
SCIENCE AND TECHNOLOGY ORGANIZATION
CENTRE FOR MARITIME RESEARCH AND EXPERIMENTATION


**Technical Report** **CMRE-FR-2014-017**

# Sensor data management to achieve information superiority in maritime situational awareness

Giampaolo Cimino, Gianfranco Arcieri, Steven Horn, Karna Bryan

October 2014

# About CMRE

The Centre for Maritime Research and Experimentation (CMRE) is a world-class NATO scientific research and experimentation facility located in La Spezia, Italy.

The CMRE was established by the North Atlantic Council on 1 July 2012 as part of the NATO Science & Technology Organization. The CMRE and its predecessors have served NATO for over 50 years as the SACLANT Anti-Submarine Warfare Centre, SACLANT Undersea Research Centre, NATO Undersea Research Centre (NURC) and now as part of the Science & Technology Organization.

CMRE conducts state-of-the-art scientific research and experimentation ranging from concept development to prototype demonstration in an operational environment and has produced leaders in ocean science, modelling and simulation, acoustics and other disciplines, as well as producing critical results and understanding that have been built into the operational concepts of NATO and the nations.

CMRE conducts hands-on scientific and engineering research for the direct benefit of its NATO Customers. It operates two research vessels that enable science and technology solutions to be explored and exploited at sea. The largest of these vessels, the NRV Alliance, is a global class vessel that is acoustically extremely quiet.

CMRE is a leading example of enabling nations to work more effectively and efficiently together by prioritizing national needs, focusing on research and technology challenges, both in and out of the maritime environment, through the collective Power of its world-class scientists, engineers, and specialized laboratories in collaboration with the many partners in and out of the scientific domain.

# Sensor data management to achieve information superiority in maritime situational awareness

Giampaolo Cimino, Gianfranco Arcieri,
Steven Horn, Karna Bryan

This document, which describes work performed under Project Maritime Situational Awareness (MSA) of the NATO Science and Technology Organization – Centre for Maritime Research and Experimentation Scientific Programme of Work, has been approved by the Director.

Intentionally blank page

**Sensor data management to achieve information superiority in maritime situational awareness**

Giampaolo Cimino, Gianfranco Arcieri, Steven Horn, Karna Bryan

**Executive Summary:** The NATO Science & Technology Organization (STO) Centre for Maritime Research and Experimentation (CMRE) Maritime Situational Awareness (MSA) project investigates the potential of technologies, tools, and techniques to address the processing of maritime data for improved MSA to enable information superiority via the development of algorithms for multi-sensor fusion, target tracking and maritime anomaly detection. This is being developed as a service-oriented (SOA) based software architecture to ensure rapid adaptability and extensibility in a NATO networked environment.
The objective of this work was to develop a system able to store heterogeneous sensor data used in the maritime domain. Furthermore, it was design objective to achieve good performance and cost effectiveness. In particular, this report describes the backend tier of the SOA infrastructure which enables the storage and access to valuable MSA data such as AIS, Radar and SAR as follows:

- The design of the backend database, which is capable of storing very large datasets (up to several million contacts per day) using COTS software and relatively simple hardware. The system achieve excellent performance as demonstrated by a simulation of 40 concurrent database users incrementally querying 6 hours of AIS data over the Mediterranean Sea while achieving less than 100 millisecond response time per transaction.
- A software layer to make the database infrastructure transparent to client applications. This software layer abstracts the database structure which can be optimized for performance without adding additional complexity to the user.
- A generic software framework which allows users to acquire, transform and send data from any source to any destination. For this framework the database can be both a source (data extraction) and/or a destination (data load). Source/destination types and data formats can be easily combined achieving a fast configurable system to acquire, store and extract data.

The system presented, designed and implemented at NATO STO CMRE for the MSA project, has been deployed since 2009 and it is still under continuous development. It is shown to be able to deliver relevant information from sensors to the information consumer or to the decision makers, in a timely manner.

