications, geographic feature and culture surveys, aeronautical/navigational regulatory authorities, as well as tactical and battlespace information sources. Many of these agencies may also accept revisions as a mission progresses, particularly for high rate-of-change information such as tactical and battlespace information, which may include local (to the mission) weather updates.

The following description presents a hypothetical, *highly* simplified, view of the specifically Mission Planning information flow in such a system, highlighting where data server principles are likely to apply. The scenario is presented, in overview, as a UML Activity Diagram (Figure 2).

Figure 2 Scenario 1: Mission Planning



From an initial Tasking Order, Planning Requests are issued by a “Mission Control” centre to a number of cooperating Platform (e.g. UAV, Refuelling, AWACS) Control centres (e.g. squadrons). Mission Planners for each Platform set about planning the Mission for their Platform (type) exchanging and reconciling their plans with the other Mission Planners to formulate a coherent overall plan to attain the Task objectives.

Detailed Mission Plans are then handed to Vehicle Control specialists who would extract specific Mission Routing and Goal information at the individual vehicle level.

After the Mission, individual vehicle reports are flowed back through debriefings to Platform Control (e.g. squadron) level, and coordinated to “Mission Control” level, and ultimately to a Tasking level report.

6.1.2 Pre-Conditions:

A threat to own or friendly forces, or non-combatants, exists.

6.1.3 Initiating Event:

A Tasking Order is issued from Command HQ.

6.1.4 Primary Activities:

i. “Mission Control” retrieves the Tasking Order.
ii. “Mission Control” analyses the Tasking Order and identifies Platforms required to fulfil.
iii. Initial (outline) Plans are proposed, and Planning requests issued.
iv. Outline Plans are retrieved by per-Platform Mission Planners and expanded per the Platform (type).
v. Detailed (expanded) Mission Plans are reviewed and reconciled with the cooperating per-Platform Mission Planners.
vi. Once all detailed Mission Plans are reconciled, “Vehicle Control” specialists prepare vehicle-level Mission Routes for each vehicle.
vii. Mission Routes are loaded to the vehicles.
viii. The Mission is performed.
ix. Vehicle Mission Reports are analysed at Vehicle Debrief, and composed into Mission report(s) for Mission Debrief.
x. Task Review in turn composes a Tasking-level report across Platforms, as the response to the originating Tasking Order.

6.1.5 Post-Conditions:

Threat quelled, quashed, or crushed.

6.1.6 Potential Data Servers:

i. Command HQ: Tasking
Centralized; Remote; Very Secure;
Tasking Orders : filed by Command HQ; retrieved by “Mission Control”
Tasking Reports : filed by “Mission Control”; retrieved by Command HQ

ii. “Mission Control” Information Management
Remote; Distributed; Secure server(s); Secure web-based access;
Outline Mission Plans
Detailed (draft, reviewed, reconciled) per-Platform Mission Plans
Mission Reports

iii. (Squadron) Information Management
Local (to the squadron); (local) Network access;
per-Vehicle Mission Route extractions
Vehicle Mission Reports

6.2 Scenario 2: Sensor Data Fusion

6.2.1 Description:

Sensor Data Fusion is a key part of any Advanced Mission management System (AMMS). The following is a brief example of an AMMS Data Fusion architecture based on the JDL model (ref. *[DFUS]*).

The Joint Directors of Laboratories (JDL) (ref. *[JDL]*) Data Fusion Group's Data Fusion Model is an ordered, multi-layered, architecture of the data and processing to achieve data fusion. It presents an approach for categorising data fusion related functions. It is an industry standard that was developed between 1987 and 1991. The model was developed by the International Society for Information Fusion (ISIF) and Fusion Information Analysis Centre (FUSIAC-US Government) and has been used by a number of major organisations. The JDL distinguishes fusion "levels", providing a useful distinction among data fusion processes that relate to the refinement of "objects", "situations", "threats", and "processes":

- **Level 0** – Sub-object Data Assessment: estimation and prediction of signal object observable states on the basis of pixel/signal level data and characterisation.
- **Level 1** – Object Assessment: estimation and prediction of entry states on the basis of observation-to-track association, continuous state estimation (e.g. kinematics) and discrete state estimation (e.g. target type and ID).
- **Level 2** – Situation Assessment: estimation and prediction of relations among entities, to include force structure and cross force relations, communications and perceptual influences, physical context, etc.
- **Level 3** – Impact Assessment: estimation and prediction of effects on situations of planned or estimated/predicted actions by the participants; to include interactions between action plans of multiple players (e.g. assessing susceptibilities and vulnerabilities to estimated/predicted threat actions given one's own planned actions).
- **Level 4** – Process Refinement (an element of Fusion-Resource Management): adaptive data acquisition and processing to support mission objectives.

Figure 3 shows only the Level 0 to 3 activities, representing a forward path of data fusion, from raw measurements and signals on the left, through front-end (Level 0) fusion to extract features of interest (targets, locations etc.) in a manner specific to the data source. In some cases this leads to further Level 0 fusion (e.g. imagery data fusing from multiple sources).

Level 1 fusion is then applied, fusing in electronic intelligence received via the communications system (e.g. from AWACS, ISTAR etc.) and also information and observations directly from the aircrew.

Level 2 fusion follows, applying the gained information to form a situational assessment, and then Level 3 fusion to form an impact assessment.

Figure 4 shows the Level 4 activities, whereby each data fusion node (of Figure 3) is closely coupled to a fusion-resource management node. Command and control information is flowed in reverse-level-order (i.e. starting from the Level 3 node and propagating through to the Level 0 nodes). This is either as mission mode information is applied, changing the fusion properties and priorities, or as each node determines requirements and priorities of itself and its supplying nodes, e.g. as a result of fault conditions occurring or to change sensor resolution or field-of-view. Command and control of sensors is referred to the Sensor Manager function(s) of the AMMS.

Figure 3 JDL Data Fusion Model

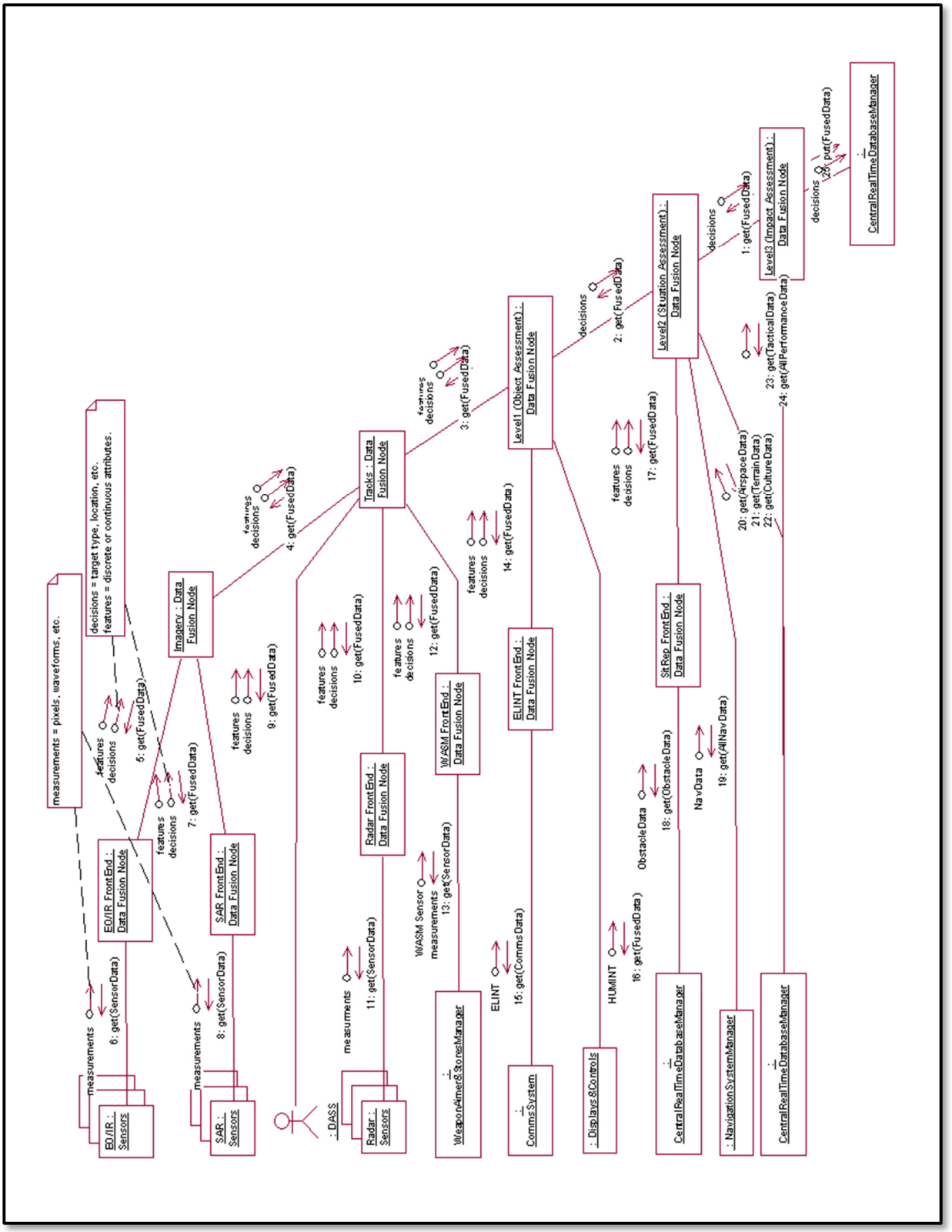
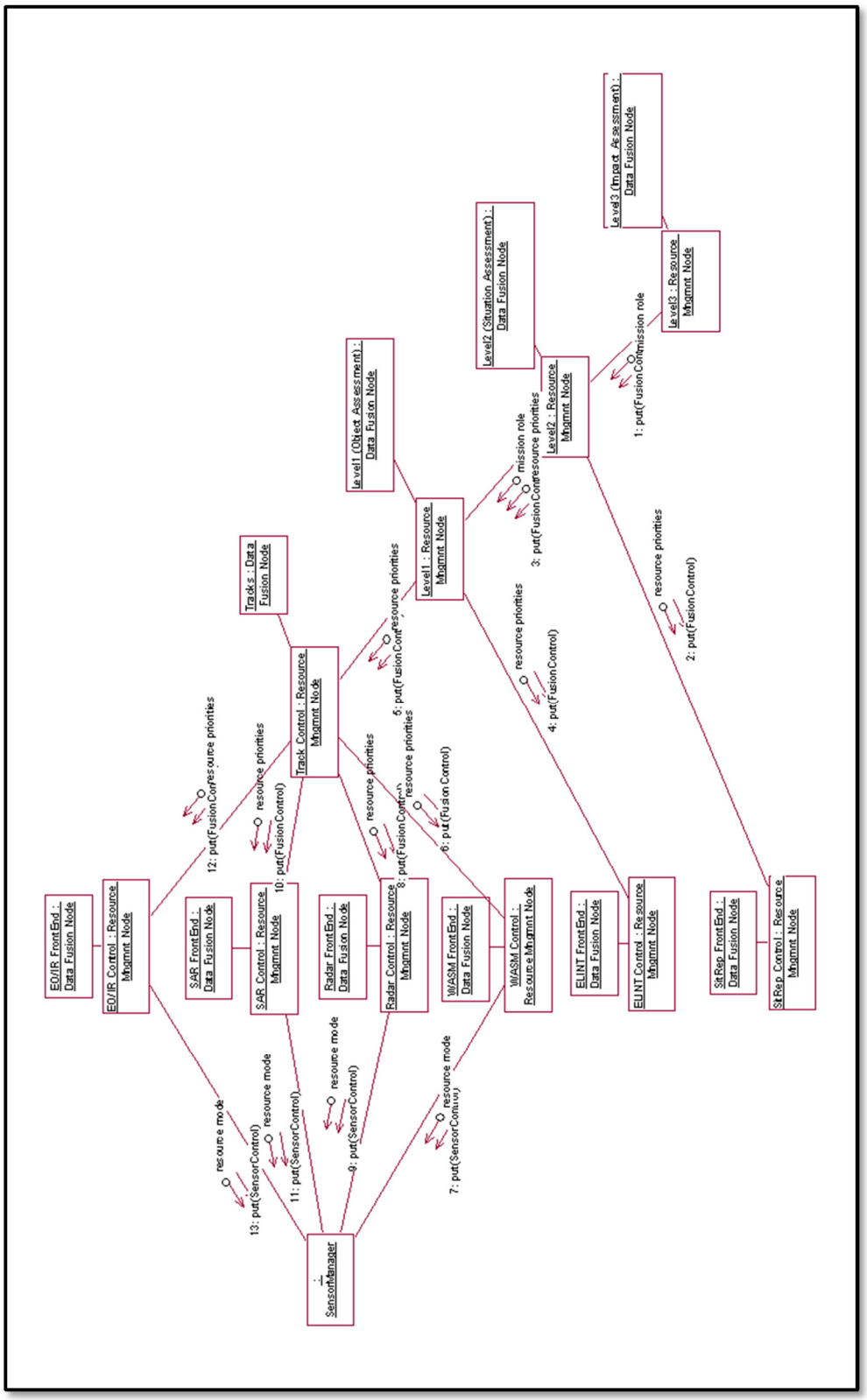


Figure 4 JDL Data Fusion Management



6.2.2 Pre-Conditions:

A battlespace exists, and the vehicle is part of it.

6.2.3 Initiating Event:

An obscured target is sought in a confused battlespace.

6.2.4 Primary Activities:

i. A *Fusion Control* request is sent down from Level 3 *Resource Management Node*.
ii. At each lower level, the *Resource Management Node* determines whether it needs to direct resources at its level.
iii. At Level 0, the EO/IR and Radar *Resource Management Nodes* send down *Sensor Control* requests to enable and aim their respective sensors.
iv. *EO/IR Sensors* are deployed to image the target (area), etc.
v. *SAR* is deployed to image the obscured target (area), etc.
vi. Sensor data is analysed by the front-end EO/IR and SAR *Data Fusion Nodes*, which extract features, tracks, etc. which are fed up to the Imagery *Data Fusion Node*. This brings together and fuses all relevant Sensor level data.
vii. At Level 0, features and threat tracks from Sensors, confirmed by DAS, wide area Radar etc. sources, are assessed and promoted to Level 1.
viii. At Levels 1 and 2 further analysis and confirmation is performed using the resources available at that level. Threat and relevance data, and reaction decisions emerge.
ix. At Level 3 fused data is committed/updated to the data server.

6.2.5 Post-Conditions:

A detailed, logical, picture of the target within the battlespace has emerged.

6.2.6 Potential Data Servers:

i. Central Real Time Database
Centralized; Real-time; Remote (from individual MMS functions);
Multi-aspect battlespace "picture"
Fused battlespace data
Confirmed battlespace reaction decisions
ii. Level 1 to 3 Resource Management
Centralized; Real-time;
Additional source availability, capabilities, and modes
iii. Level 1 to 3 Data Fusion
Centralized; Real-time;
Feature and decision filtering rules
Decision extraction and confirmation rules
Confirmed reaction decisions
iv. Logical Source (WASM, ELINT, HUMINT, etc.) Resource Management
Centralized; Real-time;
Source availability, capabilities, and modes
v. Logical Source Data Fusion
Local; Real-time;
Data filtering rules
Feature types
Feature extraction and confirmation rules
Fused features and decisions

vi. Sensor (EO/IR, SAR, Radar etc.) Resource Management
Local; *Real-time; Rapid access and response;*
Sensor capabilities and modes
Sensor control limits

vii. Sensor Data Fusion
Local; Real-time; Rapid access and response;
Fused sensor images
Image filtering rules
Feature types
Feature extraction rules
Extracted features and decisions

6.3 Scenario 3: Mission Management System

6.3.1 Description:

Studies for MoD UK into advanced UAV Mission Management Systems (MMS) explored the information flows (ref. *[MIFS]*) for a number of combinations of vehicle platform type in order to achieve likely current and future mission types, leading to mission data management requirements.

For different vehicle platform roles (such as combat or reconnaissance) different mixes of vehicle systems and specific data are required, and the studies therefore proposed a mission data sharing and distribution concept based on centralised data management. This allowed for flexible pick-and-mix provision of vehicle platform capability by limiting or restricting direct interconnections between vehicle capability providers. Thus rather than obtain, say, air vehicle state data from the Navigation System, a Flight Control System would obtain it from a centralised data management source (where the Navigation System would deposit the data). The Navigation and the Flight Control Systems are therefore connected only to the data management, not to each other, increasing the possibilities for individual System reuse and incremental enhancement greatly.

Two instances of the application of such an MMS architecture are illustrated in Figure 5 and Figure 6, for a combat UAV and a reconnaissance UAV respectively, illustrating the primary interactions between MMS data management and the relevant external data agencies (such as an aeronautical regulatory authority or a weather data supplier) or vehicle systems. For example, weather data from an external agency would be processed (in the case of a UAV as illustrated) through the vehicle ground *Control Facility*, and if relevant to the vehicle mission or operation would update the mission (route) plan (via the *MissionPlanDB* interaction). This may then in turn impinge on the operations of the Defensive Aids System (*DAS*) as it adjusts to watch for threats along the updated route.

The study specifically identified a number of data management interactions that would be required in an MMS context, each described by the exchange of mission relevant data between external sources and internal vehicle functions:

- ***TerrainDB*** geographic and cultural (e.g. land usage);
- ***AeronauticalDB*** aeronautical regulatory, restrictions, and hazards;
- ***ConfigurationDB*** the vehicle and its equipment fit;
- ***MissionPlanDB*** the vehicle's Mission Plan, including the route to take, and the threats and targets along that route;
- ***TransientDB*** short lifespan vehicle and mission information e.g. for relay back to the ground *ControlFacility*;
- ***NavigationalDB*** obstructions, no-go areas, flight corridors, etc.;
- ***TacticalDB*** threat and target details;
- ***Video/SensorDB*** threat tracks, visual, IR, and SAR (etc.) imagery;
- ***FusedSensorDataDB*** inferred threat tracks, enhanced imagery, etc.;
- ***CommunicationsDB*** frequencies, IFF codes, etc.

The studies therefore identified the need for a flexible, highly reusable, scalable, and distributable, data management architecture (illustrated in Figure 7), predicating a number of data servers distributed within the mission system computing environment managing long and short lifetime data. The need for a distinction between data having long and short lifetimes was explicitly brought out in the architecture where identified.

This distinction allows the architecture to be implemented using mixed data management technologies, allowing short term data to be rapidly and frequently accessed by using appropriate local, rapid-access, but less capable in terms of search and selection, technologies, whilst then permitting data to be committed to long term, highly capable, data management in a slower timescale more appropriate to complex search, selection, and analysis, data management technologies.

Figure 5 Combat UAV Mission Data Management Architecture

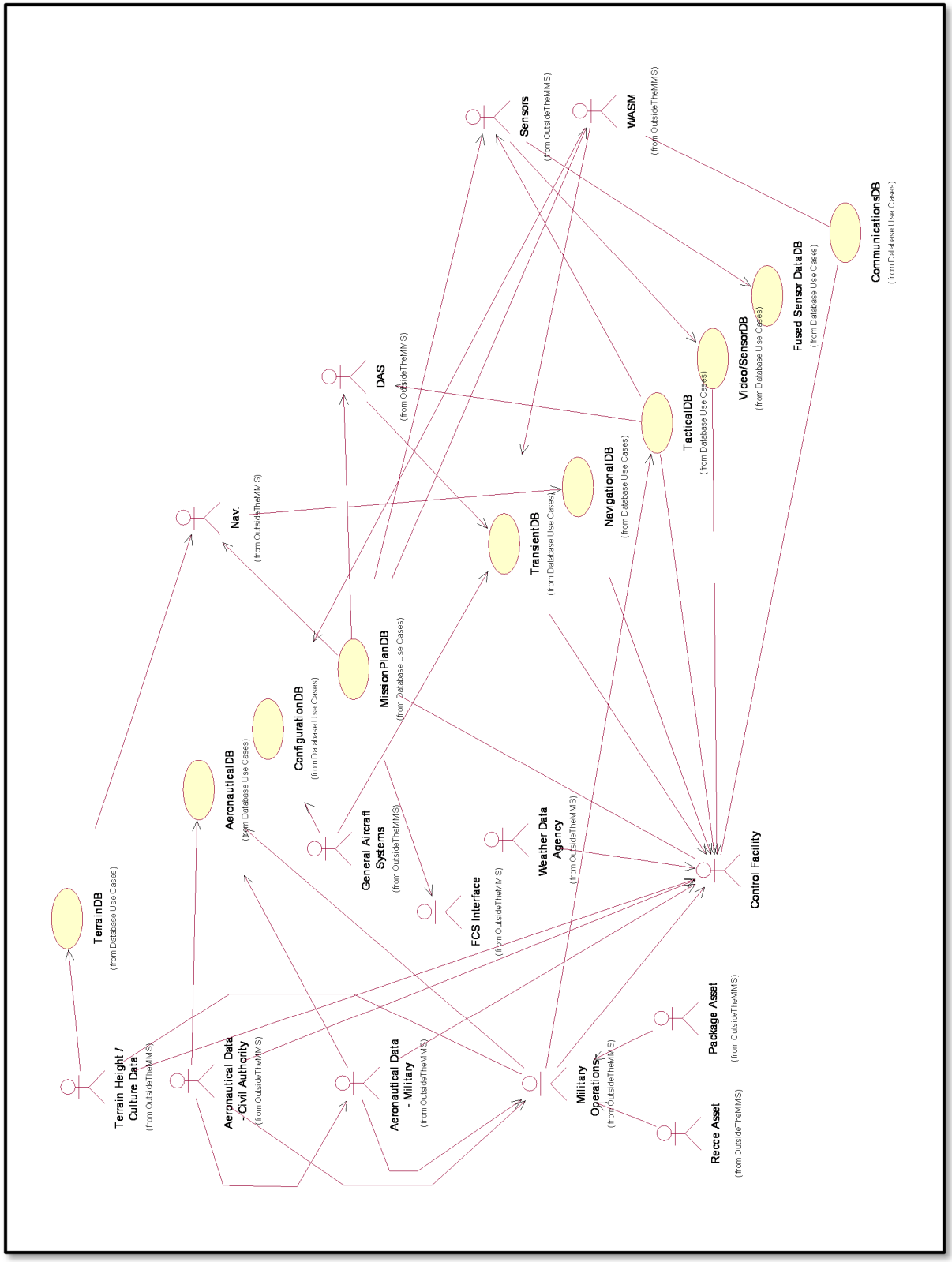
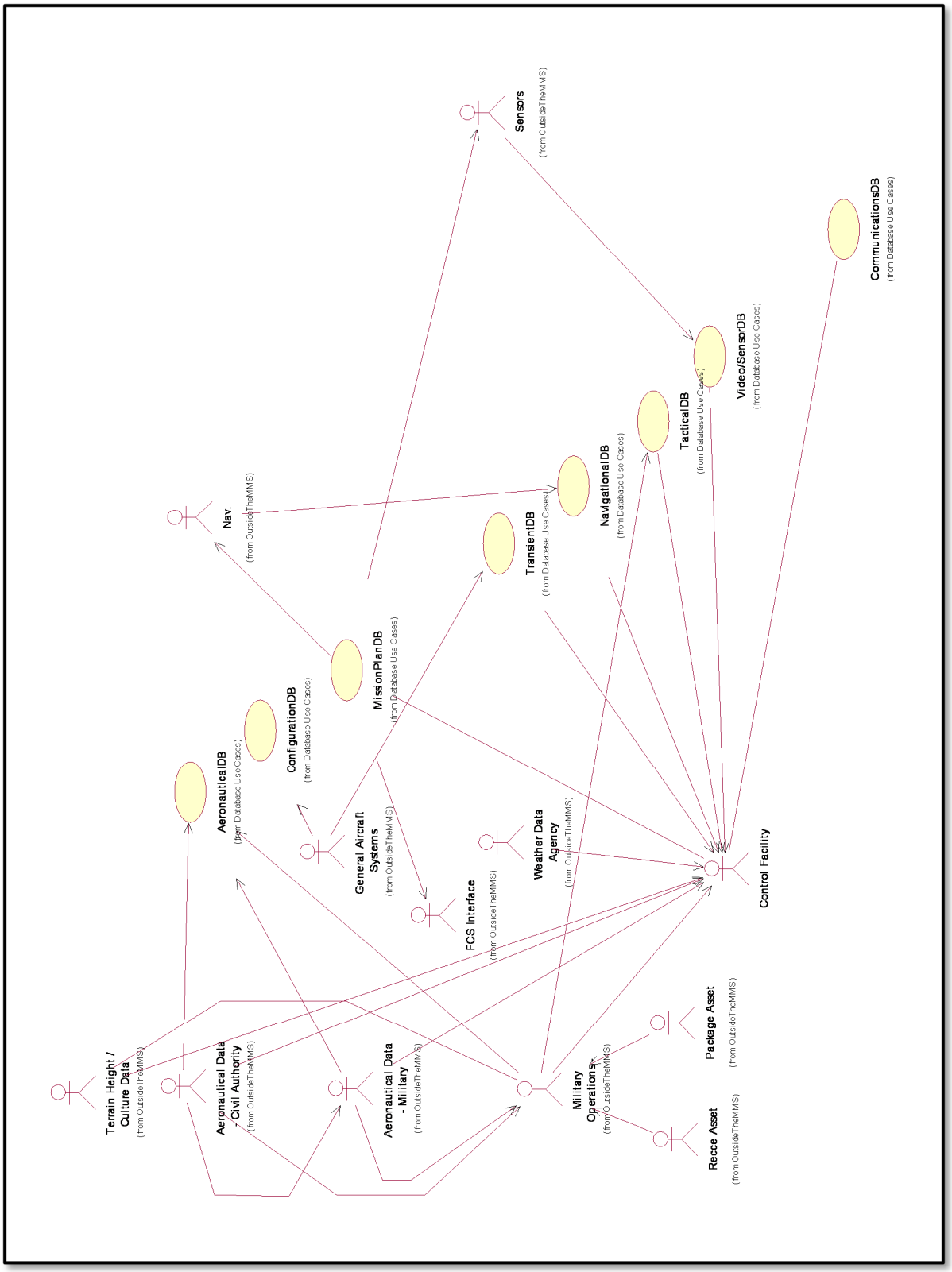