Intentionally blank page

**Sensor data management to achieve information superiority in maritime situational awareness**

Giampaolo Cimino, Gianfranco Arcieri, Steven Horn, Karna Bryan

**Abstract**: This report describes the data handling process set up at the NATO Science & Technology Organization (STO) Centre for Maritime Research and Experimentation (CMRE) which includes sensor data acquisition, processing, storage and access in support of the Maritime Situational Awareness (MSA) project. The Database Management System (DBMS) and the way in which sensor data is acquired and loaded using a database access layer framework for client applications is described. The system has been designed and developed to cope with extremely large data volumes generated by sensors and it is the foundation for supporting the CMRE MSA Service Oriented Architecture and the Fusion on Demand concept. Many aspects of this system are then analyzed: data sensor parsing, real-time database loading, database structure, database data extraction (real-time and historical). This analysis is supported with performance figures for the use of the system with real data sets. This analysis demonstrates that the system is an effective way to deliver relevant information to MSA decision makers. The whole system is currently deployed at CMRE.

**Keywords:** MSA, Database, DBMS, SQL Server, AIS, ETL, OLAP, OLTP, Radar, C#, Big Data, Microsoft .Net, Information Superiority

# Contents

# 1 Introduction

## 1.1 Rationale

The goal of the NATO STO CMRE Maritime Situational Awareness (MSA) project is to enable information superiority via the development of algorithms for multi-sensor fusion, target tracking and maritime anomaly detection. This is being developed as a service-oriented (SOA) based software architecture to ensure rapid adaptability and extensibility in a NATO networked environment. This capability will enable operators to make faster and better decisions, by presenting them with a high quality real-time and an historical picture of the maritime environment.

The MSA concept implies effective understanding of anything associated with the maritime domain, specifically with those phenomena that could impact security, the economy and the environment. Information is the foundation for MSA, therefore gathering, storing and accessing MSA datasets is of paramount importance. The data types involved are highly heterogeneous: nautical cartography, meteorological and oceanographic data (MetOc), real-time ship traffic, earth observation products, radar data and more. Work with this datasets is challenging for the non-harmonic aggregation of data types and for the huge size of some datasets, leading the designers of the system in the "Big Data" business.

This work describes the architecture implemented at CMRE to support the process graphically represented in Figure 1: . The components described in this report, belong to a larger system, Service Oriented Architecture based, described in [6] and in [7].

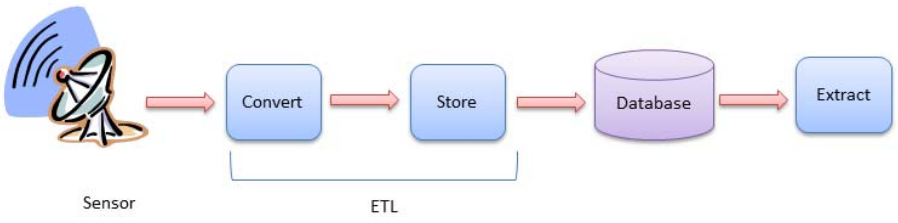


**Figure 1:** High level diagram of MSA incoming data flow.

## 1.2 Guide for the reader

The central four chapters of this paper are each aimed at specific audiences.

- Chapter 2 describes the database structure and presents a data access performance evaluation for the database instance installed at CMRE. It is recommended for software engineers, database administrators and system administrators
- Chapter 3 describes the sensor data parsing and assimilation process. It is recommended for system administrators and software engineers
- Chapter 4 is a guide on how to use the class framework to create new data acquisition applications. It is recommended for software engineers
- Chapter 5 describes how to use and how to extend the Database Access Layer (DAL). It is recommended for software engineers

# 2 Database

## 2.1 Technical foundation and system design

Storing data from heterogeneous sensors involved in the MSA domain is challenging for the following reasons:

- Sensor data volumes vary from a few ship positions detected from a Synthetic Aperture Radar (SAR) image to several millions of contacts per day retrieved from an Automated Identification System (AIS) network like AISHub[1];
- Both real-time and historical data access are required;
- The database must support continuous real-time data ingestion;
- The data is used for different purposes, e.g. visualization, post-processing, real-time processing, therefore is accessed with different patterns;
- A commodity server should be able to host the data in order to facilitate the installation of database in different organizations (keeping hardware requirements low enables cost effective system deployment).