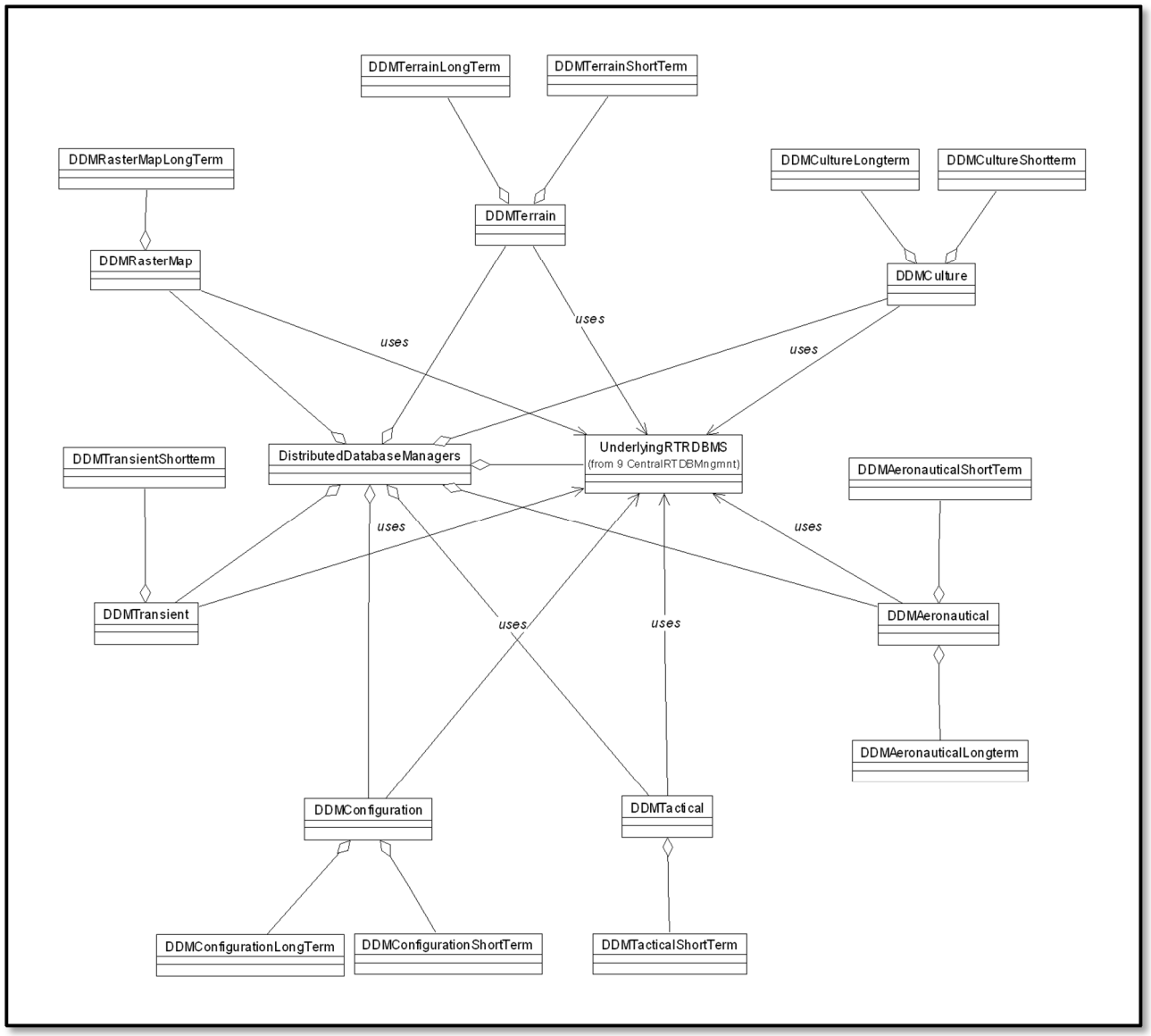
Figure 6 Reconnaissance UAV Mission Data Management Architecture



This Distributed Data Management (DDM) architecture comprises a scalable number of *Distributed Database Managers*, each of which comprises an *UnderlyingRTDBMS (Real-time Database Management System)* of some form.

The *Distributed Database Managers* collectively "contain" the data-specific logical data management functions (such as *DDMAeronautical* or *DDMTransient*) which may manifest in short-term (local, rapid access) and/or long-term (possibly remote, complex search and retrieval) forms, as necessary (such as *DDMAeronauticalShortterm* and *DDMAeronauticalLongterm*).

Figure 7 Distributed Data Management (DDM) Architecture



6.3.2 Pre-Conditions:

An AMMS exists, founded on the principles of the DDM Architecture.

By way of illustration, an example flow of information will be described (briefly), which should be typical of many information flows within an advanced MMS.

6.3.3 Initiating Event:

The *Aeronautical Data – Military* authority issues a hazard area notification update.

6.3.4 Primary Activities:

i. The notification will be passed to the *Military Operations* authority.
ii. The *Military Operations* authority will assess the notification, and flow out to the vehicle *Control Facility* and possibly to the vehicle itself.
iii. When applicable[1], the vehicle's receiving *DDMAeronautical* data management function will assess received data and push to either *DDMAeronauticalShortterm* or *DDMAeronauticalLongterm* store.
iv. The *Control Facility* will assess the notification and (if necessary) issue a mission re-plan order.
v. The revised Mission Plan is pushed out to the vehicle, to the *Nav.* system, *FCS Interface*, and (possibly) the *DAS*, *WASM* and *Sensor* management (for re-programming field-of-regard etc.).

6.3.5 Post-Conditions:

None.

6.3.6 Potential Data Servers:

i. All artefacts prefixed "DDM" in Figure 7
Mix of Centralized and Local
(content as per artefact name).
ii. DAS, FCS, Nav., WASM, Sensor Management
Local; Private; Real-time; Rapid access and response
Mission (Route) Plan, or selected, relevant, subset thereof.

[1] On-board aeronautical data is included as part of the advanced MMS architecture to facilitate on-board re-planning/optimal routing capabilities.

6.4 Scenario 4: Demonstrating Large Data Sets

From the previous Mission System relevant example scenarios, it is possible to identify themes which are worth demonstrating in an ECOA system:

- **(Content) Query-Based Data Management**, accessing (for read and/or write) long lifetime data held in data server permitting complex, content driven, search and retrieval;
- **(Rapid-Access) Indexed Data Management**, accessing (for read and/or write) data held in a data server designed and optimized for rapid access and retrieval;
- **Remote.*v*.Local Data Management**, demonstrating data server access across a network, or local to the client;
- **Secure.*v*.Non-secure Data Management**, demonstrating application of security protocols and methods to protect data and the servers and clients using it.

In addition to these themes, there will undoubtedly be situations where bespoke persistent data storage will be required, as provided for instance by access to a file system. Examples did not emerge from the previous scenarios as they were based on studies explicitly looking for where organized, reusable, data management was appropriate. However, it is recognised that individual capabilities will exist where such organized data management may be an unnecessary overhead. Loading graphic image tiles from a data repository in a digital map capability might be one such case.

Crossing these themes, example managed data sets can be chosen in order to form the basis of demonstration. In Table 1, for example, a number of managed data sets are identified and associated with the theme (or variation of a theme) the data set management falls into.

Table 1 Demonstrating Large Data Sets (Examples)

Theme / Managed Data	*(Rapid-Access) Indexed*	*Query Based*	*Remote (ECOA[1])*	*Remote (Web[2])*	*Local*	*Secure*	*Non-secure*	*Bespoke (File IO)*
HUMS Data Processing	✔		✔				✔	
HUMS Data Recording		✔	✔			✔		
Weather Data		✔	✔	✔		✔		
Vehicle State	✔				✔		✔	
Digital Map Tiles			✔				✔	✔
Target Data	✔		✔			✔		
Tactical Data		✔	✔			✔		

Notes:

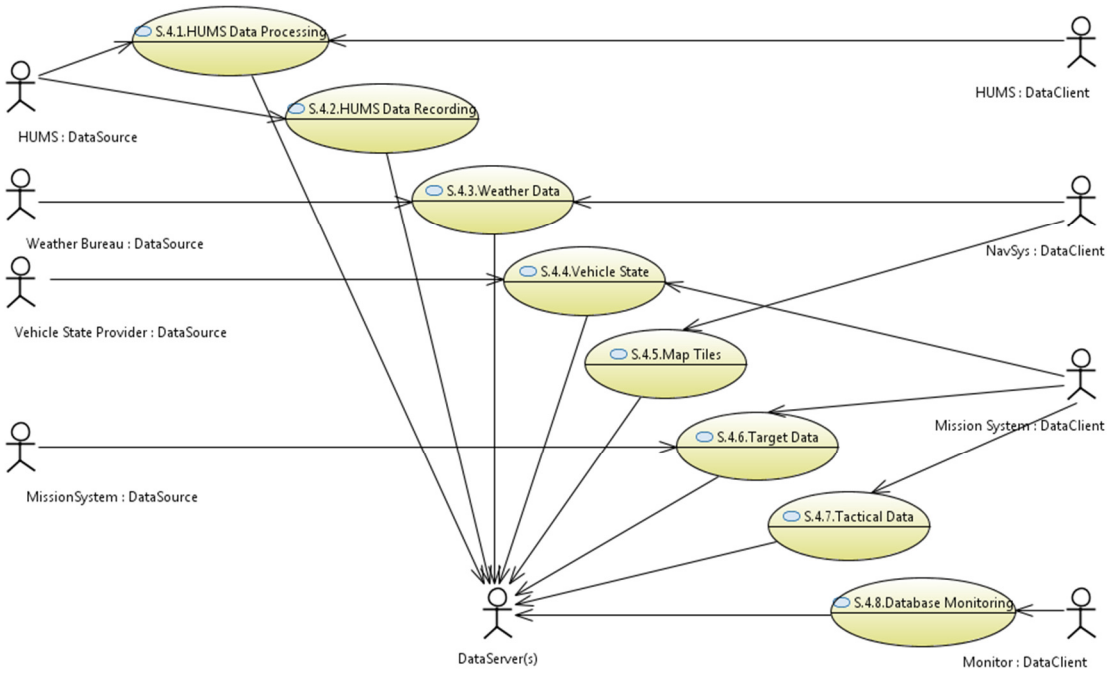
1. This column marks data management examples where the data server can be demonstrated "remote" from the client simply within an ECOA system context, i.e. the client and server are hosted on separate ECOA Computing Platforms, and therefore invoke the ELI.
2. This column marks data management examples where the data server is outside the ECOA system context, specifically, a web service accessed via the public internet.

The data management cases of Table 1 are expanded in the following sub-sections. Each describes a simplified scenario for one of the Managed Data sets of the table, intended to form the basis of an indicative demonstration of the associated themes.

Each scenario is described using a combination of textual description and UML diagrams.

Figure 8 gives an overall view of the demonstration scenarios with particular sub-systems or agencies relevant to each particular scenario, whether as a *DataSource* or as a *DataClient*. Where no *DataSource* is associated with a scenario, it implies that – for the purposes of the scenario – the data is "read only".

Figure 8 Scenario 4: Demonstrating Large Data Sets – Demonstration Scenarios



6.4.1 Scenario 4.1: HUMS Data Processing

6.4.1.1 Description:

Either periodically or on some trigger, the HUMS will scan all monitored sub-systems and collect together their Health and Usage data.

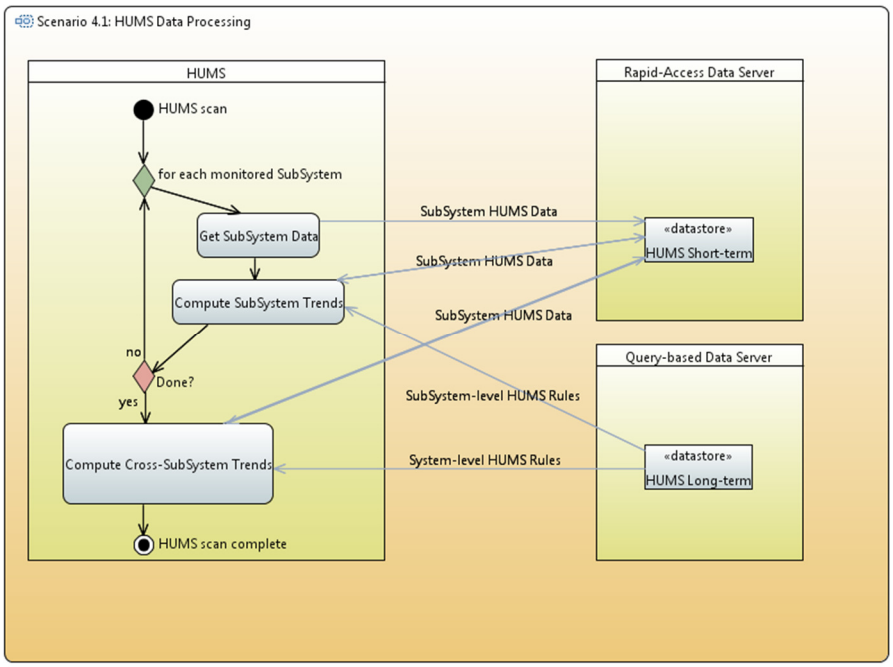
The HUMS will then process the data in order to detect trends in the data (per sub-system and across sub-systems) and raise information notices, warnings and/or alarms as necessary.

At each stage, data is cached or stored (cf. the Mission Management System).

The scenario is depicted as a UML Activity Diagram below (Figure 9)[2].

[2] In the Activity Diagrams of this document, black arrows indicate the flow of (program) control from one activity to another. Blue arrows indicate the flow of data between activities and/or data stores.

Figure 9 Scenario 4.1: HUMS Data Processing



6.4.1.2 Pre-Conditions:

None.

6.4.1.3 Initiating Event:

Periodic or on-event “HUMS scan” trigger.

6.4.1.4 Primary Activities:

i. For each monitored sub-system:
   a. HUMS gets the sub-system’s Health & Usage data;
   b. The data is cached to rapid-access store for later processing;
   c. The HUMS processes the sub-system data looking for Health & Usage trends, according to Rules obtained from long-term store;
   d. On condition, information notices, warnings, and/or alarms are raised;

ii. The HUMS processes the data looking for Health & Usage trends across the system (vehicle), according to Rules obtained from long-term store;

iii. On condition, information notices, warnings, and/or alarms are raised;

6.4.1.5 Post-Conditions:

Health and Usage for each sub-system obtained and processed for diagnostic trends.

6.4.1.6 Identified Data Servers:

i. HUMS rapid-access, short-term, data server.
ii. HUMS query-based, long-term, rule server.

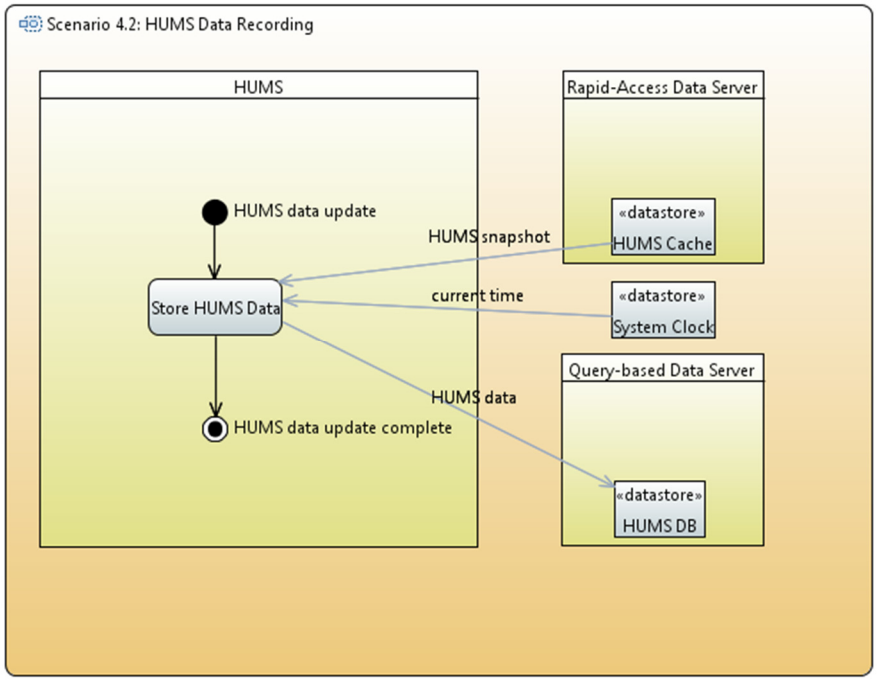
6.4.2 Scenario 4.2: HUMS Data Recording

6.4.2.1 Description:

Either periodically or on some trigger, the HUMS will snapshot the Health and Usage data collected from sub-systems and archive it to long term storage. Each snapshot is timestamped from the master system clock.

The scenario is depicted as a UML Activity Diagram below (Figure 10).

Figure 10 Scenario 4.2: HUMS Data Recording



6.4.2.2 Pre-Conditions:

None.

6.4.2.3 Initiating Event:

Periodic and/or on-event “HUMS data update” trigger.

6.4.2.4 Primary Activities:

i. For each monitored sub-system:
   HUMS takes a snapshot of the sub-system’s Health & Usage data;
ii. The data from all sub-systems is bound (packaged) and timestamped.
iii. The data package is posted to long term storage.

6.4.2.5 Post-Conditions:

Health and Usage data for all sub-systems archived for off-line processing and/or review.

6.4.2.6 Identified Data Servers:

i. HUMS rapid-access, short-term, data server.
ii. HUMS query-based, long-term, rule server.

6.4.3 Scenario 4.3: Weather Data

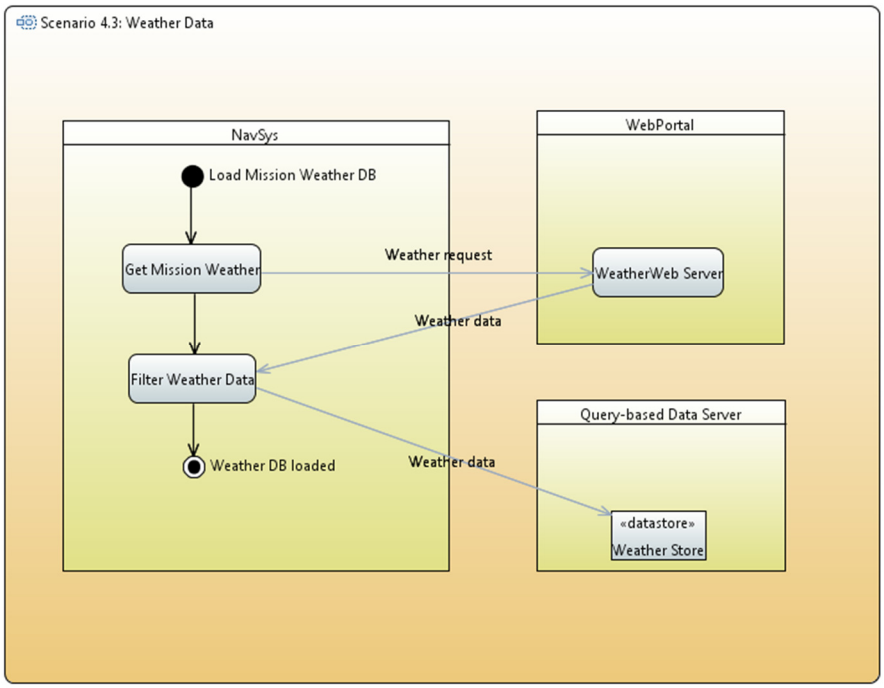
6.4.3.1 Description:

On a long term surveillance mission (for instance) it may be necessary for an advanced UAV Navigation System or Ground Control Station to request a weather update. This may be a query to a Weather Bureau via a web portal.

The received data will likely cover a larger area than the area of operation, so a filtering process may be required before the weather update is used and/or pushed to long term storage.

The scenario is depicted as a UML Activity Diagram below (Figure 11).

Figure 11 Scenario 4.3: Weather Data



6.4.3.2 Pre-Conditions:

None.

6.4.3.3 Initiating Event:

The current weather data's validity period expires.

6.4.3.4 Primary Activities:

i. The Navigation System issues a web service request on the Weather Bureau portal for the area of interest (operating area).
ii. The requested weather data is returned by the web portal.
iii. The weather data is filtered, e.g. to refine the area covered.
iv. The weather data of interest is pushed to long term data storage.

6.4.3.5 Post-Conditions:

Weather data for the operating area of interest available and stored.

6.4.3.6 Identified Data Servers:

i. Web service accessed bureau data servers.
ii. Query-based, long-term, Weather Store server.

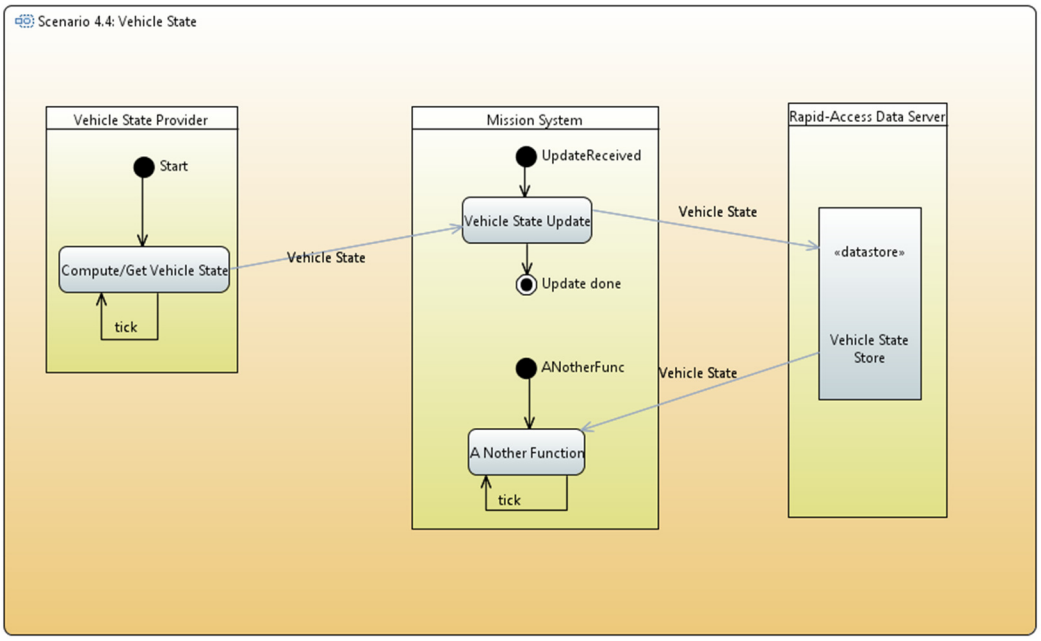
6.4.4 Scenario 4.4: Vehicle State

6.4.4.1 Description:

Rather than distribute Vehicle State (position, altitude, heading, speed etc.) to all sub-functions of the Mission System (routing, digital map, situation awareness, etc.), the MMS architecture (section 6.3) concepts might be applied, whereby the data is collected once from source and cached to a local rapid access data store. The sub-functions would then make (local) calls on the data store to access the data. Such an arrangement would also allow for a versioning mechanism, where the latest or previous versions of a data item can be accessed.

The scenario is depicted as a UML Activity Diagram below (Figure 12), with a *vehicle State Update* function receiving and caching the data, and the *A Nother Function* Mission System sub-function then using the data from the data store.

Figure 12 Scenario 4.4: Vehicle State



6.4.4.2 Pre-Conditions:

None.

6.4.4.3 Initiating Events:

The Vehicle State Provider updates the current Vehicle State and publishes/sends it to other sub-systems, including the Mission System.

6.4.4.4 Primary Activities:

i. The Vehicle State is received, for instance, on demand (if a publish-subscribe mechanism is used for Vehicle State distribution) or on update event.
ii. The Vehicle State is cached to a (local to the Mission System) rapid access data store.
iii. On demand, or periodically, other functions of the Mission System request, and receive, the Vehicle State from the (local) rapid access data store.

6.4.4.5 Post-Conditions:

None.

6.4.4.6 Identified Data Servers:

i. Mission System rapid-access, short-term, data server.

6.4.5 Scenario 4.5: Digital Map Tiles

6.4.5.1 Description:

This scenario reflects a more ordinary data management case, where a digital map application loads tiles of graphic (map) image to complete a displayed section of a map. Typically the centre section of a 7-by-7 grid of tiles is displayed, with the displayed section only covering part of the inner 5-by-5. Such a situation is illustrated in Figure 13 where some tiles (grey filled) have yet to be loaded from the tile store. The red box indicates the displayed area at some given instant when the air vehicle was moving roughly north-west.

Conventionally, each tile is stored in a separate file, named according to its position in the world-referenced coordinate system. As the air vehicle moves, the map engine discards, and loads, tiles in the outer ring of 24 so as to keep the displayed area always within the core 5-by-5 section.

Figure 13 Digital Map Tiling

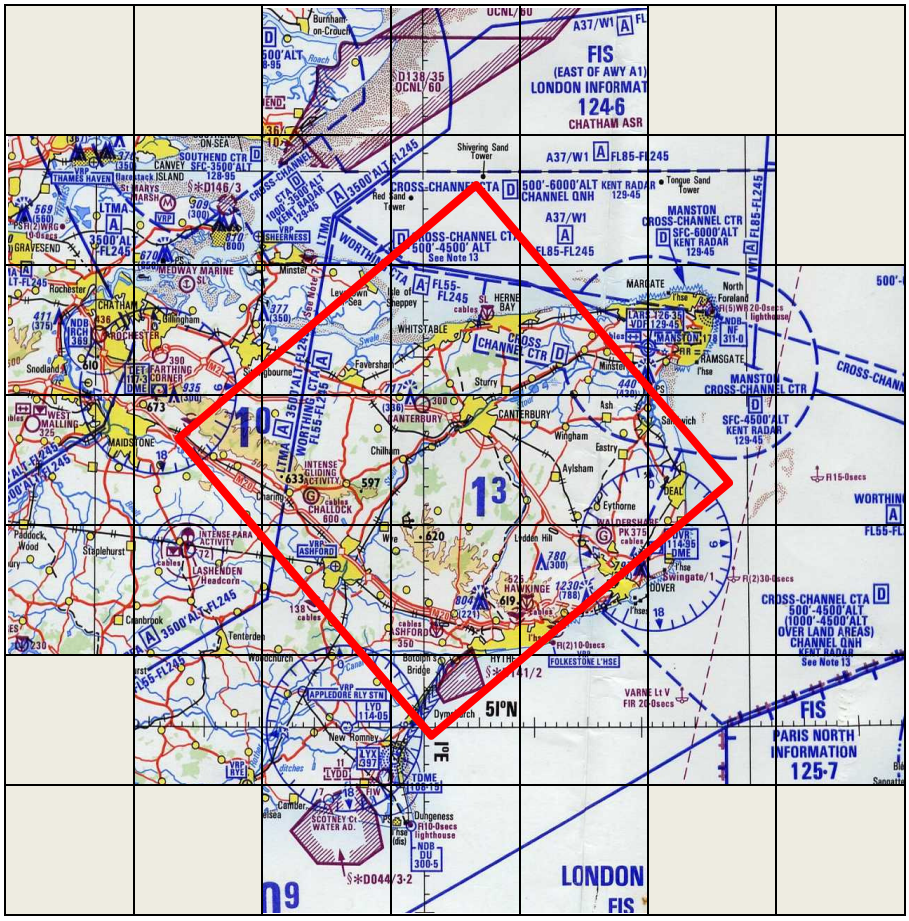
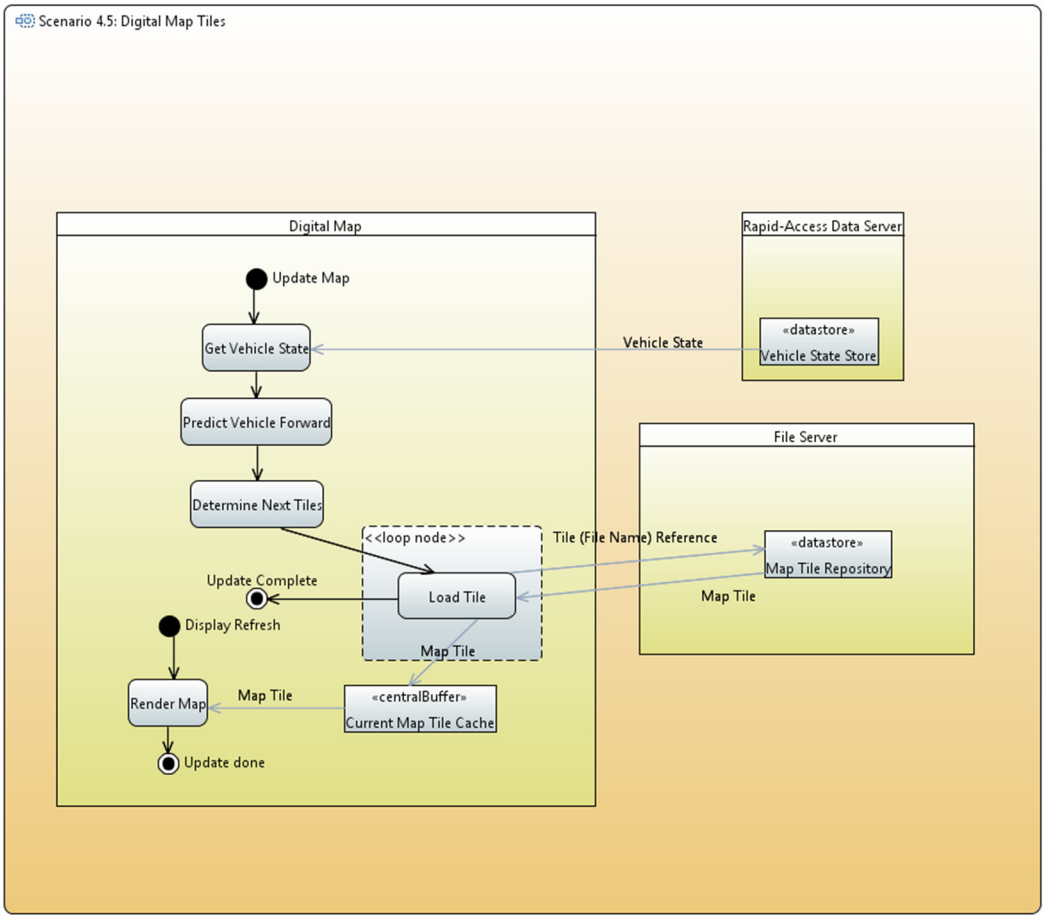


Figure 14 illustrates, as a UML Activity Diagram, the principal activities necessary (highly simplified for this scenario). Periodically, or on demand, the Digital Map function gets the latest Vehicle State, predicts forward to compute the tiles necessary to fill the grid, and then loads those tiles that are not currently loaded. The map display is continuously refreshed (at a much greater rate, e.g. 50Hz), displaying the display area within the current tile set..

Figure 14 Scenario 4.5: Digital Map Tiles



6.4.5.2 Pre-Conditions:

None.

6.4.5.3 Initiating Event:

Periodic, or on-demand, map update event.

6.4.5.4 Primary Activities:

i. The Digital Map function gets the current Vehicle State.
ii. A prediction forward is made to determine the current tile set validity.
iii. The necessary tile set is amended.
iv. New tiles are loaded from the Map Tile Repository.
v. Tile images are decompressed and the storage format is decoded.
vi. Tile bitmaps are pushed into the Map Tile Cache.
vii. The map is rendered (separately, and at a greater rate) onto the display surface using the tile bitmaps in the Map Tile Cache.

6.4.5.5 Post-Conditions:

The displayed digital map is apparently continuous in all directions.

6.4.5.6 Identified Data Servers:

i. Mission System (Vehicle State) rapid-access, short-term, data server.
ii. Mission System (Map Tile Repository) file store server.

6.4.6 Scenario 4.6: Target Data

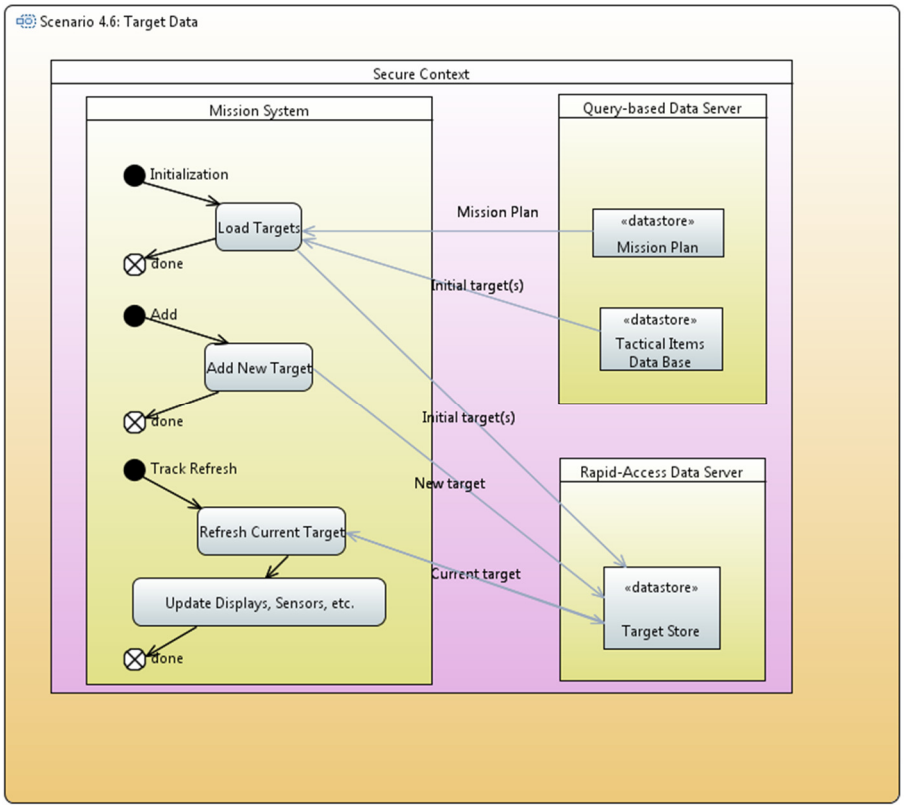
6.4.6.1 Description:

This scenario depicts a case where Tactical Item data stored in a query-based data store is copied to a rapid access, maybe local, data store for use by the mission system. Such a case may arise where the tactical item becomes a target which must be tracked, so the item data changes rapidly.

In the scenario, initial target(s) identified in the Mission Plan are loaded and cached. New targets are added as mission creep occurs, and the current target data is refreshed as the target is tracked.

All the interactions occur within a secure data context.

Figure 15 Scenario 4.6: Target Data



6.4.6.2 Pre-Conditions:

None.

6.4.6.3 Initiating Event:

i. Mission System initialization;
ii. New target identified;
iii. Periodic or on-demand target refresh request.

6.4.6.4 Primary Activities:

i. On initialization:
   a. The Mission Plan is loaded.
   b. Plan (initial) target ids. are extracted.
   c. The target data is obtained for the Tactical Items data store using the target ids.
   d. The target data is pushed to the (possibly local) rapid access Target data store.

ii. New targets identified during the mission (e.g. discovered threats) are pushed to the Target data store for use by opportunistic mission re-planning and routing functions.

iii. Periodically, or on-demand, data for tracked targets is refreshed, and displays updated.

6.4.6.5 Post-Conditions:

None.

6.4.6.6 Identified Data Servers:

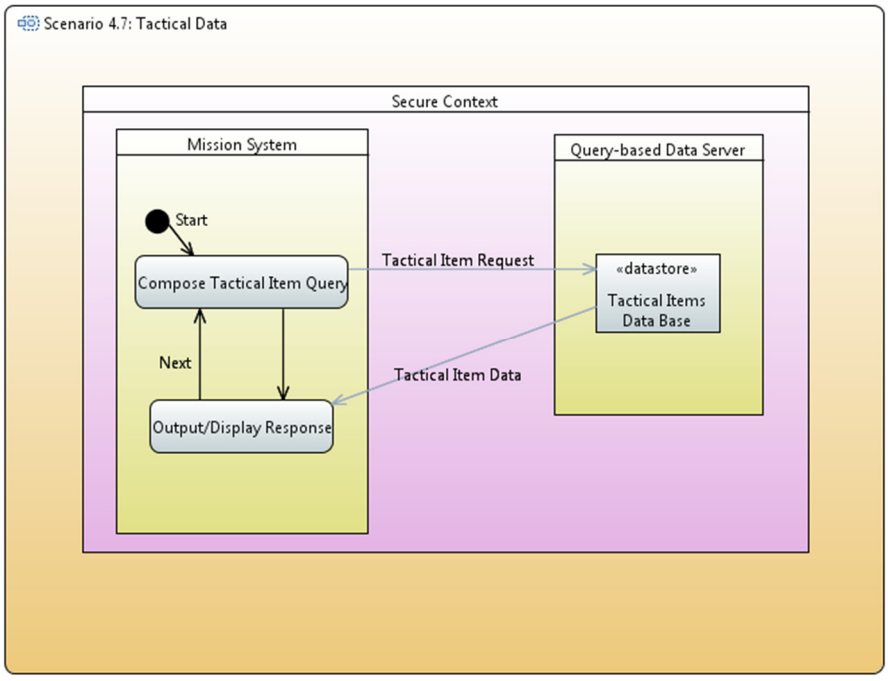
i. Mission Plan secure, query-based, data server.
ii. Tactical Items secure, query-based, data server.
iii. Target data, secure, rapid access, data server.

6.4.7 Scenario 4.7: Tactical Data

6.4.7.1 Description:

In an advanced Mission System with opportunistic mission re-planning and re-routing functions, it will be necessary to be able to make ad hoc, random, queries of the Tactical Items data store as the battlespace picture builds and develops. For instance a detected threat may require a query of the data server to identify threat type and appropriate avoidance or counter-measures.

Figure 16 Scenario 4.7: Tactical Data



6.4.7.2 Pre-Conditions:

None.

6.4.7.3 Initiating Event:

A threat is indicated.

6.4.7.4 Primary Activities:

i. The location of the threat is identified.
ii. Some identifying threat characteristic is detected.
iii. A query is posed to identify the threat type
iv. The appropriate response to the posed threat is sought.
v. The results are displayed (to the air crew or ground operator) or acted upon (autonomous air vehicle capability).

6.4.7.5 Post-Conditions:

None.

6.4.7.6 Identified Data Servers:

i. Tactical Items secure, query-based, data server.

7 Design Considerations

A number of significant design choices must be considered when creating a data server implementation, only some of which are directly related to the use or non-use of the ECOA. The most significant of these are discussed below.

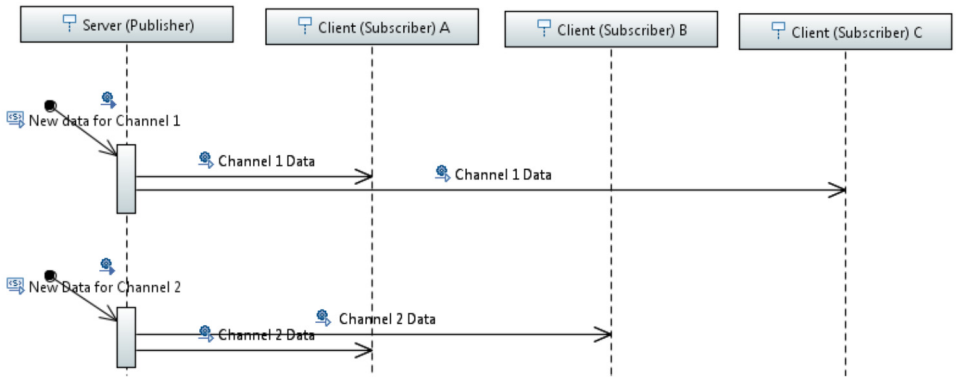
7.1 Pushmi Pullyu

The choice of whether the data server pushes data out to a client, or the client pulls data in from the server can have a profound effect on the design of that part of the system, its runtime performance, and on the ability to predict and analyse the behaviour of the system.

7.1.1 The Push Model

In the push model, data transfers are initiated entirely by the data server, either as the data publisher or as a central distribution server. It is the most common mechanism in a publish-subscribe scenario, where a client expresses information preferences in advance "subscribing" to particular data channels, and the server then publishes new data content as it occurs on those channels. The subscriber(s) have only to listen for new publications.

Figure 17 Push Model Behaviour



In the illustrated case, three clients subscribe to channels published by a server; clients A and C to channel 1, and clients A and B to channel 2. For each channel, whenever the server has new data available, the data is pushed out to the subscribing clients.

7.1.1.1 Advantages:

- The latest data is immediately (subject to server handling latency and publish interval period) available to the client when needed.
- Potentially reduced data traffic, as there are no per-transfer data requests.
- Unidirectional data transfer (e.g. network) route.
- Analysable data transfer timings and latency.

7.1.1.2 Disadvantages:

- Client-side data reception processing when the data isn't required.
- Potentially unnecessary data traffic as data is sent even if not needed.

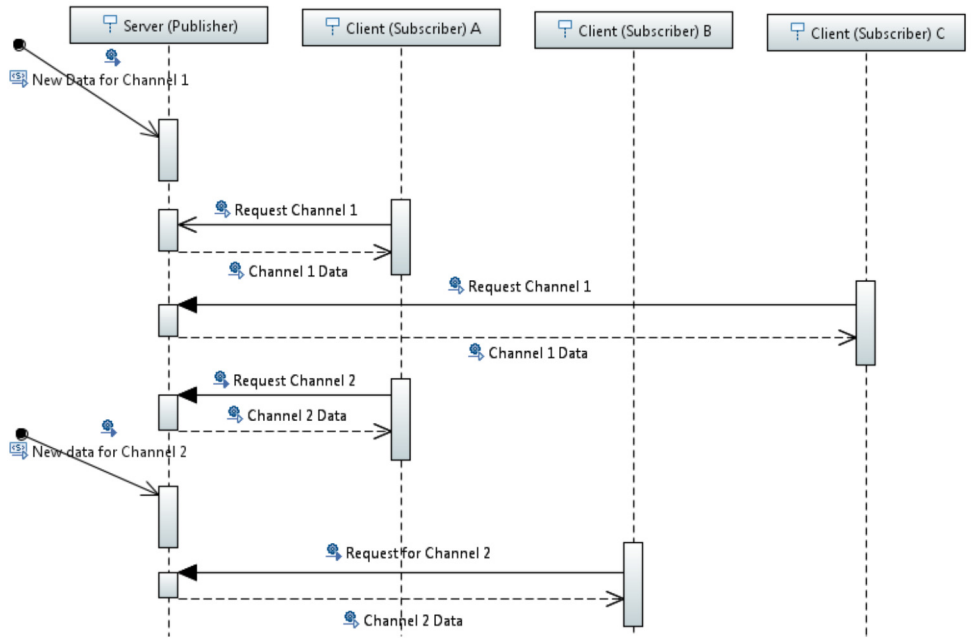
7.1.1.3 Mitigation Strategies

- Runtime (possibly one-time) definition (by client request) of per-channel data sub-sets (from the overall "Topic" data set) to be transferred (at the expense of a bi-directional interchange route).
- Runtime (possibly one-time) definition (by client request) of the period or triggering event of per-channel transfers.

7.1.2 The Pull Model

In the Pull Model, data transfers are initiated by the client with a request for data, and the server responds to the request by sending the data requested. Since data is explicitly requested, the request can carry data selection criteria, or even processing instructions, to the server, so that data sub-sets or syntheses can be returned.

Figure 18 Pull Model Behaviour



In the illustrated case, as before, clients A and C require data from channel 1, and clients A and B from channel 2. However, each client must request the data when required, and the server will return it.

The update of new data at the server is often asynchronous of requests from clients. The diagram therefore illustrates a case where the server receives a data update between requests from clients A and B, in which case, client B receives more recent data than A. This situation very often leads to time-tagging the data on reception at the server. A client can then know the age of the data when it is received (by looking at the time-tag). However time-tagging will increase the net amount of data transferred.

7.1.2.1 Advantages:

- Data transfer occurs only when the data is needed (whether periodic or on asynchronous demand).
- Transferred data sub-set can be tailored/selected on a request-by-request basis.
- Server side processing can be utilized to process/synthesize data sets according to the request.

7.1.2.2 Disadvantages:

- Additional data traffic due to the request.
- Data selection/processing/synthesis latency between the request and the response from the server.
- Potential latencies between source data update at the server and the data delivered on request.
- Bi-directional data transfer (e.g. network) route required.

7.1.2.3 Mitigation Strategies

- Predefinition of complex data sub-setting criteria/rules, which are then stored server-side, and invoked by much reduced individual requests (e.g. predefined SQL queries/views).
- Concentration of data processing capability at the server, leaving client processing free to concentrate on data usage.

7.2 Operation Choice

The ECOA provides for implementing both the push and pull models.

7.2.1 Request-Response (Pull Model)

ECOA request-response operations closely correspond to the traditional data client-server, pull model, relationship. The client sends a data request (perhaps composing data selection criteria or processing instructions using a query language or other mechanism), which is sent to the data server. On receipt of the request the data server returns the requested data selected or synthesized (from a greater data set) according to the request criteria/instructions if appropriate.

For small, low latency, requests, synchronous invocation of the ECOA request-response operation may be used. When the requested data set is large (involving a significant (e.g. network) transfer period), or there can be a significant (e.g. selection or processing) latency between the request and the response, asynchronous invocation of the ECOA request-response operation is likely to be preferable.

7.2.2 Events (Push Model)

A simple push model scheme can be implemented using ECOA Events. When the data is updated, and a data transfer trigger occurs (e.g. a period expires) the server simply packages the required data set into an ECOA Event and sends it to all subscribing clients.

Aircraft state data would be a typical example of a basic data server implemented this way. Periodically, the server packages the current aircraft state (position, altitude/height, speed, attitude, etc.) and sends it out.

It should be noted that ECOA Events can be **SENT_BY_PROVIDER** or **RECEIVED_BY_PROVIDER**. It is therefore possible to implement the push model where the data server is the ECOA Service *provider* (with the client *referencing* the Service), OR where the data server is actually the *referencing* ASC and the data client is the Service *provider* ASC.

7.2.3 Versioned Data (Implementation Defined)

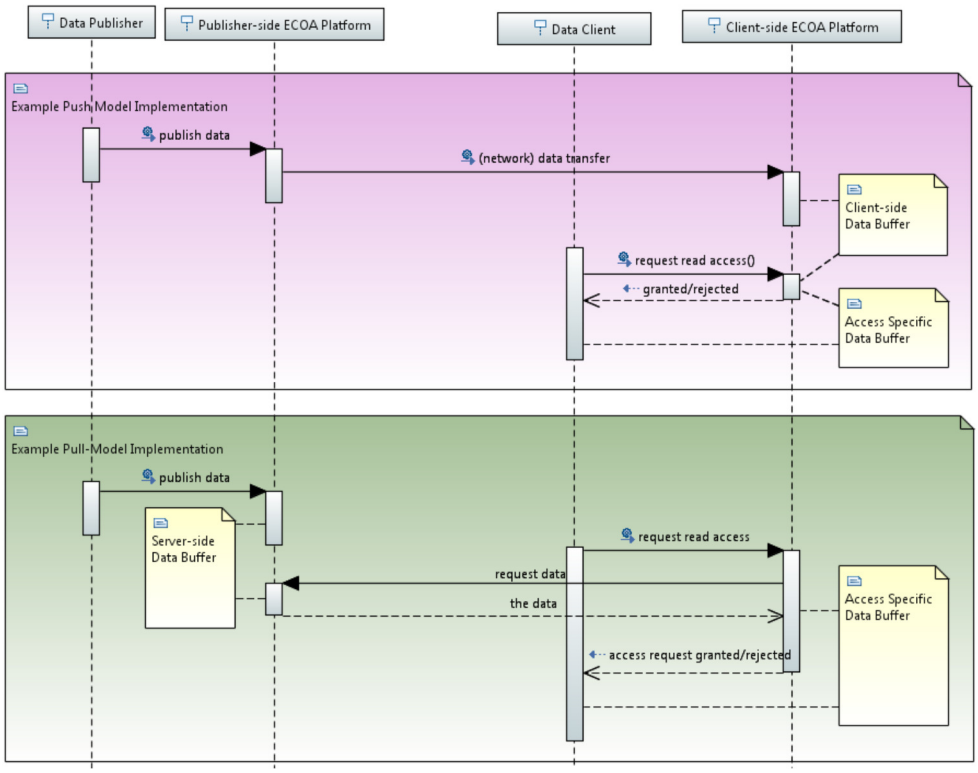
Another form of the client-server relationship can be implemented using ECOA Versioned Data “operations”. ECOA Version Data provides a mechanism by which a client can apparently simply view data content created by the server, albeit with an access control mechanism by which the client requests (read) access to the data.