On the other hand, the data involved is straightforward to store:

- Sensor data is Write-Once Read-Many, that is, once an information has been written in the database, it is not modified (no SQL update is performed);
- The data schema is simple and joins operations are rarely requested, limiting the number of foreign key relations.

The first technical decision was to favour relational databases over NoSQL databases. Despite the increasing popularity NoSQL database [1], the organization decided to benefit of its strong know-how in relational databases limiting the risk of an unexplored technology. Therefore, Microsoft SQL Server 2008 R2 was chosen as the database backend, hosted on a Windows Server 2008 R2 operating system.

Once the database type was set, the main architectural choice was the use of two concurrent relational databases one for On-Line Transaction Processing (OLTP) and one for On-Line Analytical Processing (OLAP). This pattern was inspired by database design best-practices used in Information Technology systems from different domains, such as enterprise, banking and finance. The idea is graphically explained in Figure 2: .

[1] AIS hub is an open AIS data sharing service which is available at http://aishub.net.

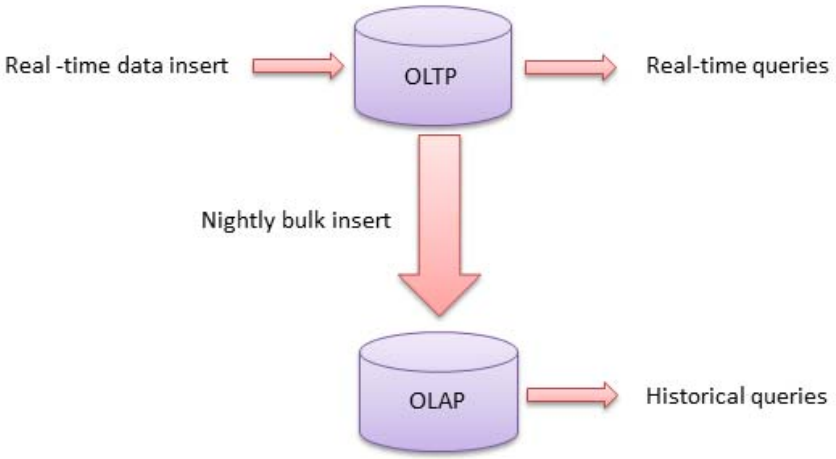


**Figure 2:** Diagram of OLAP/OLTP data flow.

The real-time ingestion of the sensor data is performed on the OLTP database while the long-running historical queries are executed versus the OLAP database. This separation leads to more scalability over the number of input sensors (on OLTP) and over the number of long running parallel queries (on OLAP). A nightly batch process loads the past day's data from OLTP to OLAP, using a bulk insert technique[2] and deletes from OLTP those records which are older than a retention policy threshold (normally four weeks). The current day's data is read by the clients from the OLTP database vice the real-time stream, which partially breaks the normal data flow. Although the real-time data is also retrieved from OLTP the overall performance is not adversely affected due to the relatively small data volume. The drawback of the OLAP/OLTP pattern is an increase of the complexity of the system as a whole, which requires more hardware, more database administration time and more system administrator efforts. To mitigate the increased system complexity overhead, the internal structure of the two databases (table schemas and their relationships) was kept as simple as possible.

In general, in the OLTP/OLAP pattern, the two databases have diverse schemas to favour different queries goals:

- OLTP: insert instructions and standard simple queries with small data sets and few fields

[2] Bulk insert is an efficient process or method provided by a database management system to load multiple rows of data into a database table (from www.wikipedia.org)

- OLAP: complex and long-running queries with large result-sets and a number of fields

Due to the relatively simple structure of the CMRE MSA data, the same schema is used for both OLAP and OLTP. The only difference is the way in which the data is split over different tables across time (sharding), as described in section 2.2:

- OLTP: tables split per sensor, per week of the year[3]
- OLAP: tables split per sensor, per month

The division per week of OLTP is optimal because it avoids the growth of the number of records per table. However, in OLAP, this strategy would result in an excessive number of tables (56 tables per year per sensor). Therefore, OLAP is split on monthly basis.