The ECOA does not define, when the client and server are not co-located, whether Versioned Data transfers are implemented as push or pull model transfers – that is whether the data is transferred from the server to the client when (read) access is requested, or whether the transfer occurs when a data update is published (by the server). ECOA simply requires that when a client receives (read) access to the data, the latest published data is available. Whether the ECOA Software Platform code has already transferred (a copy of) the data, or whether it is transferred on demand, is left as an implementation decision (for the ECOA Software Platform designer). The two possible cases are illustrated (as UML sequence diagrams) below (Figure 19).

The first scenario illustrates an example push-model implementation. When the Versioned Data item is published by the server, a copy is immediately made in a data buffer in the ECOA platform of each subscribing client (only one client is illustrated). When the client requests read access, a copy of the data is made by the ECOA platform that will be unique for this particular access, and the client is (all being well) given access to the contents of that buffer. Another publish by the server will not therefore over-write the contents of the per-access copy.

In the second (pull-model) example implementation scenario, the Versioned Data item is, when “published” by the server, only copied into a buffer on the server side. Only when a client requests access to the data is it copied from the server side to the client side. As before, the client side ECOA platform places the data in a buffer specific to the access request, and it is to this that the client is then given access.

Figure 19 ECOA Versioned Data - Push/Pull Implementations



7.3 Data Encoding

7.3.1 Base64 Encoding

Base64 (ref. *[B64]*) is one of a number of binary-to-text encoding transforms that allow binary data to be transferred over text streams. This is done by taking three bytes (24 bits) and treating these as four 6 bit values (still 24 bits). Each of the four 6 bit values is then mapped onto one of 64 printable universal ASCII characters, for instance 'A' to 'Z', 'a' to 'z' and '0-9' (plus two others such as '+' and '/').

The encoded message, now pure text, can then be transmitted over any text-capable channel, and decoded at the other end, without fear of control codes and nulls contained in the original data interfering with the transport protocols. The expense is the 3:4 increase in transmitted number of bytes.

If the original data is not wholly divisible into 24 bit groups, the end of an encoded message is padded with a 65th (special) character (such as '=').

Figure 20 Example Base64 Encoded Value

```
VGhpcyBpcyBhIHN0cmluZyBUaGlzIGlzIGEgc3RyaW5nIFRoaXMgaXMgYSBzdHJpbmcgVGhpcyBp
cyBhIHN0cmluZyBUaGlzIGlzIGEgc3RyaW5nIFRoaXMgaXMgYSBzdHJpbmcgVGhpcyBpcyBhIHN0
cmluZyBUaGlzIGlzIGEgc3RyaW5nIFRoaXMgaXM=
```

7.3.2 XML

The Extensible Markup Language (XML) (ref. *[XML]*) "…*describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them.*"

"*XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.*"

XML therefore provides a structured, text based, highly platform independent representation of data that can be stored and transferred between servers and clients. Because of its platform independent nature, and the ability to create self-describing data hierarchies, XML has rapidly become a *de facto* standard for data interchange across the internet.

An apparent disadvantage of transferring an XML representation of data is the quantity of XML text required to represent the data. However with compression, XML text can be compressed by 60 to 90% or more, and even with the resulting binary Base64 encoded, the transmitted byte-count can be very much less than that of the original XML. Compression and decompression though impose a processing overhead.

7.3.3 Key-Length-Value (KLV)

Key-Length-Value is a data encoding standard, used to embed data into binary streams, particularly video feeds. Data items are encoded as Key-Length-Value triplets, where the *key* identifies the data using a numerical (e.g. hash-coded) value, the *length* specifies the data's length (in bytes), and the *value* is the data itself. It is defined in SMPTE 336M-2007 (ref. *[KLV]*), approved by the Society of Motion Picture and Television Engineers. The *value* is often itself a set of KLV packets, subdividing the data set.

The length of the *key* value (1, 2, 4 or 16 bytes), and its meaning, need to be fixed and specified for a given application. The *length* field too may be 1, 2, or 4 byte numbers, or can be an encoded using Basic Encoding Rules (BER) (ref. *[BER]*) where the first byte of the *length* field states how many following bytes are used to represent the *length* value (up to 127). Potentially then, a complete KLV record could be as short as 3 bytes (1 byte *key* + 1 byte *length* + 1 byte *value*), or as long as $16+(1+127)+2^{127}$ bytes!

KLV encoding has the advantage over XML in that it has a purely binary overhead, and unpacking data does not require text string parsing. So where a single XML data item comprising a tag, the value, and an end-tag, such as:

```
<AltitudeFt>10254</AltitudeFt>
```

is 30 bytes of transmitted XML (though this might become 15 bytes or less after compression and Base64 encoding), the KLV equivalent might be 12 bytes, 4 for each of the *key*, *length*, and *value*, for instance:

Byte	1	2	3	4	5	6	7	8	9	10	11	12
Meaning	*Key*				*Length*				*Value*			
Value	17751				4				10254			

Whilst far more efficient than pure-text XML, the efficiency of KLV encoding over unstructured binary will very much depend on the size and structure of the data be conveyed. It would be relatively inefficient, for instance, for transferring one-byte data values, as three bytes would be required. However as the individual data item sizes increase, the Key-Length relative overhead diminishes and the benefits of a self-describing data structure dominate.

The disadvantage of KLV encoding over XML is that the transferred data packets are binary and only readable by being able to identify KLV blocks, and recognise the data item by its *key* value.

8 ECOA Data Server Designs

This chapter takes the four data server types identified in para. 6.4 (query-based, rapid-access indexed, direct file I/O, and web service accessed) and proposes an example ECOA Service and Provider ASC to illustrate each.

The object of defining an ECOA Service for each of these data server types, rather than simply accessing a COTS data server API directly within a client code, is that it provides one more opportunity for the client code to be portable and reusable within the ECOA universe.

8.1 Query-based Data Servers

Query-based data servers are used by composing statements and commands using a Query Language which are then passed by the client to the server. The server parses (interprets) the statement or command and invokes the functions necessary to implement and respond to that statement or command. The result is then passed back to the client.

Several query languages exist, some of which are built upon XML, but the most common is the ISO standardized Structured Query Language (SQL) (ref. *[SQL]*). SQL is an English-oriented language comprising structured statements that are easily understood by both humans and machines.

For instance, the SQL command:

SELECT * FROM Weather WHERE Region='SouthEast' and Country='UK';

instructs the server to retrieve all entries (as denoted by the "*******") in the "***Weather***" data structure that have fields named "***Region***" and "***Country***" having the values of "***SouthEast***" and "***UK***" respectively.

Data structures in an SQL data server are organised as Tables, each comprising multiple Row entries, each of which will comprise one or more Fields containing data values.

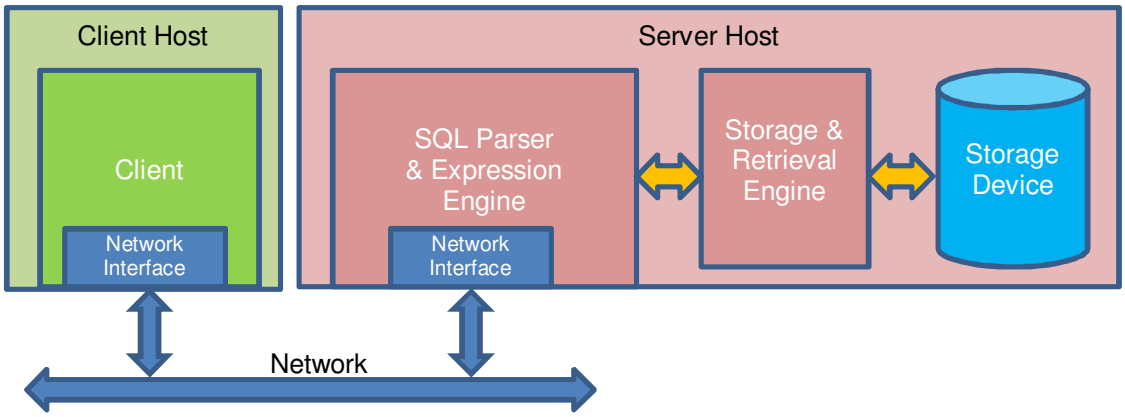
As well as providing the means for organising large amounts of data (by using Tables) data servers driven by SQL provide an engine for processing (often) complex search and retrieval criteria, including cross-referencing, selecting, and sorting according to, data taken from multiple Tables. For instance:

```
SELECT DISTINCT Data_Items.Data_Item, Data_Items.Description, Sources.Source_System,
     Formats.Format
  FROM (Formats
    INNER JOIN Data_Items ON Formats.Format = Data_Items.Format)
    INNER JOIN (Systems
      INNER JOIN Sources ON Systems.System = Sources.Source_System)
      ON Data_Items.Data_Item = Sources.Data_Item
  GROUP BY Data_Items.Data_Item, Data_Items.Description, Sources.Source_System,
     Formats.Format
  ORDER BY Data_Items.Data_Item, Sources.Source_System;
```

returns a number of uniquely ("***DISTINCT***") valued data items, grouped ("***GROUP BY***") and sorted ("***ORDER BY***") according to certain criteria, taken from a number of different tables ("*System*", "*Data_Items*", "*Sources*", etc.) within the data server.

SQL-driven data servers provide a mechanism for transferring SQL statements from, and responses to, client applications, possibly over a network. ODBC (ref. *[ODBC]*) for instance is a very common software interface (API) standard that allows programmatic connection to, and usage of, a data server using SQL as the access language.

Figure 21 Typical SQL Client-Data Server Configuration



8.1.1 Advantages:

Query-based data servers can provide:

- Very high levels flexibility in the storage and usage of data, e.g.:
  - Data structures (tables) can be created and deleted at runtime;
  - Specialist subset data structures can be created and deleted at runtime using search/selection criteria;
  - Runtime modification of individual data values within data structures.
- Complete platform independence between the client and server as the exchanged data is plain (query language) text.
- High levels of reusability of query language statements, commands, and scripts.
- A standards-based implementation of data server capability.
- Powerful, scalable, data search and retrieval capabilities, based on advanced search criteria definition.
- Scalable server-side data analysis, correlation, and processing capabilities.

8.1.2 Disadvantages:

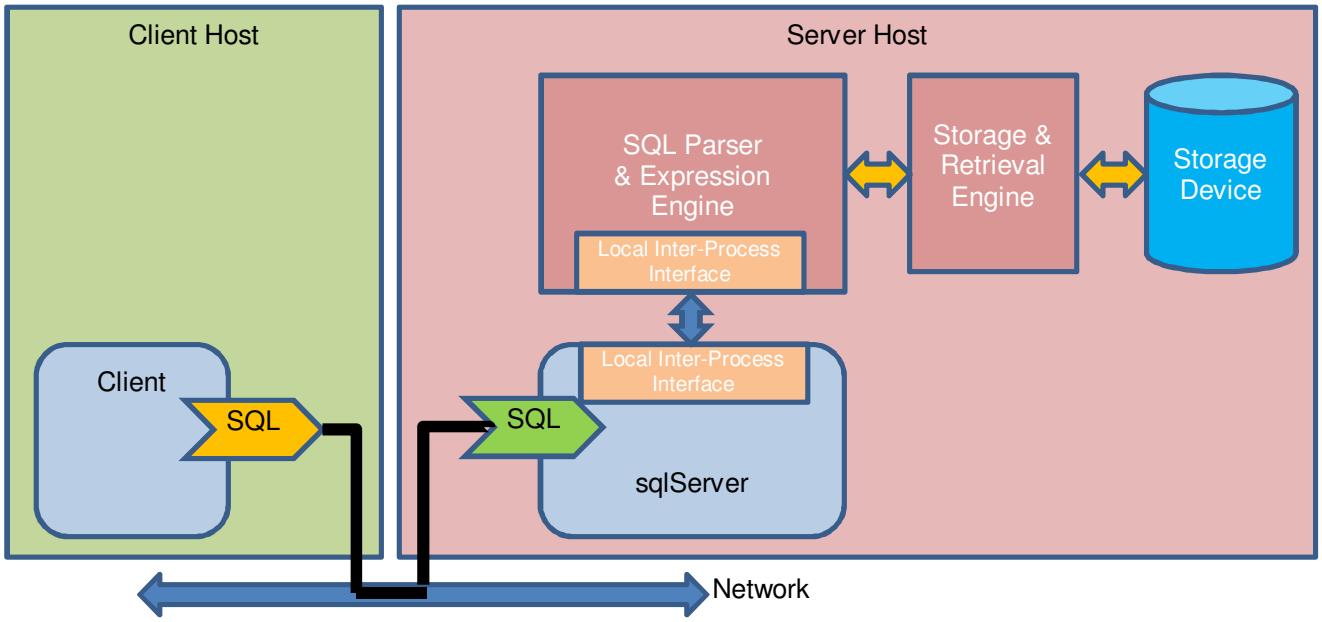
However, for a query-based data server the response time, compared with an indexed data server, will be higher, due to:

- Having to parse and interpret the query language;
- Data search (based on the query) and recovery times.

8.1.3 An ECOA SQL Data Management Service

In the ECOA domain, we can envisage a practical translation of the arrangement of Figure 21 into that of Figure 22:

Figure 22 ECOA SQL Client-Data Server Configuration



Here, the data server itself is 'hidden' behind an ECOA *sqlServer* ASC, with localized inter-process communications. All traffic between Client(s) and Server is now in the ECOA domain, so may physically invoke the ELI network protocol (as illustrated), or not if the Client is deployed to the same host as the Server.

8.1.3.1 Requirements

The following is a set of base requirements for an ECOA Service that can be used to provide access to a SQL query-based data server.

1) The SQL Service should allow an ECOA client ASC to connect to (access) a nominated (named) data server, possibly across a network.
2) The SQL Service should allow an ECOA client ASC to create and destroy data tables on a data server that it is connected to.
3) The SQL Service should allow an ECOA client ASC to store to, search for, and retrieve data from, tables, using SQL statements.
4) The SQL Service should minimise the data trafficked between the Service Provider and the data server itself.
5) The SQL Service should allow an ECOA client ASC to register to be notified whenever a table is updated.
6) The SQL Service should allow an ECOA client ASC to pre-load tables with data taken from a SQL data file.
7) The SQL Service should allow an ECOA client ASC to save the current contents of the data server to a SQL data file.
8) The SQL Service should allow an ECOA client ASC to interrogate the storage and content limits of the data server.
9) The SQL Service should allow an ECOA client ASC to create and destroy 'view' tables on the data server, using SQL statements defining specific data selection criteria.
10) The SQL Service should allow an ECOA client ASC to retrieve data composed (by the server) into 'view' tables without having to re-specify the selection criteria.

8.1.3.2 Required Operations

From para. 8.1.3.1, the minimum set of Service Operations is then:

1) Create (get) a connection to a data server.
2) Destroy (end) a connection.

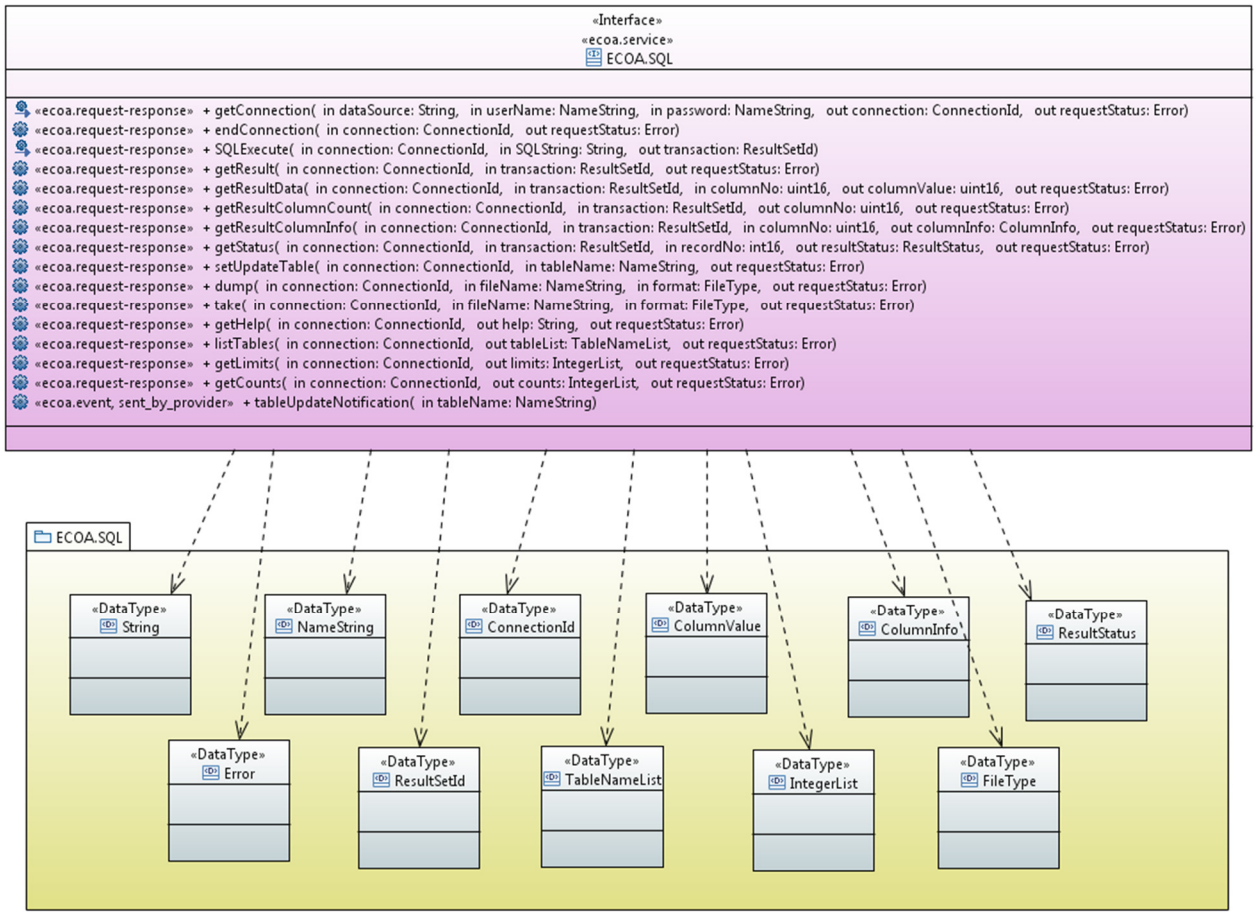
3) Execute an SQL statement as an identifiable “transaction”.
4) Get the result(s) of a particular transaction.
5) Register for table update notifications.
6) Load (take) or save (dump) table contents from/to an SQL file.
7) Report (get) the data server limits and meta-data.

The creation and deletion of data tables and views (requirements (2) and (9)) and the selection and retrieval of data (requirements (3) and (10)), will be achieved by use of SQL statements, and therefore accomplished using operation (3).

8.1.3.3 Service Definition

Expanding this minimal set with a few additional operations, we get the Service Definition depicted in Figure 23. For clarity of purpose, the details of the actual data types, all of which are defined in the *ECOA.SQL* Types Library, are omitted from this diagram. To clearly maintain the conventional client-server (pull-model) relationship, this ECOA.SQL Service is defined in terms of ECOA Request-Response Operations. Only a posted notification is expressed as an ECOA Event Operation. Each operation is described after the diagram.

Figure 23 ECOA SQL Service Definition (as a UML Interface Class)



8.1.3.3.1 getConnection(…)

This ECOA Request-Response Operation will create a connection to a named *dataSource*, using the authentication parameters *username* and *password*. A unique Connection Identifier (*connection*) will be returned, together with a *requestStatus* code.

8.1.3.3.2 endConnection(…)

This ECOA Request-Response Operation will end (close) the identified *connection*, and return a *requestStatus* code.

8.1.3.3.3 SQLExecute(…)

This ECOA Request-Response Operation will send the *SQLString* statement to the data server at *connection*, and return a *transaction* handle. This *transaction* handle will be used in subsequent operations to extract the data resulting from execution of the *SQLString* statement.

8.1.3.3.4 getResult(…)

This ECOA Request-Response Operation will transfer the data associated with a *transaction* on the data server at *connection*, from the (remote) data server to the (local) client, and return a *requestStatus* code. The result data can now be accessed without further network activity.

The result data will be presented as a table of values with a number of rows (which may be zero) each of a number of fields (columns) (which may also be zero).

8.1.3.3.5 getResultData(…)

This ECOA Request-Response Operation will extract the data value from field *columnNo* of the result data associated with a *transaction* on the data server at *connection*, return the data value in *columnValue*, and return a *requestStatus* code.

8.1.3.3.6 getResultColumnCount(…)

This ECOA Request-Response Operation will return the number of fields (columns) in the result data associated with a *transaction* on the data server at *connection*, return the data value in *columnValue*, and return a *requestStatus* code.

8.1.3.3.7 getResultColumnInfo(…)

This ECOA Request-Response Operation will return the meta-data for the field (column) *columnNo* of the result data associated with a *transaction* on the data server at *connection*, return the meta-data in *columnInfo*, and return a *requestStatus* code. The meta-data includes the field name, a code indicating the field's data type (integer, real, data, etc.), and the field's data type size (in number of bytes).

8.1.3.3.8 getStatus(…)

This ECOA Request-Response Operation will return (as *resultStatus*) the status of the result data associated with a *transaction* on the data server at *connection*, and return a *requestStatus* code.

A *resultStatus* is different from a *requestStatus*, the latter indicating whether a particular Service Operation succeeded and why if it did not, whilst the former returns information from the underlying data server about the status of the particular result data.

8.1.3.3.9 setUpdateTable(…)

This ECOA Request-Response Operation will register the client for notification whenever the table *tableName*, on the data server at *connection*, is updated (i.e. has a value changed), and return a *requestStatus* code.

8.1.3.3.10 dump(…)

This ECOA Request-Response Operation will save the entire data store, on the data server at *connection*, to the file *filename*, and return a *requestStatus* code. The file will be written as SQL or XML depending on the value of the *format* parameter.

8.1.3.3.11 take(…)

This ECOA Request-Response Operation will read from the file *filename*, and execute the statements contained, on the data server at *connection*, and return a *requestStatus* code. The file will be expected to contain either SQL or XML depending on the value of the *format* parameter.

XML files can be used to populate tables on the data server. SQL files can be used to populate tables and perform data queries and extractions. The result of only the last SQL statement that produces a result will be available for retrieval by the client.

8.1.3.3.12 getHelp(…)

This ECOA Request-Response Operation will return (in parameter *help*) a text string of useful "help" information from the data server at *connection*, and return a *requestStatus* code.

8.1.3.3.13 listTables(…)

This ECOA Request-Response Operation will return, as *tableList*, a list of the names of the tables currently held on the data server at *connection*, and return a *requestStatus* code.

8.1.3.3.14 getLimits(…)

This ECOA Request-Response Operation will return, as *limits*, a list of limit values relevant to the data server at *connection*, and return a *requestStatus* code.

The list will include:

- the maximum length of an ECOA.SQL.String;
- the maximum length of an ECOA.SQL.NameString;
- the maximum size of an ECOA.SQL.CLOB[3];
- the maximum size of an ECOA.SQL.BLOB[4];
- the maximum length of an ECOA.SQL.IntegerList;
- the maximum length of an ECOA.SQL.TableNameList;
- the maximum number of simultaneous connections that are supported.

8.1.3.3.15 getCounts(…)

This ECOA Request-Response Operation will return, as *counts*, a list of current values relevant to the data server at *connection*, and return a *requestStatus* code.

The list will include:

- the number of tables;
- the number of transactions completed;
- the number of transactions pending;
- the number of connections.

8.1.3.3.16 tableUpdateNotification(…)

This ECOA Event Operation, sent by the provider to the client, reports that the table *tableName* has been updated (changed). No indication of what the change might be is given or available. A **tableUpdateNotification(…)** Event will only be sent if the client has previously registered by invoking the **setUpdateTable(…)** Operation.

---

[3] CLOB = Character Large Object

[4] BLOB = Binary Large Object

8.1.3.4 Service Definition XML

The following are a few lines of the formal ECOA Service Definition XML for this Service. The entire Service Definition is not given here for brevity, but it will be seen that each Operation of the Service is defined using the data types depicted in the UML diagram above (Figure 23).

In ECOA, a Service Definition XML comprises a `<serviceDefinition>` XML element composed of an `<operations>` list XML element. Each Operation provided by the Service is listed (in this case either as `<requestresponse>` XML elements or as `<event>` XML elements) within the `<operations>` element. Each Operation lists its input and output parameters as either `<input>` and `<output>` XML elements.

Listing 1 SQL Service Definition XML

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0" >
  <use library="ECOA.SQL"/>
  <operations>
    <requestresponse name="getConnection">
      <input  name="dataSource" type="ECOA.SQL:String"/>
      <input  name="userName" type="ECOA.SQL:NameString"/>
      <input  name="password" type="ECOA.SQL:NameString"/>
      <output name="connection" type="ECOA.SQL:ConnectionId"/>
      <output name="requestStatus" type="ECOA.SQL:Error"/>
    </requestresponse>
    <requestresponse name="endConnection">
      <input name="connection" type="ECOA.SQL:ConnectionId"/>
      <output name="requestStatus" type="ECOA.SQL:Error"/>
    </requestresponse>
       :
       : (snipped)
       :
    <event name="tableUpdateNotification" direction="SENT_BY_PROVIDER">
      <input name="tableName" type="ECOA.SQL:NameString"/>
    </event>
  </operations>
</serviceDefinition>
```

The ECOA XML files for the example Services and ASCs discussed in this document are available from the ECOA website (ref. *[CODE]*).

8.1.4 An ECOA SQL Data Service Provider ASC

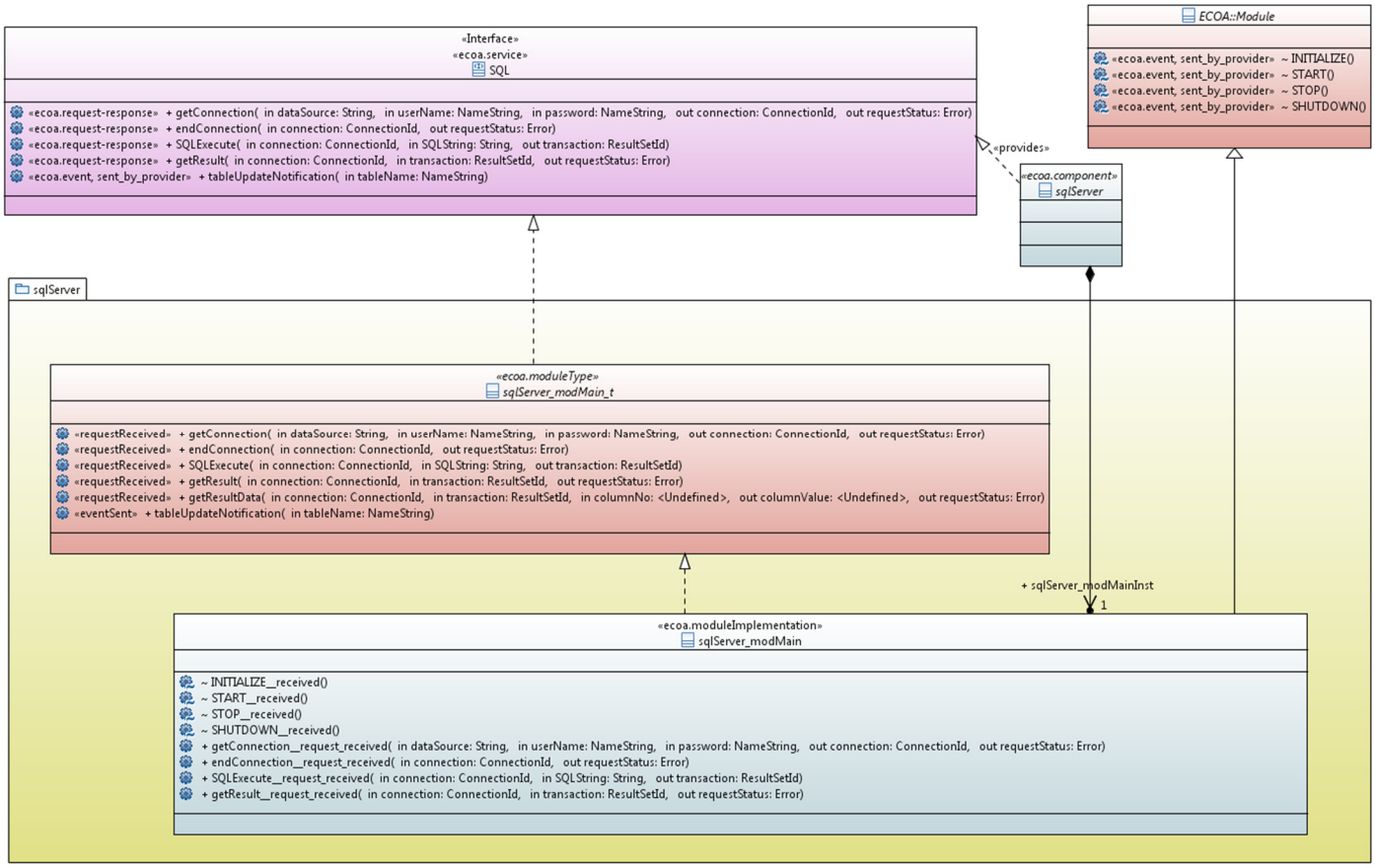
For the present purposes, the defined ECOA SQL Service will be provided by a self-contained *sqlServer* ECOA ASC, described in UML in Figure 24, and defined by the ECOA Component Implementation XML that follows (Listing 2).

For brevity within this document, only the first few Operations are shown in each of the UML entities in the diagram. The complete set of Service Operations was shown in Figure 23 and need not be repeated. Likewise, the (ASC) Component Implementation XML following has been snipped where repetition of similar structures, one for each operation, has been curtailed.

Referring to Figure 24, the *sqlServer* ECOA ASC, represented as an UML class and stereotyped *«ecoa.component»*, ***provides*** the *SQL «ecoa.service»* - indicated by the UML realization relationship (closed-headed dashed arrow, stereotyped *«provides»*) – by being ***composed of*** the *«ecoa.moduleImplementation» sqlServer_modMain*. At runtime, the single specific instance of the *sqlServer_modMain* implementation is named *sqlServer_modMainInst* – as indicated by the decorations on the ***composed of*** (or "***aggregation***") relationship arrow.

The *sqlServer_modMain* implementation is itself a realization[5] of two interface definitions, one defining the lifecycle operations that all ECOA Modules must provide and here depicted as the UML Interface class *ECOA.Module*, and the other being the Module Type definition ***derived from*** (or "***specialized***" from) the *SQL* ECOA Service, namely the *«ecoa.moduleType»* *sqlService_modMain_t*. Thus *sqlService_modMain_t* exports all the Module Operations defined by the *SQL* Service, but they are now stereotyped as *«requestReceived»* (or in the case of **tableUpdateNotification()** *«eventSent»*) because they are Operations on the ***provider*** Module Type[6]. Similarly, the Operations appear again on the Module Implementation class (*sqlService_modMain*), along with the (implementation of) Operations from the *ECOA::Module* abstract class.

Figure 24 sqlServer ASC Design (as UML Class Diagram)



8.1.4.1 Component Implementation XML

In ECOA, a Component Implementation XML formally defines the construction of an ASC, comprising a `<componentImplementation>` XML element composed of one or more `<moduleType>`, `<moduleImplementation>`, and `<moduleInstance>` XML elements. Each `<moduleImplementation>` and `<moduleInstance>` element expresses the realization of a Module Type and the instantiation (at runtime) of the Module Implementation respectively.

[5] In UML, a "realization" denotes where an interface definition is made real by an implementation.

[6] If the ASC ***referenced*** the Service rather than ***providing*** it, the Operations of the Module Type would be stereotyped *«requestSent»* (or in the case of **tableUpdateNotification()** *«eventReceived»*). A ***provider*** receives Request-Response Operations sent by a client (***referrer***).

A Module Type, defined in a <moduleType> element, lists the Operations that will be implemented by the Module (i.e. Operations defined by the provided Service(s)). Each implemented Operation is listed (in this case either as <requestReceived> XML elements or as <eventReceived> XML elements) within an <operations> element. Each Operation lists its input and output parameters as either <input> and <output> XML elements.

The <componentImplementation> element also includes a list of Operation Link elements, one for each implemented Service Operation implemented by the ASC and one for each Module-to-Module Operation implemented. Each Operation Link specifies the link source and destination. In the present case, the Operation Links comprise <requestLink> elements, which name the *SQL* Service as the source (<clients> XML element) and name a code implementation operation (within the Module Instance) as the destination (<server> XML element), and an <eventLink> element which is reversed since the Event Operation is sent by the Module (using <senders> and <receivers> XML elements).

Listing 2 sqlServer Component Implementation XML

```
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
          componentDefinition="sqlServer">
  <use library="ECOA.SQL"/>
  <moduleType name="sqlServer_modMain_t" hasUserContext="false"
                                         hasWarmStartContext="false">
    <operations>
    <requestReceived name="getConnection">
      <input  name="dataSource" type="ECOA.SQL:String"/>
      <input  name="userName" type="ECOA.SQL:NameString"/>
      <input  name="password" type="ECOA.SQL:NameString"/>
      <output name="connection" type="ECOA.SQL:ConnectionId"/>
      <output name="requestStatus" type="ECOA.SQL:Error"/>
    </requestReceived>
    <requestReceived name="endConnection">
      <input name="connection" type="ECOA.SQL:ConnectionId"/>
      <output name="requestStatus" type="ECOA.SQL:Error"/>
    </requestReceived>
    <requestReceived name="SQLExecute">
      <input name="connection" type="ECOA.SQL:ConnectionId"/>
      <input name="SQLString" type="ECOA.SQL:String"/>
      <output name="transaction" type="ECOA.SQL:ResultSetId"/>
    </requestReceived>
    <requestReceived name="getResult">
      <input name="connection" type="ECOA.SQL:ConnectionId"/>
      <input name="transaction" type="ECOA.SQL:ResultSetId"/>
      <output name="requestStatus" type="ECOA.SQL:Error"/>
    </requestReceived>
        :
        : (snipped)
        :
    <eventSent name="tableUpdateNotification">
          <input name="tableName" type="ECOA.SQL:NameString"/>
    </eventSent>
    </operations>
  </moduleType>

  <moduleImplementation   name="sqlServer_modMain"
          moduleType="sqlServer_modMain_t"
          language="C" />
```

```
  <moduleInstance name="sqlServer_modMainInst"
          implementationName="sqlServer_modMain" relativePriority="1"/>

   <requestLink>
    <clients>
      <service instanceName="SQL" operationName="getConnection"/>
    </clients>
    <server>
      <moduleInstance instanceName="sqlServer_modMainInst"
          operationName="getConnection"/>
    </server>
  </requestLink>
   <requestLink>
    <clients>
      <service instanceName="SQL" operationName="endConnection"/>
    </clients>
    <server>
      <moduleInstance instanceName="sqlServer_modMainInst"
          operationName="endConnection"/>
    </server>
  </requestLink>
   <requestLink>
    <clients>
      <service instanceName="SQL" operationName="SQLExecute"/>
    </clients>
    <server>
      <moduleInstance instanceName="sqlServer_modMainInst"
          operationName="SQLExecute"/>
    </server>
  </requestLink>
   <requestLink>
    <clients>
      <service instanceName="SQL" operationName="getResult"/>
    </clients>
    <server>
      <moduleInstance instanceName="sqlServer_modMainInst"
          operationName="getResult"/>
    </server>
  </requestLink>
        :
        : (snipped)
        :
  <eventLink>
    <senders>
      <moduleInstance instanceName="sqlServer_modMainInst"
          operationName="tableUpdateNotification"/>
    </senders>
    <receivers>
      <service  instanceName="SQL" operationName="tableUpdateNotification"/>
    </receivers>
  </eventLink>
</componentImplementation>
```

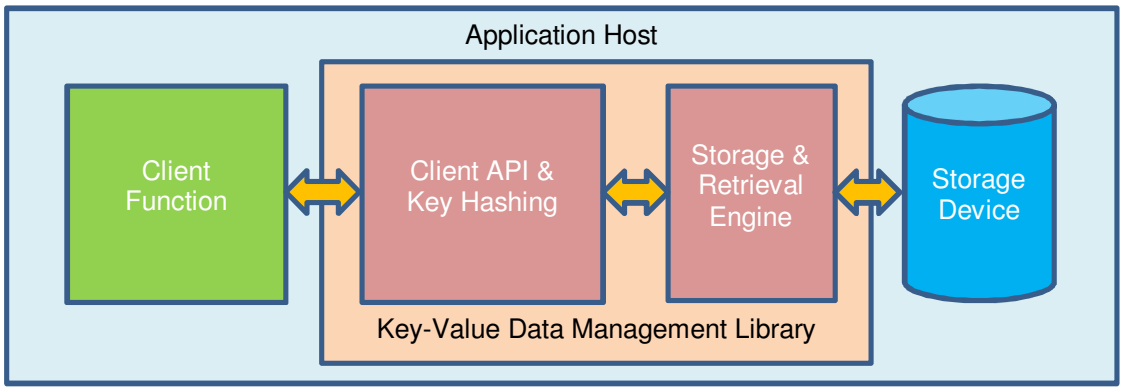
8.2 Rapid-Access (Indexed) Data Servers

Indexed data servers use a single atomic[7] value to index into a data space. A hashing algorithm is applied to reduce a key value (which may be of a compound data type) to an index value. The data Value (which can also be of a compound data type is then stored into the indexed data space, or recovered from it. The reduction of data location to a simple indexing into a data space means that data storage and retrieval is likely to be far faster than when parsing a location criterion such as with SQL.

An indexed, or "Key-Value", data server (since the data is presented as a Key and a Value) would tend not to be used across a network, but rather as a way of managing large amounts of data within an application program. A Key-Value data server could be thought of as a large indexed array of data with optimized, efficient, insertion, search and retrieval functionality, with the benefit of persistence when the data is stored to, say, a disc drive. Indeed, with optimal use of data caching in memory, data retrieval speeds could approach those of simple memory-to-memory (e.g. array reference) transfers.

The major processing element of a Key-Value data server implementation will be the conversion of Keys into hash value indices into the data space.

Figure 25 Typical Key-Value Data Server Configuration



8.2.1 Advantages:

Indexed data servers can provide:

- Rapid data storage and retrieval.
- Use of compound/record data types for both keys and values (e.g. via key hashing).

Potential for easy, fast, no-disc implementation for transient data.

8.2.2 Disadvantages:

In any hash-based indexing scheme it is possible that a given hash value can be generated by multiple input keys. In such cases, the consequences must be clear to the user, whether the data server will simply overwrite existing stored data whenever a duplicate key-hash is generated, or whether multiple entries are stored and retrieved. Storage and retrieval times for data can therefore be affected.

Similarly, search and retrieve operations based on a partial, or wide-field, key value (should such be supported by the data server implementation) will require client-side search overhead to distinguish between multiple data values returned by the server. For instance, if the key is a record containing a date and a time field, and the data server permits retrieval of a list of values stored against keys containing a particular date irrespective of the time, the client would need to process/search that returned list to isolate a particular record/entry.

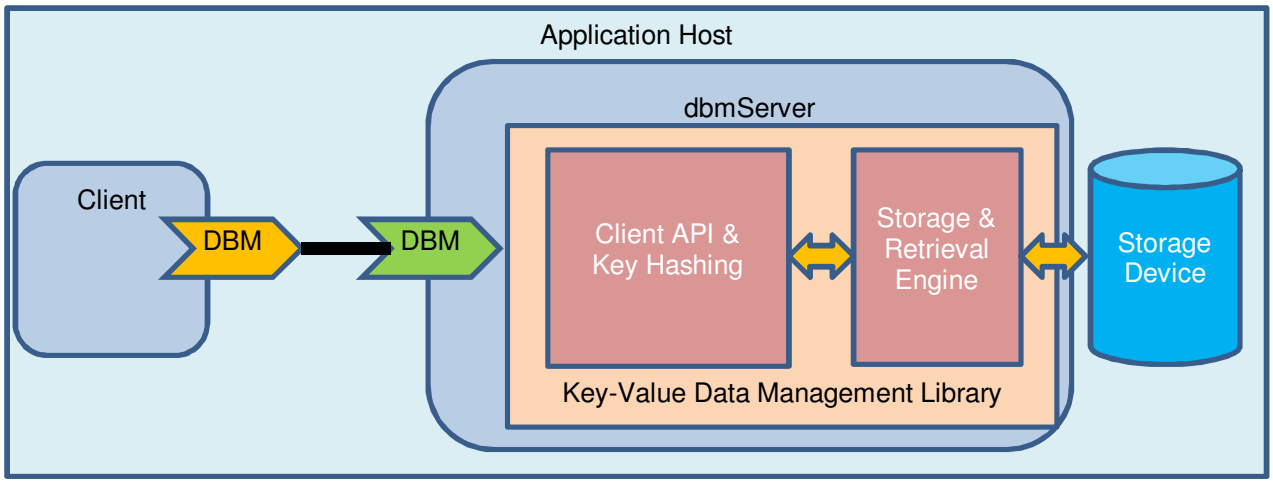
---

[7] "atomic" in the sense of being "indivisible".

8.2.3 An ECOA Key-Value Data Management Service

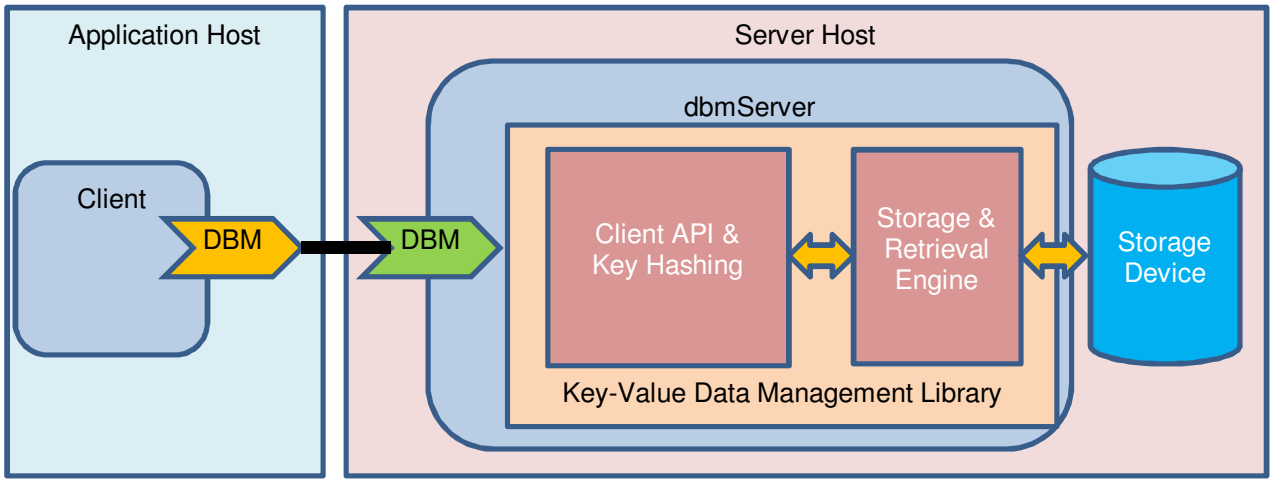
In the ECOA domain, we can envisage a practical translation of the arrangement of Figure 25 into that of Figure 26:

Figure 26 ECOA Key-Value Data Server Configuration



Here, the data server itself is hidden within an ECOA *dbmServer* ASC. All traffic between Client(s) and Server is now in the ECOA domain, so may physically invoke the ELI network protocol if the Client is deployed to the different host than the Server (Figure 27) (at the expense of some performance as network delays will be invoked).

Figure 27 ECOA Key-Value Data Server Configuration (Multiple Hosts)



8.2.3.1 Requirements

The following is a set of base requirements for an ECOA Service that can be used to provide access to an indexed, Key-Value, data server, named "*DBM*" (for Data Base Manager)

1) The DBM Service should allow an ECOA client ASC to connect to (access) a nominated (named) data server.
2) The DBM Service should allow an ECOA client ASC to create and destroy indexed (by a Key) data items on a data server that it is connected to.
3) The DBM Service should allow an ECOA client ASC to store to, search for, and retrieve data from the data server.
4) The DBM Service should allow an ECOA client ASC to pre-load data taken from an XML data file.
5) The DBM Service should allow an ECOA client ASC to save the current contents of the data server to an XML data file.

6) The DBM Service should allow an ECOA client ASC to interrogate the storage and content limits of the data server.

8.2.3.2 Required Operations

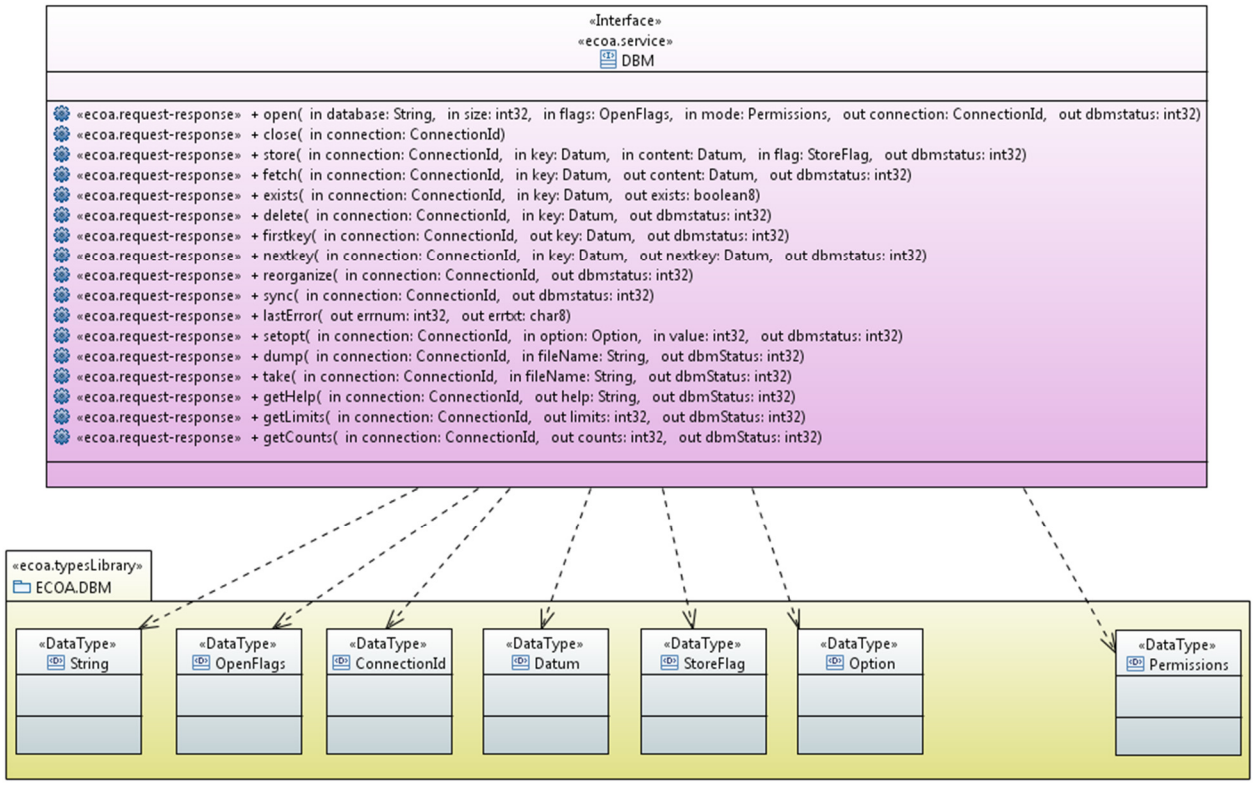
From para. 8.2.3.1, the minimum set of Service Operations is then:

1) Open (get) a connection to a data server.
2) Close (end) a connection.
3) Store/fetch data according to the value of a Key.
4) Fetch all Key items sequentially.
5) Load (take) or save (dump) data contents from/to an XML file.
6) Manage the data server limits and meta-data.

8.2.3.3 Service Definition

Largely adopting the API set of GNU *gdbm* (itself based on the Unix *dbm*) (ref. *[dbm], [gdbm]*), which will give a certain familiarity to programmers writing client code, and augmenting with other potentially useful operations, we get the Service Definition depicted in Figure 28. For clarity of purpose, the details of the actual data types, all of which are defined in the *ECOA.DBM* Types Library, are omitted from this diagram. Again, the Service is defined in terms of ECOA Request-Response Operations. Each operation is described after the diagram.

Figure 28 ECOA Key-Value Service Definition (as a UML Interface Class)



8.2.3.3.1 open(…)

This ECOA Request-Response Operation will open (create) a connection to a named *database*. A unique Connection Identifier (*connection*) will be returned, together with a *dbmStatus* code. If a file-based underlying storage mechanism is employed, then the database file will be opened according to the *flags* specified (for read, write, etc.), and created (if necessary) with the *permissions* specified.

8.2.3.3.2 close(…)

This ECOA Request-Response Operation will close (end) the identified *connection*.

8.2.3.3.3 store(…)

This ECOA Request-Response Operation will store the data *content* to the database at *connection*, indexed for retrieval using the given *key*, and return a *dbmStatus* code

8.2.3.3.4 fetch(…)

This ECOA Request-Response Operation will look for an entry indexed by *key*, in the database at *connection*. If an entry is found, the Operation will return the data as *content*. A *dbmStatus* code (which may indicate no key/data found) is always returned.

8.2.3.3.5 exists(…)

This ECOA Request-Response Operation will look for an entry indexed by *key*, in the database at *connection*, and will return *exists* set either "TRUE" or "FALSE" as appropriate. A *dbmStatus* code is also returned.

8.2.3.3.6 delete(…)

This ECOA Request-Response Operation will delete any entry indexed by *key*, from the database at *connection*, and return a *dbmStatus* code

8.2.3.3.7 firstkey(…)

This ECOA Request-Response Operation will return the "first" *key* found in the database at *connection*, and return a *dbmStatus* code. There is no correlation between the order in which key value are returned by **firstkey(…)** and **nextkey(…)** and the order entries were made, nor any form of sort order of (unhashed) key values. It is only guaranteed that successive calls to **nextkey(…)**, after an initial call to **firstkey(…)**, will return every key in the database once.

8.2.3.3.8 nextkey(…)

This ECOA Request-Response Operation will return the "next" *key* found in the database at *connection*, and return a *dbmStatus* code.

8.2.3.3.9 reorganize(…)

This ECOA Request-Response Operation will, if appropriate to the underlying implementation, reorganize the database at *connection* so as to minimize the size of the database file (e.g. after a large number of entry deletions), and return a *dbmStatus* code.

8.2.3.3.10 sync(…)

This ECOA Request-Response Operation will, if appropriate to the underlying implementation, ensure that the database at *connection* has flushed all internal buffers and written all data to the database file, and return a *dbmStatus* code.

8.2.3.3.11 lastError(…)

This ECOA Request-Response Operation will return, as *errtxt*, an English text explanation of the error code *errnum*, and return a *dbmStatus* code

8.2.3.3.12 setopt(…)

This ECOA Request-Response Operation will allow the client to control the (implementation specific) underlying database at *connection*, and return a *dbmStatus* code. Settings are likely to include such parameters as cache size, and whether the database automatically synchronizes to the database file.

8.2.3.3.13 dump(…)

This ECOA Request-Response Operation will save the entire database at *connection*, to the file *filename*, and return a *requestStatus* code. The file will be written as XML.

8.2.3.3.14 take(…)

This ECOA Request-Response Operation will read from the file *filename*, and execute the statements contained, on the database at *connection*, and return a *requestStatus* code. The file will be expected to contain XML.

8.2.3.3.15 getHelp(…)

This ECOA Request-Response Operation will return (in parameter *help*) a text string of useful "help" information from the database at *connection*, and return a *dbmStatus* code

8.2.3.3.16 getLimits(…)

This ECOA Request-Response Operation will return, as *limits*, a list of limit values relevant to the database at *connection*, and return a *dbmStatus* code.

The list will include:

- the maximum length of an ECOA.DBM.String;
- the maximum length of an ECOA.DBM.NameString;
- the maximum size of an ECOA.DBM.Datum.

8.2.3.3.17 getCounts(…)

This ECOA Request-Response Operation will return, as *counts*, a list of current values relevant to the database at *connection*, and return a *dbmStatus* code.

The list will include:

- ***TBD***.

8.2.3.4 Service Definition (XML)

The following are a few lines of the formal ECOA Service Definition XML for this Service. The entire Service Definition is not given here for brevity, but it will be seen that each Operation of the Service is defined using the data types depicted in the UML diagram above (Figure 28).

In ECOA, a Service Definition XML comprises a `<serviceDefinition>` XML element composed of an `<operations>` list XML element. Each Operation provided by the Service is listed (in this case as `<requestresponse>` XML elements) within the `<operations>` element. Each Operation lists its input and output parameters as either `<input>` and `<output>` XML elements.

Listing 3 DBM Service Definition XML

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">
  <use library="ECOA.DBM"/>
  <operations>
    <requestresponsename="open">
      <input  name="database"  type="ECOA.DBM:String"/>
      <input  name="size"  type="int32"/>
      <input  name="flags"  type="ECOA.DBM:OpenFlags"/>
      <input  name="mode"  type="int32"/>
      <output name="connection"  type="ECOA.DBM:ConnectionId"/>
      <output name="dbmStatus"  type="int32"/>
    </requestresponse>
```

```
    <requestresponsename="close">
      <input  name="connection"  type="ECOA.DBM:ConnectionId"/>
    </requestresponse>
    <requestresponsename="store">
      <input  name="connection"  type="ECOA.DBM:ConnectionId"/>
      <input  name="key"  type="ECOA.DBM:Datum"/>
      <input  name="content"  type="ECOA.DBM:Datum"/>
      <input  name="flag"  type="ECOA.DBM:StoreFlag"/>
      <output name="dbmStatus"  type="int32"/>
    </requestresponse>
       :
       : (snipped)
       :
  </operations>
</serviceDefinition>
```

The ECOA XML files for the example Services and ASCs discussed in this document are available from the ECOA website (ref. *[CODE]*).

8.2.4 An ECOA Key-Value Data Service Provider ASC

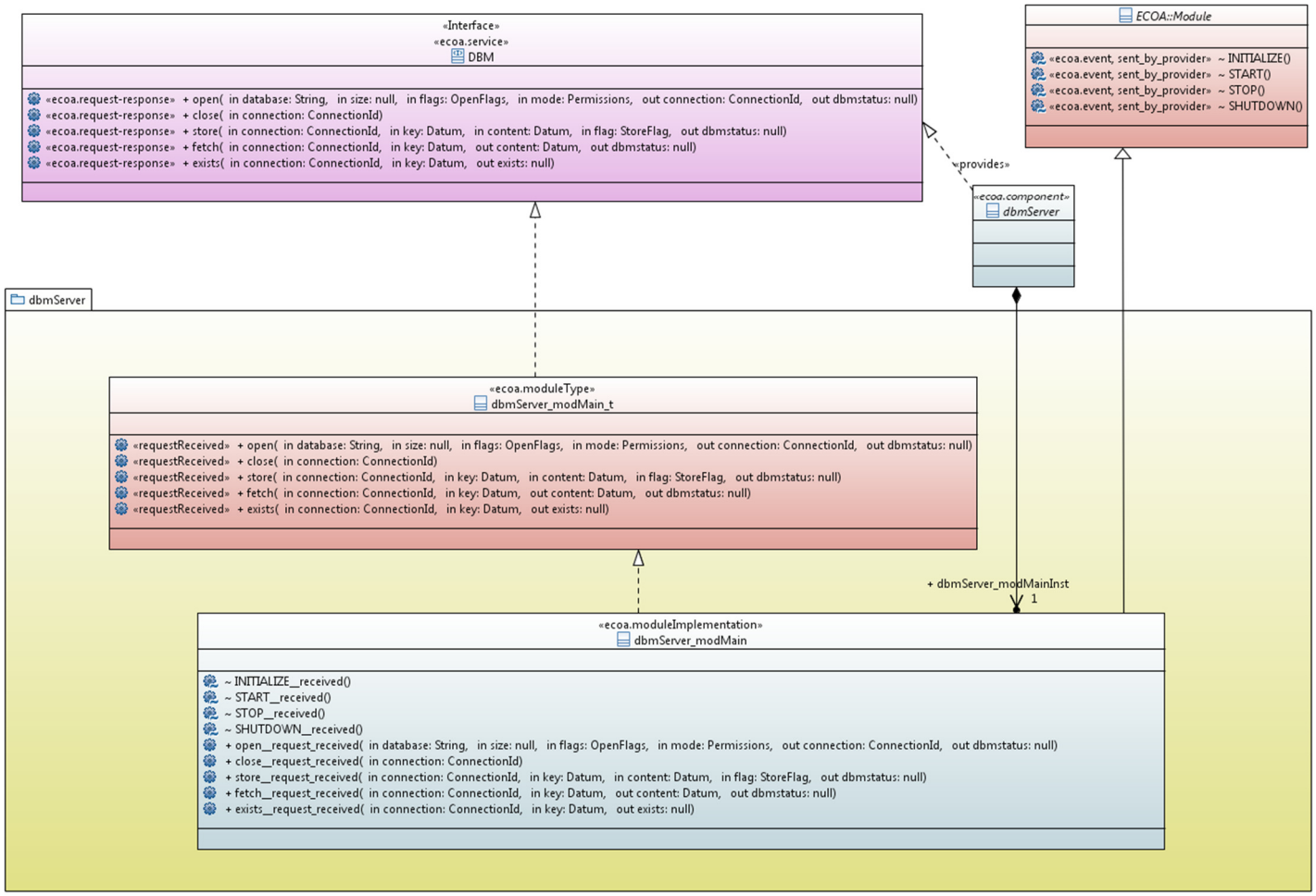
For the present purposes, the defined ECOA DBM Service will be provided by a self-contained *dbmServer* ECOA ASC, described in UML in Figure 29, and defined by the ECOA Component Implementation XML that follows (Listing 4).

For brevity within this document, only the first few Operations are shown in each of the UML entities in the diagram. The complete set of Service Operations was shown in Figure 28 and need not be repeated. Likewise, the (ASC) Component Implementation XML following has been snipped where repetition of similar structures, one for each operation, has been curtailed.

Referring to Figure 29, the *dbmServer* ECOA ASC, represented as an UML class and stereotyped *«ecoa.component»*, ***provides*** the *DBM* *«ecoa.service»* - indicated by the UML realization relationship (closed-headed dashed arrow, stereotyped *«provides»*) – by being ***composed of*** the *«ecoa.moduleImplementation»* *dbmServer_modMain*. At runtime, the single specific instance of the *dbmServer_modMain* implementation is named *dbmServer_modMainInst* – as indicated by the decorations on the ***composed of*** (or "***aggregation***") relationship arrow.

The *dbmServer_modMain* implementation is itself a realization of two interface definitions, one defining the lifecycle operations that all ECOA Modules must provide and here depicted as the UML Interface class *ECOA.Module*, and the other being the Module Type definition ***derived from*** (or "***specialized***" from) the *DBM* ECOA Service, namely the *«ecoa.moduleType»* *dbmService_modMain_t*. Thus *dbmService_modMain_t* exports all the Module Operations defined by the *DBM* Service, but they are now stereotyped as *«requestReceived»* because they are Operations on the ***provider*** Module Type. Similarly, the Operations appear again on the Module Implementation class (*dbmService_modMain*), along with the (implementation of) Operations from the *ECOA::Module* abstract class.

Figure 29 dbmServer ASC Design (as UML Class Diagram)



8.2.4.1 Component Implementation XML

In ECOA, a Component Implementation XML formally defines the construction of an ASC, comprising a `<componentImplementation>` XML element composed of one or more `<moduleType>`, `<moduleImplementation>`, and `<moduleInstance>` XML elements. Each `<moduleImplementation>` and `<moduleInstance>` element expresses the realization of a Module Type and the instantiation (at runtime) of the Module Implementation respectively.

A Module Type, defined in a `<moduleType>` element, lists the Operations that will be implemented by the Module (i.e. Operations defined by the provided Service(s)). Each implemented Operation is listed (in this case as `<requestReceived>` XML elements) within an `<operations>` element. Each Operation lists its input and output parameters as either `<input>` and `<output>` XML elements.

The `<componentImplementation>` element also includes a list of Operation Link elements, one for each implemented Service Operation implemented by the ASC and one for each Module-to-Module Operation implemented. Each Operation Link specifies the link source and destination. In the present case, the Operation Links are all `<requestLink>` elements, which all name the *DBM* Service as the source (`<clients>` XML element) and name a code implementation operation (within the Module Instance) as the destination (`<server>` XML element).

Listing 4 dbmServer Component Implementation XML

```
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
          componentDefinition="dbmServer">
  <use library="ECOA.DBM"/>
  <moduleType name="dbmServer_modMain_t" hasUserContext="true"
                                         hasWarmStartContext="false">
    <operations>
      <requestReceived name="open">
        <input  name="database"  type="ECOA.DBM:String"/>
        <input  name="size"  type="int32"/>
        <input  name="flags"   type="ECOA.DBM:OpenFlags"/>
        <input  name="mode"  type="int32"/>
        <output name="connection"  type="ECOA.DBM:ConnectionId"/>
        <output name="dbmStatus"  type="int32"/>
      </requestReceived>
      <requestReceived name="close">
        <input  name="connection"  type="ECOA.DBM:ConnectionId"/>
      </requestReceived>
      <requestReceived name="store">
        <input  name="connection"  type="ECOA.DBM:ConnectionId"/>
        <input  name="key"   type="ECOA.DBM:Datum"/>
        <input  name="content" type="ECOA.DBM:Datum"/>
        <input  name="flag"  type="ECOA.DBM:StoreFlag"/>
        <output name="dbmStatus"  type="int32"/>
      </requestReceived>
      <requestReceived name="fetch">
        <input  name="connection"  type="ECOA.DBM:ConnectionId"/>
        <input  name="key"   type="ECOA.DBM:Datum"/>
        <output name="content" type="ECOA.DBM:Datum"/>
        <output name="dbmStatus"  type="int32"/>
      </requestReceived>
      <requestReceived name="exists">
        <input  name="connection"  type="ECOA.DBM:ConnectionId"/>
        <input  name="key"   type="ECOA.DBM:Datum"/>
        <output name="exists"  type="boolean8"/>
      </requestReceived>
        :
        : (snipped)
        :
    </operations>
  </moduleType>

  <moduleImplementation  name="dbmServer_modMain"
          moduleType="dbmServer_modMain_t" language="C" />

  <moduleInstance name="dbmServer_modMainInst"
          implementationName="dbmServer_modMain" relativePriority="1"/>

  <requestLink>
    <clients>
      <service instanceName="DBM" operationName="open"/>
    </clients>
    <server>
      <moduleInstance instanceName="dbmServer_modMainInst" operationName="open"/>
    </server>
  </requestLink>
```

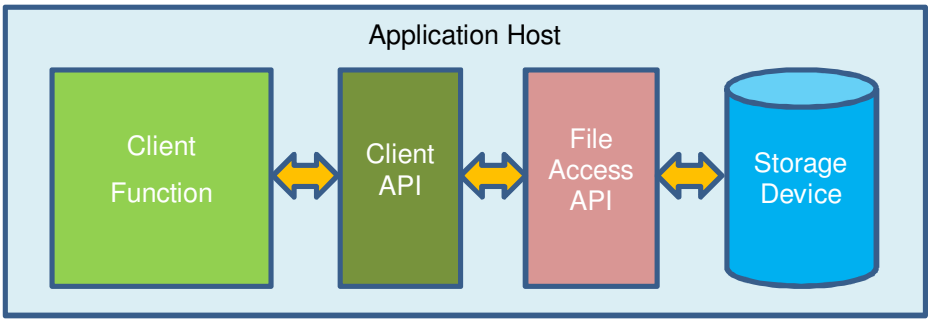
```
    <requestLink>
      <clients>
        <service instanceName="DBM" operationName="close"/>
      </clients>
      <server>
        <moduleInstance instanceName="dbmServer_modMainInst"
            operationName="close"/>
      </server>
    </requestLink>
    <requestLink>
      <clients>
        <service instanceName="DBM" operationName="store"/>
      </clients>
      <server>
        <moduleInstance instanceName="dbmServer_modMainInst"
            operationName="store"/>
      </server>
    </requestLink>
    <requestLink>
      <clients>
        <service instanceName="DBM" operationName="fetch"/>
      </clients>
      <server>
        <moduleInstance instanceName="dbmServer_modMainInst"
            operationName="fetch"/>
      </server>
    </requestLink>
    <requestLink>
      <clients>
        <service instanceName="DBM" operationName="exists"/>
      </clients>
      <server>
        <moduleInstance instanceName="dbmServer_modMainInst"
            operationName="exists"/>
      </server>
    </requestLink>
        :
        : (snipped)
        :
  </componentImplementation>
```

8.3 File-based Data Servers

Sometimes it is more practical or convenient to create an application specific data server using a standard File Access API. Scenario 4.5 described one possible case, where graphic images are stored as tiles, one per file, that are loaded and rendered at runtime in an order dependent on the flight path of the air vehicle.

The application is then likely to be constructed in a manner similar to Figure 30, where the operating system specific *File Access API* code library is used by a *Client API* code library, which is in turn used by the *Client Function(s)* itself/themselves. Thus in the case of Scenario 4.5, the *Client API* would comprise functions working with client-space data objects (positions, orientations, tiles, etc.), whilst the *Client API* implementation would do the conversion into file-space data objects (file names, handles, indexing, etc.). The *Client Function* is therefore abstracted away from actual files, and independent of the operating system implementation.

Figure 30 Typical File-based Data Server Configuration



8.3.1 Advantages:

Creating a bespoke, application specific, file-based data server can provide:

- Maximally efficient persistent storage of application specific data;
- No need to translate between as-stored and as-used data formats;
- Easy addition of new stored data by adding additional files;

8.3.2 Disadvantages:

Conversely, a bespoke, application specific, file-based data server will be precisely that:

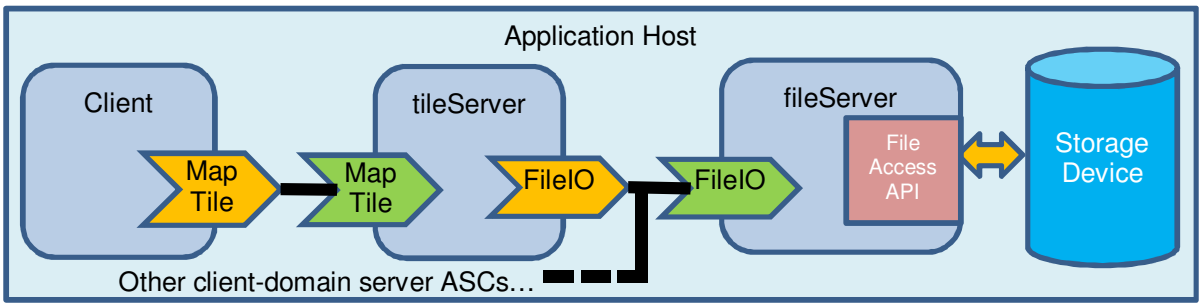
- Stored data (files) likely to be computing platform/operating system specific (due to characteristics such as data packing, endianness, etc.);
- Compounded addition of new data items to existing stored data structures and files;
- Need to organize/manage data files (into folders/directories);
- Tendency for the file organization/folder hierarchy requirements to evolve/change over time;

8.3.3 An ECOA File IO Data Access Service

In the ECOA domain, whilst the *Client API* of Figure 30 could be recast as an ECOA ASC directly (making it an ECOA "driver" ASC since it uses the operating system's File Access API), it might be more effective to realise the File Access API as an ECOA File Access ASC in its own right. Other application specific server ASCs could then re-use the common File Access ASC.

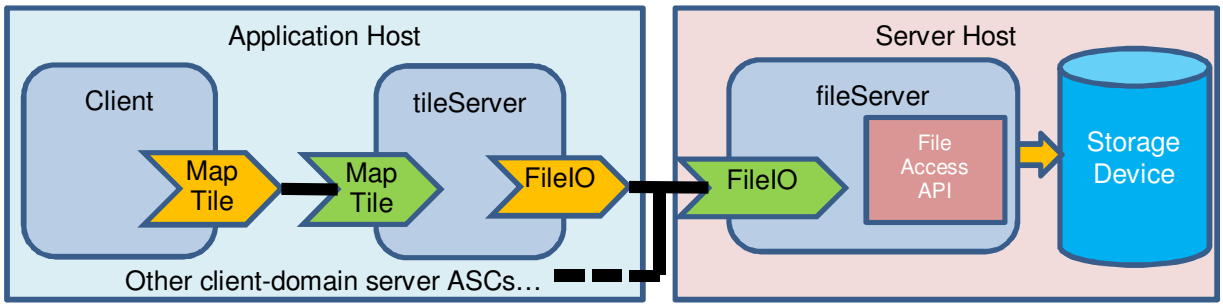
A Scenario 4.5 implementation might then become as Figure 31, where the *Client API* is transformed to an ECOA *tileServer* ASC providing a "*Map Tile*" Service, and the operating system specific File Access API is hidden within an ECOA *fileServer* ASC providing a "*FileIO*" Service.

Figure 31 ECOA File-based Data Server Configuration



In such a scheme, the ECOA *fileServer* might then be deployed "remotely" from some or all of the clients (*tileServer* etc.), either in a separate ECOA Protection Domain on the same computing platform, or physically remotely on a separate host computing platform (Figure 32) (at the expense of some performance as network delays will be invoked).

Figure 32 ECOA File-based Data Server Configuration (Multiple Hosts)



8.3.3.1 Requirements

The following is a set of base requirements for an ECOA Service that can be used to provide access to files, named "*FileIO*".

1) The FileIO Service should provide a simple, minimal, mapping to file system operations.
2) The FileIO Service should allow an ECOA client ASC to open and close files.
3) The FileIO Service should allow an ECOA client ASC to read and write text or binary from opened files.
4) The FileIO Service should allow an ECOA client ASC to create and delete files (subject to the underlying host platform).
5) The FileIO Service should allow an ECOA client ASC to change the current read/write position within an open file.
6) The FileIO Service should allow an ECOA client ASC to browse the file storage structure.

8.3.3.2 Required Operations

From para. 8.3.3.1, the minimum set of Service Operations is then:

1) Open/create a named file.
2) Close an open file.
3) Write/read data (text or binary) to/from an open file.
4) Delete a named file.
5) Set/get the read/write point to/as an indexed location within an open file.
6) "Open" a folder/directory listing.
7) Read/get entries in a folder/directory listing.
8) "Close" a folder/directory listing.

Additional operations may be required for more detailed manipulation and interrogation of files and folders, as application needs arise. These may include exposing, as ECOA Service Operations, detailed file access and management operations such as:

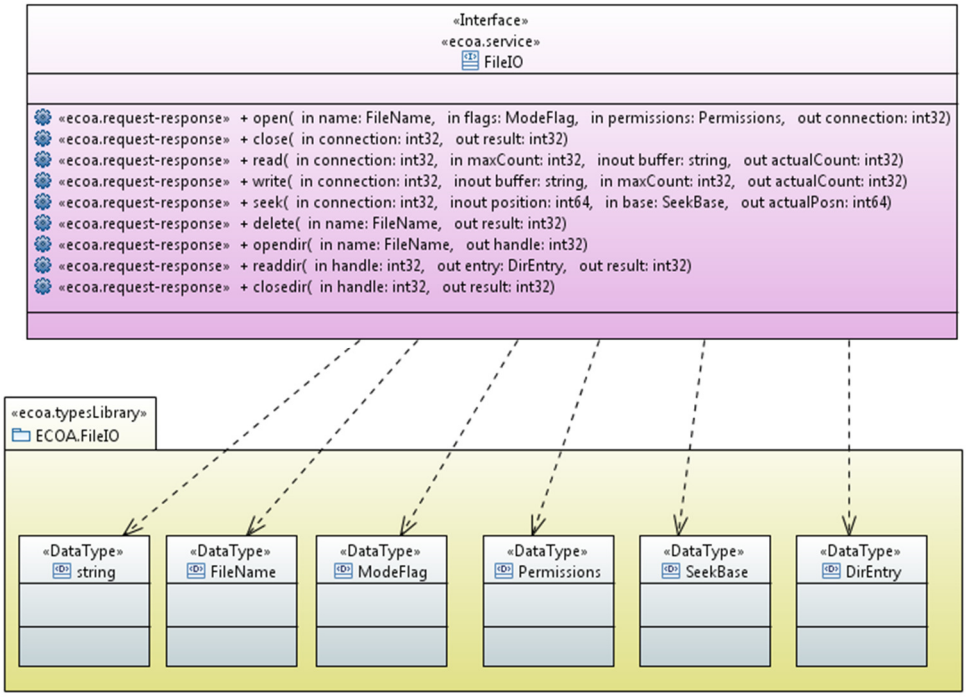
9) Folder creation/deletion.
10) Purging, flushing, and synchronizing data files and buffers.
11) Opening a new file using a file handle.
12) Folder/directory traversal ("walking").
13) Get/set file status and attributes.

etc.

8.3.3.3 Service Definition

Largely adopting (a small part of) the POSIX file API, which will give a certain familiarity to programmers writing client code, we get the Service Definition depicted in Figure 33. For clarity of purpose, the details of the actual data types, all of which are defined in the *ECOA.FileIO* Types Library, are omitted from this diagram. Each operation is described after the diagram.

Figure 33 ECOA File IO Service Definition (as a UML Interface Class)



8.3.3.3.1 open(…)

This ECOA Request-Response Operation will open (or create) a connection to a file named *name*. A unique Connection Identifier (*connection*) will be returned. The file will be opened according to the *flags* specified (for read, write, etc.), and created (if necessary) with the *permissions* specified.

8.3.3.3.2 close(…)

This ECOA Request-Response Operation will close the file identified by *connection*. If the Operation is successful, *result* is returned set to zero, otherwise it will be non-zero.

8.3.3.3.3 read(…)

This ECOA Request-Response Operation will read up to *maxCount* bytes from the file associated with *connection*. If the read is successful, the Operation will return the data in *buffer*, and the number of bytes read in *actualCount*.

actualCount will be returned equal to *maxCount* unless:

i) the end of the file is encountered, in which case it will be set to the number of bytes actually read up to the end of the file;
ii) if *maxCount* is greater than the maximum length of the *ECOA.FileIO:string* type (65536);
iii) if *connection* is invalid, in which case *actualCount* will be returned as zero (0).

8.3.3.3.4 write(…)

This ECOA Request-Response Operation will write up to *maxCount* bytes from *buffer*, to the file associated with *connection*. The Operation will return the number of bytes actually written in *actualCount*, which may be less than *maxCount* if an error occurs (such as running out of file space).

8.3.3.3.5 seek(…)

This ECOA Request-Response Operation will position the current read/write point *position* bytes from the *base* reference point within the file associated with *connection*.

base can be one of the values:

i) *ECOA.FileIO.SEEK_SET* which sets the read/write point *position* bytes from the beginning of the file;
ii) *ECOA.FileIO.SEEK_END* which sets the read/write point *position* bytes from the end of the file;
iii) *ECOA.FileIO.SEEK_CUR* which sets the read/write point *position* bytes from the current position of the read/write point.

The Operation returns the final (absolute) position (i.e. relative to the beginning of the file) of the read/write point as *actualPosn*.

To obtain the current (absolute) position of the read/write point, the Operation can be invoked with *position* = 0 and *base* = *ECOA.File:SEEK_CUR*.

8.3.3.3.6 delete(…)

This ECOA Request-Response Operation will delete file named *name* (if it exists). If the Operation is successful (i.e. the file is deleted), *result* is returned set to zero, otherwise it will be non-zero.

8.3.3.3.7 opendir(…)

This ECOA Request-Response Operation will “open” a directory contents listing for the file directory *name*. If the Operation is successful, *handle* provides a reference to that contents listing, otherwise *handle* will be returned zero.

A valid (non-zero) *handle* is required by the *readdir* and *closedir* Operations.

8.3.3.3.8 readdir(…)

This ECOA Request-Response Operation will return, as *entry*, one entry from the directory contents listing associated with *handle*. Successive invocations of the Operation will return each entry in turn. If no more entries are present (all have been read or the directory is empty), *entry* is returned with a null entry name.

If the Operation is successful (including a null entry return), *result* is returned set to zero, otherwise it will be non-zero.

8.3.3.3.9 closedir(…)

This ECOA Request-Response Operation will close the directory contents listing associated with *handle*. If the Operation is successful, *result* is returned set to zero, otherwise it will be non-zero.

8.3.3.4 Service Definition (XML)

The following are a few lines of the formal ECOA Service Definition XML for this Service. The entire Service Definition is not given here for brevity, but it will be seen that each Operation of the Service is defined using the data types depicted in the UML diagram above (Figure 33).

In ECOA, a Service Definition XML comprises a <serviceDefinition> XML element composed of an <operations> list XML element. Each Operation provided by the Service is listed (in this case as <requestresponse> XML elements) within the <operations> element. Each Operation lists its input and output parameters as either <input> and <output> XML elements.

Listing 5 FileIO Service Definition XML

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">
  <use library="ECOA.FileIO"/>
  <operations>
    <requestresponse name="open">
      <input name="Name" type="ECOA.FileIO:FileName"/>
      <input name="Flags" type="ECOA.FileIO:ModeFlag"/>
      <input name="Permissions" type="ECOA.FileIO:Permissions"/>
      <output name="Handle" type="int32"/>
    </requestresponse>
    <requestresponse name="close">
      <input name="Handle" type="int32"/>
      <output name="Result" type="int32"/>
    </requestresponse>
    <requestresponse name="read">
      <input name="Handle" type="int32"/>
      <input name="MaxCount" type="int32"/>
      <output name="Buffer" type="ECOA.FileIO:string"/>
      <output name="ActualCount" type="int32"/>
    </requestresponse>
       :
       : (snipped)
       :
  </operations>
</serviceDefinition>
```

The ECOA XML files for the example Services and ASCs discussed in this document are available from the ECOA website (ref. *[CODE]*).

8.3.4 An ECOA File IO Service Provider ASC

For the present purposes, the defined ECOA FileIO Service will be provided by a self-contained *fileServer* ECOA ASC, described in UML in Figure 34, and defined by the ECOA Component Implementation XML that follows (Listing 6).

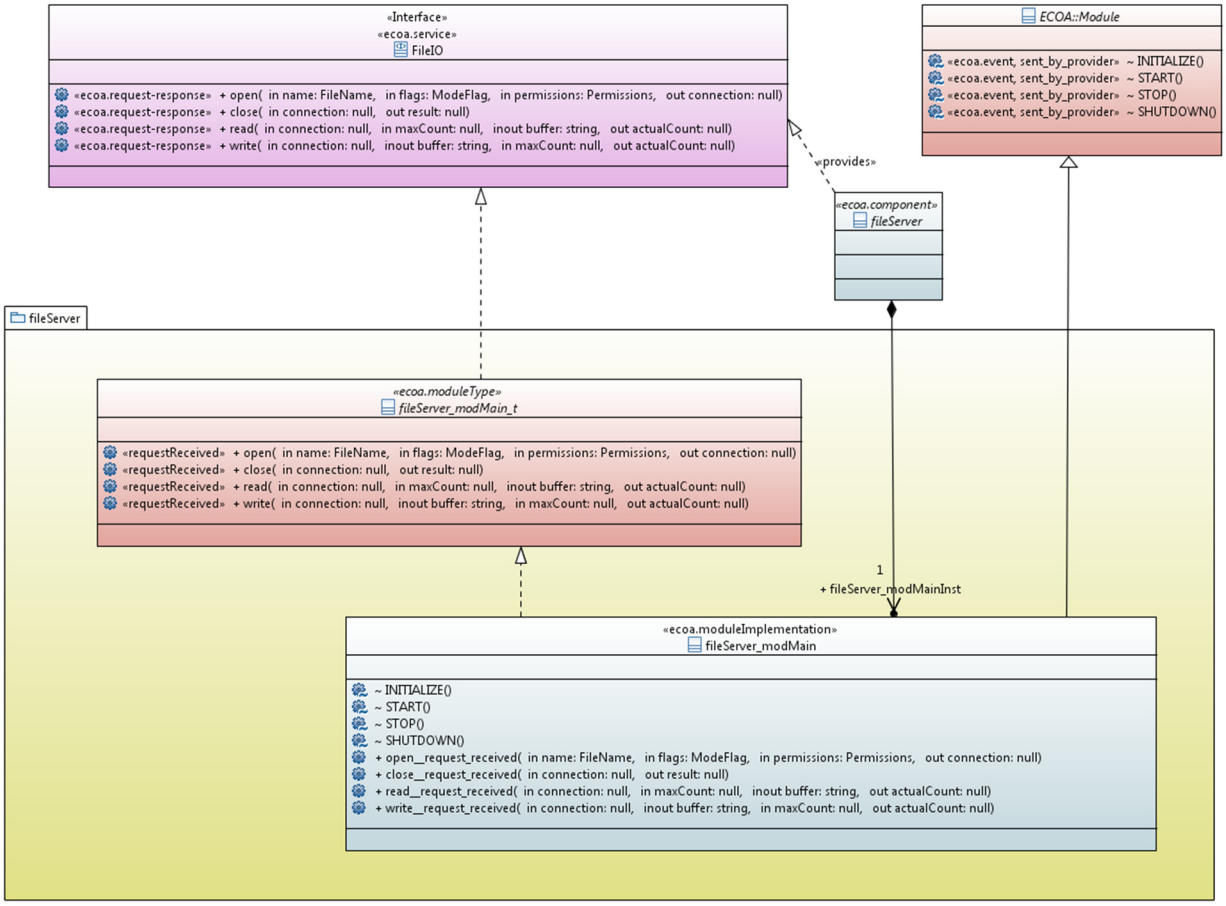
For brevity within this document, only the first few Operations may be shown in each of the UML entities in the diagram. The complete set of Service Operations was shown in Figure 33 and need not be repeated. Likewise, the (ASC) Component Implementation XML following has been snipped where repetition of similar structures, one for each operation, has been curtailed.

Referring to Figure 34, the *fileServer* ECOA ASC, represented as an UML class and stereotyped *«ecoa.component»*, ***provides*** the *FileIO «ecoa.service»* - indicated by the UML realization relationship (closed-headed dashed arrow, stereotyped *«provides»*) – by being ***composed of*** the

«ecoa.moduleImplementation» *fileServer_modMain*. At runtime, the single specific instance of the *fileServer_modMain* implementation is named *fileServer_modMainInst* – as indicated by the decorations on the ***composed of*** (or "***aggregation***") relationship arrow.

The *fileServer_modMain* implementation is itself a realization of two interface definitions, one defining the lifecycle operations that all ECOA Modules must provide and here depicted as the UML Interface class *ECOA.Module*, and the other being the Module Type definition ***derived from*** (or "***specialized***" from) the *FileIO* ECOA Service, namely the *«ecoa.moduleType»* *fileService_modMain_t*. Thus *fileService_modMain_t* exports all the Module Operations defined by the *FileIO* Service, but they are now stereotyped as *«requestReceived»* because they are Operations on the ***provider*** Module Type. Similarly, the Operations appear again on the Module Implementation class (*fileService_modMain*), along with the (implementation of) Operations from the *ECOA::Module* abstract class.

Figure 34 fileServer ASC Design (as UML Class Diagram)



8.3.4.1 Component Implementation XML

In ECOA, a Component Implementation XML formally defines the construction of an ASC, comprising a `<componentImplementation>` XML element composed of one or more `<moduleType>`, `<moduleImplementation>`, and `<moduleInstance>` XML elements. Each `<moduleImplementation>` and `<moduleInstance>` element expresses the realization of a Module Type and the instantiation (at runtime) of the Module Implementation respectively.

A Module Type, defined in a `<moduleType>` element, lists the Operations that will be implemented by the Module (i.e. Operations defined by the provided Service(s)). Each implemented Operation is listed (in this case as `<requestReceived>` XML elements) within an `<operations>` element. Each Operation lists its input and output parameters as either `<input>` and `<output>` XML elements.

The <componentImplementation> element also includes a list of Operation Link elements, one for each implemented Service Operation implemented by the ASC and one for each Module-to-Module Operation implemented. Each Operation Link specifies the link source and destination. In the present case, the Operation Links are all <requestLink> elements, which all name the *FileIO* Service as the source (<clients> XML element) and name a code implementation operation (within the Module Instance) as the destination (<server> XML element).

Listing 6 fileServer Component Implementation XML

```
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
          componentDefinition="fileServer">
  <use library="ECOA.FileIO"/>
  <moduleType name="fileServer_modMain_t" hasUserContext="true"
                                          hasWarmStartContext="false">
    <operations>
      <requestReceived name="open">
        <input name="name" type="ECOA.FileIO:FileName"/>
        <input name="flags" type="ECOA.FileIO:ModeFlag"/>
        <input name="permissions" type="ECOA.FileIO:Permissions"/>
        <output name="handle" type="int32"/>
      </requestReceived>
      <requestReceived name="close">
        <input name="handle" type="int32"/>
        <output name="result" type="int32"/>
      </requestReceived>
     <requestReceived name="read">
        <input name="handle" type="int32"/>
        <input name="maxCount" type="int32"/>
        <output name="buffer" type="ECOA.FileIO:string"/>
        <output name="actualCount" type="int32"/>
      </requestReceived>
      <requestReceived name="write">
        <input name="handle" type="int32"/>
        <input name="buffer" type="ECOA.FileIO:string"/>
        <input name="maxCount" type="int32"/>
        <output name="actualCount" type="int32"/>
      </requestReceived>
      <requestReceived name="seek">
        <input name="handle" type="int32"/>
        <input name="position" type="int64"/>
        <input name="base" type="ECOA.FileIO:SeekBase"/>
        <output name="actualPosn" type="int64"/>
      </requestReceived>
        :
        : (snipped)
        :
    </operations>
  </moduleType>

  <moduleImplementation  name="fileServer_modMain"
          moduleType="fileServer_modMain_t" language="C" />

  <moduleInstance name="fileServer_modMainInst"
          implementationName="fileServer_modMain" relativePriority="1"/>
```

```
    <requestLink>
      <clients>
        <service instanceName="FileIO" operationName="open"/>
      </clients>
      <server>
        <moduleInstance instanceName="fileServer_modMainInst"
            operationName="open"/>
      </server>
    </requestLink>
     <requestLink>
      <clients>
        <service instanceName="FileIO" operationName="close"/>
      </clients>
      <server>
        <moduleInstance instanceName="fileServer_modMainInst"
            operationName="close"/>
      </server>
    </requestLink>
    <requestLink>
      <clients>
        <service instanceName="FileIO" operationName="read"/>
      </clients>
      <server>
        <moduleInstance instanceName="fileServer_modMainInst"
            operationName="read"/>
      </server>
    </requestLink>
    <requestLink>
      <clients>
        <service instanceName="FileIO" operationName="write"/>
      </clients>
      <server>
        <moduleInstance instanceName="fileServer_modMainInst"
            operationName="write"/>
      </server>
    </requestLink>
          :
          : (snipped)
          :
  </componentImplementation>
```

8.4 Web Servers

With the explosion of web and cloud based computing, and now common-place distribution of data via web portals, including data such as tasking orders and mission plans in the military domain (cf. Scenario 1), it is but right that this document should look at a web access approach using ECOA constructs.

The usual form of a web access context is illustrated in Figure 35. A Data Server, often an SQL database, provides the data storage and retrieval element, but this is accessed via a Web Service application and HTTP Server. In addition to the Data Server Security mechanisms, a Firewall would be present in any system accessed from the Internet.

Authorised client applications make requests of the Data Server engine normally in one of two ways:

a) Scripting Engine

Direct HTTP access operates by the client invoking a server side scripting engine (e.g. ASP, PHP, JSP, Java Script, etc.) by requesting the relevant web document for the action the client wishes to have performed, which the scripting engine executes. Parameters are passed by the client as extensions to the web document request. Such a request might be:

http://www.myweather2.com/developer/forecast.ashx?uac=<accessCode>&query=51.368,0.123

which requests the weather forecast (web document *forecast.ashx*) for the location given (as a latitude and longitude) by the *query* parameter. The HTTP server at *www.myweather2.com* will access the web document, pass it to the script engine (in the role of the Web Server Application in Figure 35) which executes it for the location given, builds a response document (as an XML document for instance), and posts it back to the HTTP Server as the response to the web page request.

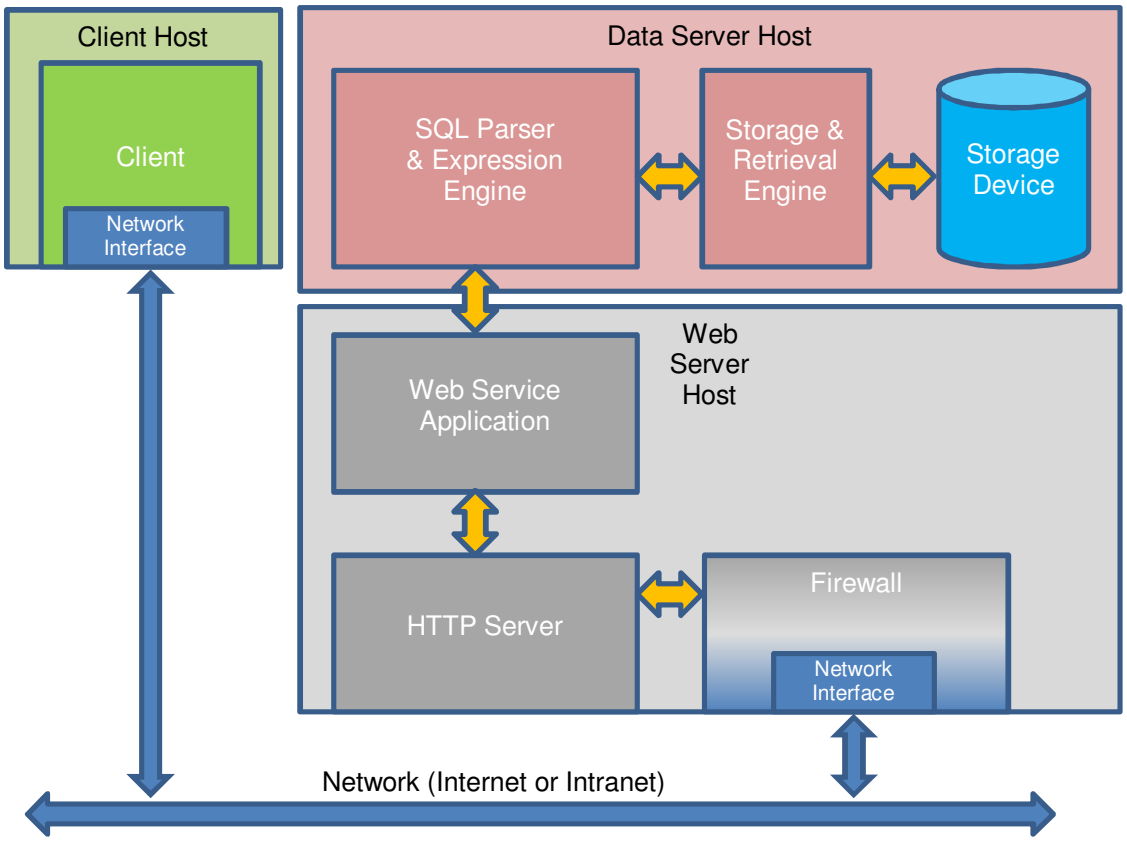
b) Data Exchange Protocol

The other form of web data access mechanism is to employ a data exchange protocol, such as SOAP (ref. [SOAP]). SOAP provides the "*definition of the XML-based information which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment.*" SOAP is a one-way message exchange paradigm ideally suited to employing TCP and HTTP as transport protocols, though more complex interaction patterns are built by combining one-way messaging, particularly request-response patterns including Remote Procedure Call. A very simple SOAP request package might be:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsd="http://www.w3.org/1999/XMLSchema"
      xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <add SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" >
      <a xsi:type="xsd:int">10</a>
      <b xsi:type="xsd:int">20</b>
    </add>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

which invokes an "add" RPC with the parameters "a" and "b" set to the values "10" and "20" respectively. The receiving network Server (whether HTTP, TCP, or whatever) passes the SOAP XML package to the SOAP execution engine (in the role of the Web Server Application in Figure 35) which parses the XML, extracts the request and parameters, performs the requested function, and packages and returns the result response (as a SOAP XML package) back to the network Server (e.g. as the HTTP request response or as a return TCP message).

Figure 35 Typical Web Service Data Server Configuration



8.4.1 Advantages:

Employing the internet and web as the communications infrastructure provides a very high degree of platform independence of client and service processing, whilst also providing the potential for access from anywhere in the world.

Web technology also opens the potential to use universally available client access tools such as web browsers, rather than having to deploy and use bespoke, proprietary, tooling.

8.4.2 Disadvantages:

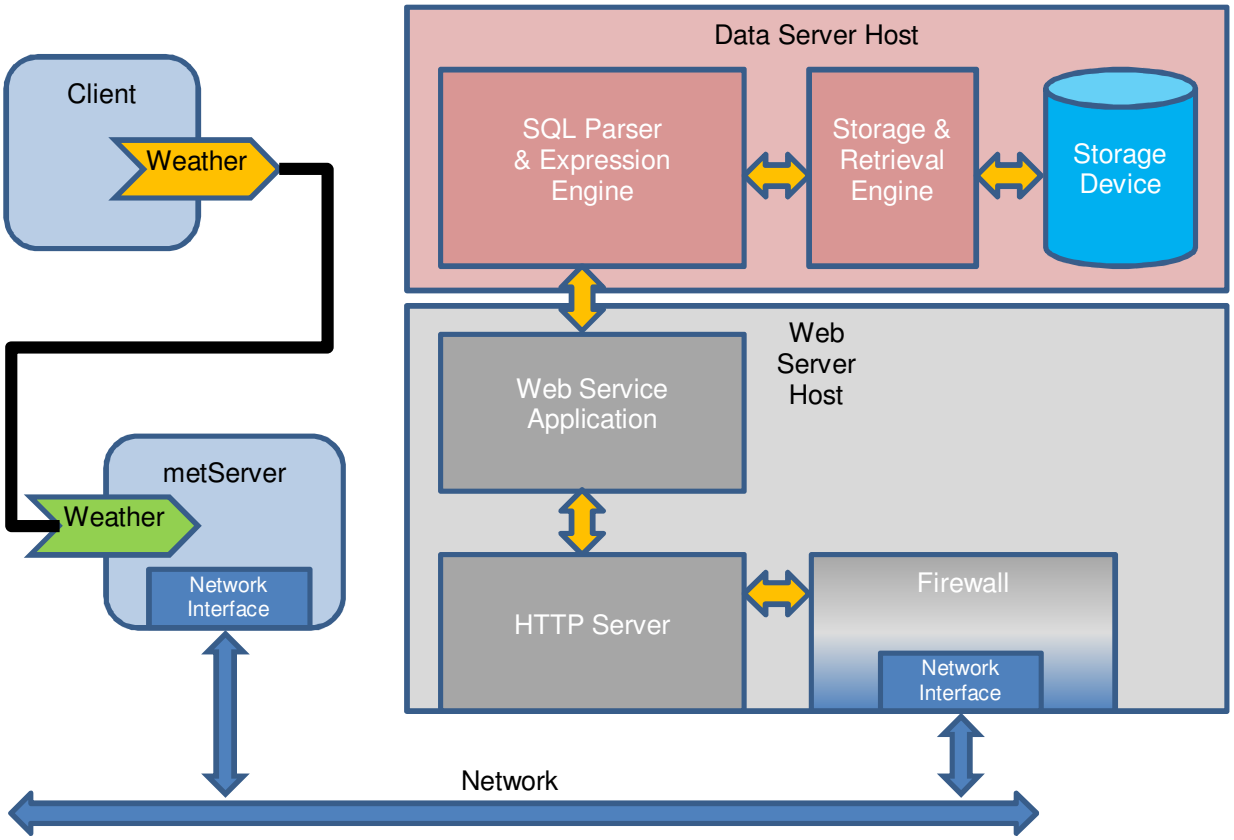
The downside of global accessibility is the potential for unauthorised access.

8.4.3 An ECOA Web Server Access Service

It is unlikely to be possible, and probably undesirable, to define a “one size fits all” generic web service interface as has been done for the other data server types. Instead, a particular service type example will be chosen, and an ECOA interface to that service type will be constructed, allowing room to implement different instances. Scenario 4.3 described one possible case, where a “Weather” service is used to provide weather data to a Navigation System.

A Scenario 4.3 implementation might then become as Figure 36, where an ECOA *metServer* ASC provides a “*Weather*” Service, hiding the specifics of accessing the web service from the client. The client is insulated from the details of accessing the web service, including the specific web service provider.

Figure 36 ECOA-ized Web Service Data Server Configuration



8.4.3.1 Requirements

The following is a set of base requirements for an example ECOA Weather Service, named "*Weather*".

1) The Weather Service should provide the current weather at a location.
2) The Weather Service should provide forecast weather at a location for a given future time.
3) The Weather Service should, as a minimum, allow locations to be expressed as latitude and longitude.
4) The Weather Service should provide weather data including, at least; temperature (maximum, minimum, current), wind direction and speed, air pressure, cloud cover, precipitation, and overall outlook/synopsis.
5) The Weather Service should provide forecast data for at least seven days ahead.

8.4.3.2 Required Operations

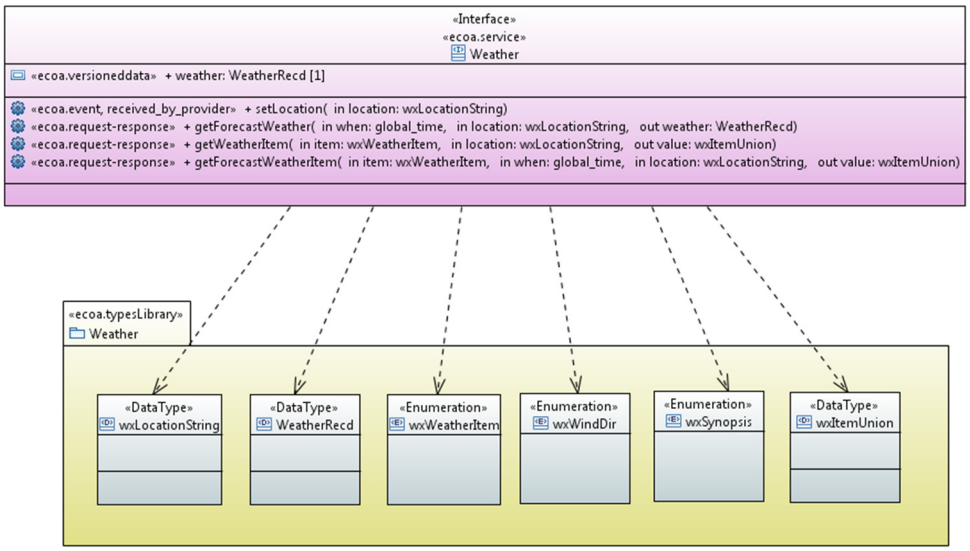
From para. 8.4.3.1, the minimum set of Service Operations might be:

1) Get current weather at location.
2) Get forecast weather at location for time.
3) Get current individual weather data item (e.g. temperature) at location.
4) Get forecast individual weather data item (e.g. temperature) at location at time.

8.4.3.3 Service Definition

Expanding this minimal set, we get the Service Definition depicted in Figure 37. For clarity of purpose, the details of the actual data types, all of which are defined in the `Weather` Types Library, are omitted from this diagram. In this case, the primary point of client access to the Service is to be implemented as an ECOA Version Data item (`weather`). Each operation is described after the diagram.

Figure 37 ECOA (Weather) Web Server Access Service Definition (as a UML Interface Class)



8.4.3.3.1 weather

This ECOA Versioned Data item will contain the current weather (as a *WeatherRecd*) at the location previously set using the **setLocation()** operation. If a location has not been set, the *weather* Versioned Data item will not be readable.

8.4.3.3.2 setLocation(…)

This ECOA Event Operation will set the current weather location to the *location* given. The *location* is given as a text string (data type *wxLocationString*), and will, as a minimum be accepted when in the form of a comma separated latitude-longitude pair, such as "51.354, 13.445". Other possibilities might include a location specified as a postcode or city name.

8.4.3.3.3 getForecastWeather(…)

This ECOA Request-Response Operation will return, as *weather*, the forecast weather at the *location* given for the time specified by *when*.

8.4.3.3.4 getWeatherItem(…)

This ECOA Request-Response Operation will return, as *value*, the current value of the weather data item specified by *item*, at the *location* given. If the *location* is given as NULL, then the location set by a previous **setLocation()** operation will be used.

item will be an enumeration specifier, with a unique value for each weather data item. *value* will be an ECOA *VariantRecord* type with a field for each data item type. For example:

```
getWeatherItem( MAXTEMP, berlin, &wxItem );
maxTemp = wxItem.u_param.MaxTempC;
getWeatherItem( WINDIR, berlin, &wxItem );
windDir = wxItem.u_param.WindDir;
```

8.4.3.3.5 getForecastWeatherItem(…)

This ECOA Request-Response Operation will return, as *value*, the forecast value of the weather data item specified by *item*, at the *location* given, at the time given by *when*.

8.4.3.4 Service Definition (XML)

The following listing is the formal ECOA Service Definition XML for this Service. It will be seen that each Operation of the Service is defined using the data types depicted in the UML diagram above (Figure 37).

In ECOA, a Service Definition XML comprises a `<serviceDefinition>` XML element composed of an `<operations>` list XML element. Each Operation provided by the Service is listed within the `<operations>` element. Each Operation lists its input and output parameters as either `<input>` and `<output>` XML elements (where relevant).

Listing 7 Weather Service Definition XML

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0" >
  <operations>
    <data name="weather" type="Weather:WeatherRecd"/>
    <event name="setLocation" direction="RECEIVED_BY_PROVIDER">
      <input  name="location" type="Weather:wxLocationString"/>
    </event>
    <requestresponse name="getForecastWeather">
      <input  name="when"     type="ECOA:global_time"/>
      <input  name="location" type="Weather:wxLocationString"/>
      <output name="weather"  type="Weather:WeatherRecd"/>
    </requestresponse>
    <requestresponse name="getWeatherItem">
      <input  name="item"     type="Weather:wxWeatherItem"/>
      <input  name="location" type="Weather:wxLocationString"/>
      <output name="weather"  type="Weather:WeatherRecd"/>
    </requestresponse>
    <requestresponse name="getForecastWeatherItem">
      <input  name="item"     type="Weather:wxWeatherItem"/>
      <input  name="when"     type="ECOA:global_time"/>
      <input  name="location" type="Weather:wxLocationString"/>
      <output name="weather"  type="Weather:WeatherRecd"/>
    </requestresponse>
  </operations>
</serviceDefinition>
```

The ECOA XML files for the example Services and ASCs discussed in this document are available from the ECOA website (ref. *[CODE]*).

8.4.4 An ECOA Web Server Access Provider ASC

For the present purposes, the defined ECOA Weather Service will be provided, as discussed in para. 8.4.3, by a self-contained *metServer* ECOA ASC, described in UML in Figure 38, and defined by the ECOA Component Implementation XML that follows (Listing 8).

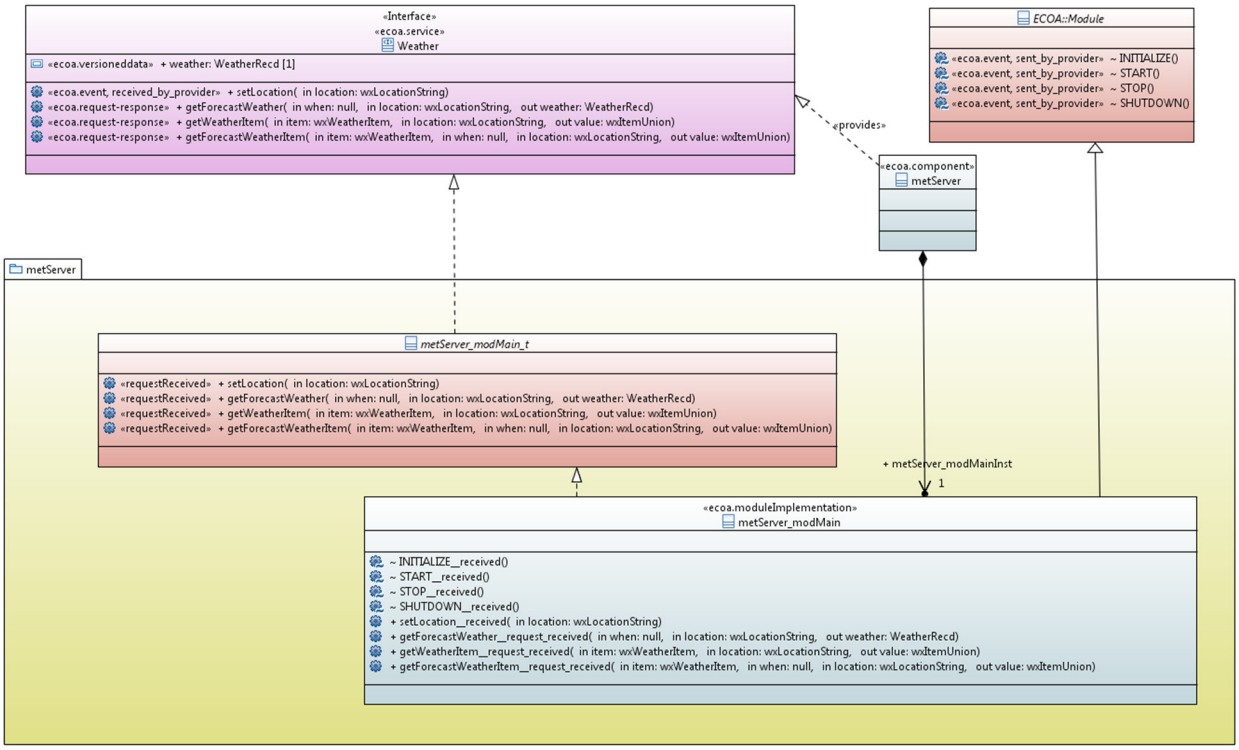
For brevity within this document, only the first few Operations may be shown in each of the UML entities in the diagram. The complete set of Service Operations was shown in Figure 37 and need not be repeated. Likewise, the (ASC) Component Implementation XML following has been snipped where repetition of similar structures, one for each operation, has been curtailed.

Referring to Figure 38, the *metServer* ECOA ASC, represented as an UML class and stereotyped *«ecoa.component»*, ***provides*** the *Weather «ecoa.service»* - indicated by the UML realization relationship (closed-headed dashed arrow, stereotyped *«provides»*) – by being ***composed of*** the *«ecoa.moduleImplementation» metServer_modMain*. At runtime, the single specific instance of the *metServer_modMain* implementation is named *metServer_modMainInst* – as indicated by the decorations on the ***composed of*** (or "***aggregation***") relationship arrow.

The *metServer_modMain* implementation is itself a realization of two interface definitions, one defining the lifecycle operations that all ECOA Modules must provide and here depicted as the UML Interface class *ECOA.Module*, and the other being the Module Type definition ***derived from*** (or "***specialized***" from) the

Weather ECOA Service, namely the *«ecoa.moduleType»* *metService_modMain_t*. Thus *metService_modMain_t* exports all the Module Operations defined by the *Weather* Service, but they are now stereotyped as *«requestReceived»* because they are Operations on the ***provider*** Module Type. Similarly, the Operations appear again on the Module Implementation class (*metService_modMain*), along with the (implementation of) Operations from the *ECOA::Module* abstract class.

Figure 38 metServer ASC Design (as UML Class Diagram)



8.4.4.1 Component Implementation XML

In ECOA, a Component Implementation XML formally defines the construction of an ASC, comprising a `<componentImplementation>` XML element composed of one or more `<moduleType>`, `<moduleImplementation>`, and `<moduleInstance>` XML elements. Each `<moduleImplementation>` and `<moduleInstance>` element expresses the realization of a Module Type and the instantiation (at runtime) of the Module Implementation respectively.

A Module Type, defined in a `<moduleType>` element, lists the Operations that will be implemented by the Module (i.e. Operations defined by the provided Service(s)). Each implemented Operation is listed (in this case as `<requestReceived>` XML elements) within an `<operations>` element. Each Operation lists its input and output parameters as either `<input>` and `<output>` XML elements.

The `<componentImplementation>` element also includes a list of Operation Link elements, one for each implemented Service Operation implemented by the ASC and one for each Module-to-Module Operation implemented. Each Operation Link specifies the link source and destination. In the present case, the Operation Links are all `<requestLink>` elements, which all name the *Weather* Service as the source (`<clients>` XML element) and name a code implementation operation (within the Module Instance) as the destination (`<server>` XML element).

Listing 8 metServer Component Implementation XML

```
<componentImplementation xmlns="http://www.ecoa.technology/implementation-2.0"
                         componentDefinition="metServer">

  <use library="Weather"/>

 <moduleType name="metServer_modMain_t" hasUserContext="true"
                                        hasWarmStartContext="false">
    <operations>
      <dataWritten name="weather" type="Weather:WeatherRecd"/>
      <eventReceived name="setLocation">
        <input  name="location" type="Weather:wxLocationString"/>
      </eventReceived>
      <requestReceived name="getForecastWeather">
        <input  name="when"     type="ECOA:global_time"/>
        <input  name="location" type="Weather:wxLocationString"/>
        <output name="weather"  type="Weather:WeatherRecd"/>
      </requestReceived>
      <requestReceived name="getWeatherItem">
        <input  name="item"     type="Weather:wxWeatherItem"/>
        <input  name="location" type="Weather:wxLocationString"/>
        <output name="value"    type="Weather:wxItemUnion"/>
      </requestReceived>
        <input  name="item"     type="Weather:wxWeatherItem"/>
        <input  name="when"     type="ECOA:global_time"/>
        <input  name="location" type="Weather:wxLocationString"/>
        <output name="value"    type="Weather:wxItemUnion"/>
      </requestReceived>
      <eventReceived name="Tick"/>
    </operations>
  </moduleType>

  <moduleImplementation   name="metServer_modMain"
                          moduleType="metServer_modMain_t" language="C" />

  <moduleInstance name="metServer_modMainInst"
                          implementationName="metServer_modMain"
                          relativePriority="1"/>

  <triggerInstance name="metServer_Ticker" relativePriority="2"/>

  <eventLink>
    <senders>
      <trigger instanceName="metServer_Ticker" period="300.0" />
    </senders>
    <receivers>
      <moduleInstance instanceName="metServer_modMainInst" operationName="Tick"/>
    </receivers>
  </eventLink>

   <dataLink>
     <writers>
       <moduleInstance instanceName="metServer_modMainInst"
                          operationName="weather"/>
     </writers>
     <readers>
```

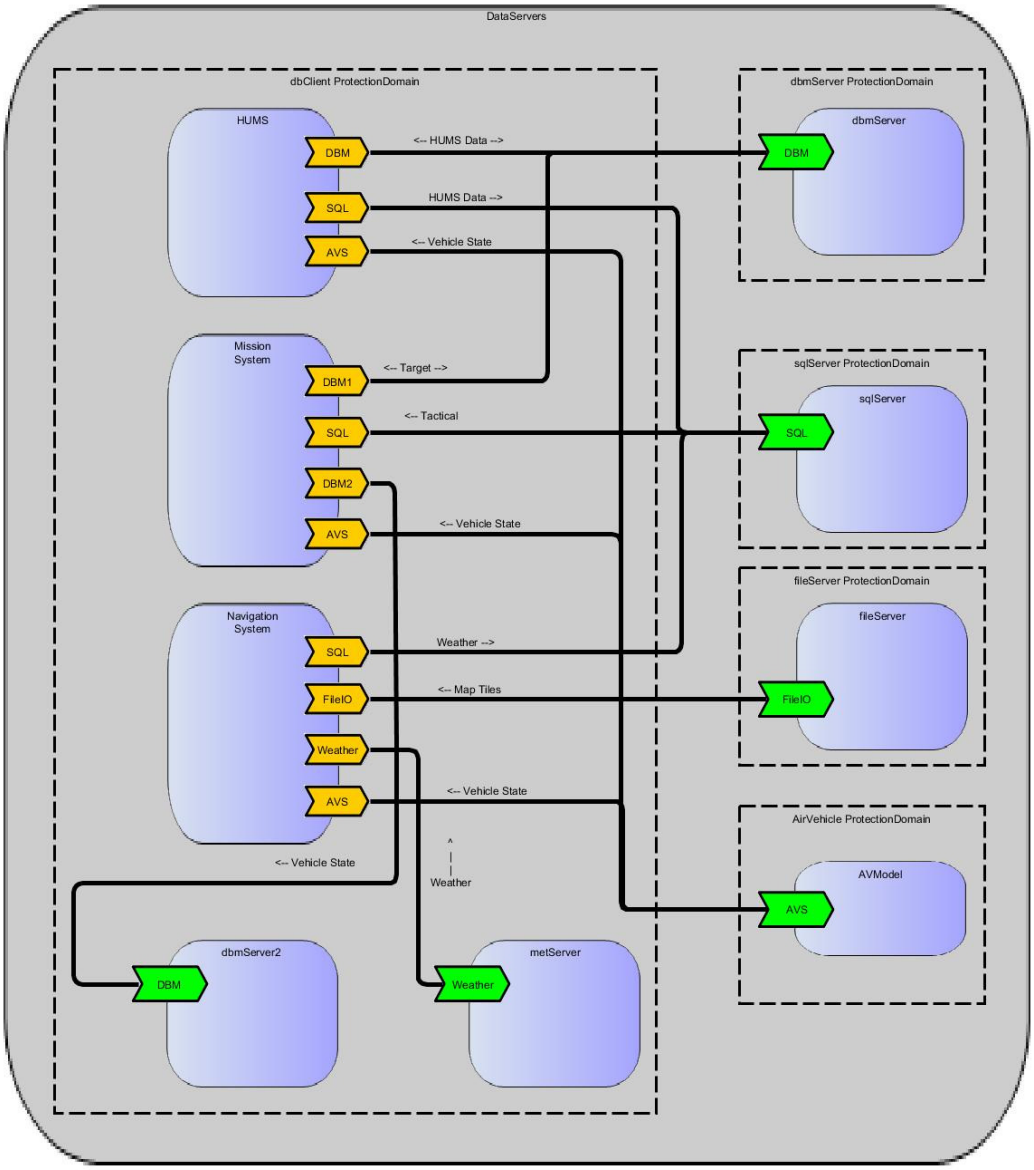
```
        <service instanceName="Weather" operationName="weather"/>
      </readers>
    </dataLink>

    <eventLink>
      <senders>
        <service instanceName="Weather" operationName="setLocation"/>
      </senders>
      <receivers>
        <moduleInstance instanceName="metServer_modMainInst"
                        operationName="setLocation"/>
      </receivers>
    </eventLink>

    <requestLink>
      <clients>
        <service instanceName="Weather" operationName="getForecastWeather"/>
      </clients>
      <server>
        <moduleInstance instanceName="metServer_modMainInst"
                        operationName="getForecastWeather"/>
      </server>
    </requestLink>

    <requestLink>
      <clients>
        <service instanceName="Weather" operationName="getWeatherItem"/>
      </clients>
      <server>
        <moduleInstance instanceName="metServer_modMainInst"
                        operationName="getWeatherItem"/>
      </server>
    </requestLink>

    <requestLink>
      <clients>
        <service instanceName="Weather" operationName="getForecastWeatherItem"/>
      </clients>
      <server>
        <moduleInstance instanceName="metServer_modMainInst"
                        operationName="getForecastWeatherItem"/>
      </server>
    </requestLink>

  </componentImplementation>
```

9 ECOA Data Server Demonstration

To demonstrate the concepts expressed in this document, an example hypothetical, and highly contrived, software system has been composed and implemented using the ECOA. The system, illustrated as an ECOA Assembly Diagram in Figure 39, has an instance of each of the four types of data server provider discussed, accessed by Mission System oriented data server clients. These data server clients implement the behaviours expressed in Scenario 4 (para. 6.4).

The Assembly Diagram is, for information, annotated with data flow arrows and identifications, and also how the ASCs are deployed into five separate ECOA Protection Domains. The demonstration can be built and run with all five Protection Domains running on one host, or distributed across multiple hosts.

Figure 39 ECOA Data Servers Demonstration Assembly



9.1 The Demonstration Mission System Oriented ASCs

Chapter 8 proposed example ECOA Services and Provider ASC implementations to illustrate each of the data server types discussed. The following paragraphs briefly describe the Mission System oriented ECOA

Services and ASCs created in order to exercise the Data Server Scenarios discussed in Chapter 6 (as condensed for demonstration in Section 6.4), and the Design Considerations discussed in Chapter 7.

9.1.1 AVModel ASC

In order to create a time variant behaviour within the demonstration system, the Air Vehicle Model (*AVModel*) ASC provides a source of simulated air vehicle state (position, altitude/height, speed, heading etc.).

Listing 9 AVModel ASC Definition XML

```
<componentType xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912"
               xmlns:xs="http://www.w3.org/2001/XMLSchema"
               xmlns:ecoa-sca="http://www.ecoa.technology/sca-extension-2.0">
  <service name="AVS">
    <ecoa-sca:interface syntax="AVS" qos="…"/>
  </service>
</componentType>
```

9.1.1.1 AVS Service

The Air Vehicle State (AVS) Service acts as a simple push-model data server using an ECOA (SENT_BY_PROVIDER) Event Operation. Periodically, the *AVSUpdate* ECOA Event is sent to all referencing ASCs.

Listing 10 AVS Service Definition XML

```
<serviceDefinition xmlns="http://www.ecoa.technology/interface-2.0">
  <use library="AVS"/>
  <operations>
     <event name="AVSUpdate" direction="SENT_BY_PROVIDER" >
          <input name="avs" type="AVS:AVS" />
     </event>
  </operations>
</serviceDefinition>
```

The Event carries a single data item of type *AVS* (from the AVS Data Types library), an XML encoded structure as follows:

Listing 11 An Example *AVS* Data Item

```
<AVS>
     <Latitude units="Degrees" direction="North">51.0</Latitude>
     <Longitude units="Degrees" direction="East">0.1</Longitude>
     <Height units="feet" reference="MeanSeaLevel">10123</Height>
     <Speed units="metresPerSec">123.56</Speed>
     <Heading units="Degrees" reference="NorthClockwise">234.6</Heading>
</AVS>
```

The reasons for using XML in this case are two:

i) To provide an example of XML encoded data;
ii) Hypothetically, the structure can be expanded by adding additional data items (such as air vehicle attitude, rate-of-climb, rate-of-turn, and so forth). This should not affect any existing clients of the Service, since the programmed interface remains unchanged.

9.1.2 HUMS ASC

The Health and Usage Management System (*HUMS*) ASC will implement the functional behaviour of the Scenarios of paras. 6.4.1 and 6.4.2.

The *HUMS* ASC will periodically simulate some sub-system data from the current air vehicle state, and store it to the *dbmServer* ASC of Figure 39 (acting as the "HUMS Short-term" store of Figure 9) in the form of KLV records. The ASC will also (at a longer period) snapshot (simulated) data from the *dbmServer* (acting as the "HUMS Cache" of Figure 10) to the *sqlServer* ASC ("HUMS DB").

9.1.3 MissionSys ASC

The Mission System (*MissionSys*) ASC will implement the functional behaviour of the Scenarios of paras. 6.4.4, 6.4.6 and 6.4.7.

When a vehicle state update is received (from the *AVModel* ASC) , the ASC will record a copy to the *dbmServer2* ASC of Figure 39 (acting in the role of "Vehicle State Store" of Figure 12).

The ASC will also retrieve, on demand, (simulated) Mission Plans and Initial targets from the *sqlServer* ASC (acting as the "Mission Plan" and "Tactical Items" databases) making a fast access (local) copy to the *dbmServer2* ASC (acting as the "Target Store").

In addition, the *MissionSys* ASC will provide a mechanism for on-demand query of the "Tactical Items" database (implemented in the *sqlServer* ASC), and display the retrieved content.

9.1.4 NavSys ASC

The Navigation System (*NavSys*) ASC will implement the functional behaviour of the Scenarios of paras. 6.4.3 and 6.4.5.

When a vehicle state update is received, the *NavSys* ASC will, acting in the role of a Digital Map engine, use the contained data to load one or more (simulated) map tiles from the "Map Tile repository" (of Figure 14) implemented by use of the *fileServer* ASC.

Periodically the *NavSys* ASC will also make queries on a weather source provided by the *metServer* ASC, storing a subset of the received data to the *sqlServer* ASC (acting as the "Weather Store" of Figure 11).

9.2 Build and Execution

The demonstration has been implemented in C, and built and run on an ECOA POSIX platform, with the five Protection Domains identified in Figure 39 hosted, for demonstration purposes, on Linux, Windows (with a POSIX compatible runtime), and VxWorks hosts.

Neither detailed ECOA Assembly, Implementation, and Deployment XMLs, nor the demonstration source code will be presented here, but are available under the same IP, and distribution restrictions, as this document.

The available code does not implement any of the "higher" functions (such as compression and security protocols) limiting itself to showing outline implementation of the various ECOA interfaces exercised using simple example data types. It is intended solely as a sign-post to the direction that solutions might take.

9.3 Warranty

The software is developed for and on behalf of BAE Systems (Operations) Limited, Dassault Aviation, Bull SAS, Thales Systemes Aeroportes, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Leonardo MW Ltd, and the copyright is owned by BAE Systems (Operations) Limited, Dassault Aviation, Bull SAS, Thales Systemes Aeroportes, GE Aviation Systems Limited, General Dynamics United Kingdom Limited and Leonardo MW Ltd

The software is developed by BAE Systems (Operations) Limited, Electronic Systems, and is the Intellectual Property of BAE Systems (Operations) Limited, Electronic Systems.

The information set out in the software is provided solely on an 'as is' basis and the co-developers of the software make no warranties expressed or implied, including no warranties as to completeness, accuracy or fitness for purpose, with respect to any of the information.