To make this architecture truly effective, the databases need to run on separate hardware, enabling the parallelization of the read and write operations on the two databases. Hardware virtualization has not been taken into account because is usually not recommended for database servers. Hard disk performance is very important to ensure the overall system performs effectively. In particular, OLTP requires fast disks with little space, while OLAP requires significant space, but speed is less important. The CMRE databases are hosted on two twin HP DL 380 G6 machines with 144 GB RAM and 2 Intel Xeon E5649 2.53 GHz microprocessors. The disks of the two servers are hosted in an external Storage Area Network (SAN) using a fibre channel link to a SAN EMC VNX 5300, with a 24 hard drive SAS, 900 GB each at 10K rpm. For an estimation of the potential throughput of the described hardware see [5].

## 2.2 Table structure

As of the time this report was written, the CMRE MSA database contains two principal types of data: AIS and radar (ground based and ship borne). Both types of data are stored in the OLAP and OLTP database, according to the scheme described in section 2.1. Within the databases, data is organized into separate tables according to sensor type, data type and time frame. The naming convention used for the tables describes this division as indicated below:

*<sensor>_<data type>_S<sensor ID>_<year>_<time range ID>*

where sensor ID is a unique integer number assigned to each sensor. For example: AISHub (sensor ID 4) AIS static data in OLTP database (weekly split) for week 33 of year 2012 would be labelled as:

*AIS_Static_S04_2012_33*

Another example: Radar contacts (Positioning) from R/V Alliance Radar (sensor ID 6) in OLAP database (monthly split) for February 2012 would be labelled as:

[3] Weeks in a year are numbered according to the ISO 8601 standard

*Radar_Contacts_S06_2012_02*

Sensor data is delivered in atomic items, e.g. AIS is sent using National Marine Electonics Association (NMEA) Very High Frequency (VHF) Data-link Message (VDM) sentences (see [2] and [3]) which consist of one or more lines of text. During the database design phase, different database schemas had been tested, all involving a level of normalization in order to reduce storage space. These approaches require a complex insert instruction since one data message may insert records in multiple tables. Furthermore, querying on this structure also requires complex selects which join multiple tables. All these operations decrease the overall database performance when the data volume is massive (see 2.3). Therefore, a simple design was chosen: store each atomic data item in a single database record (e.g. one AIS sentence per record). This approach will cause data redundancy in the database, since tables are not normalized. On the other hand, de-normalization is a common practice in database administration to avoid slow join operations. The drawback to this approach is an increased need for disk space to store larger tables. Fortunately disks are currently a relatively inexpensive resource.

In the following subsections, each sensor currently being archived in the database is described. Note that, as generic rule, data is stored in International System units, and positions are stored in geographic coordinates (decimal degree, latitude relative to north, WGS84).

### 2.2.1 Automatic Identification System (AIS) data

As described in [2] and [3], AIS data is transmitted using NMEA VHF Data-link Message (VDM). Several types of messages are received, but only VDM AIS message types 1, 2, 3, 5, 18, 19, and 24 are loaded into the MSA database. These messages are effectively classified into two types: static (5, 19, and 24) and positioning[4] (1, 2, 3, 18, and 19). Positioning messages report the unit position and are transmitted at a frequency that depends on the unit's speed and manoeuvering status. Static messages contain voyage and size/identity information and are transmitted at lower rate than the position messages. According to this classification, two types of tables are generated in the database for a given AIS dataset: AIS_Contacts (for positioning data) and AIS_Static (for static data). Table 1: describes the AIS Contacts table.

[4] In this paper the words positioning, kinematic and contacts are used interchangeably

| Field name | Field Type | Allow null | Notes | Mapping to AIS message ID |
|---|---|---|---|---|
| Time | int | No | Primary Key. Epoch time | N.A. |
| Id | int | No | Primary Key. MMSI | All |
| Latitude | float | No | Index. Decimal Degree relative to north | 1, 2, 3, 18, 19 |
| Longitude | float | No | Index. Decimal Degree | 1, 2, 3, 18, 19 |
| CourseOverGround | float | Yes | Decimal Degree relative to north | 1, 2, 3, 18, 19 |
| TrueHeading | float | Yes | Decimal Degree relative to north | 1, 2, 3, 18, 19 |
| SpeedOverGround | float | Yes | [knots] | 1, 2, 3, 18, 19 |
| RateOfTurn | float | Yes | [deg/min] | 1, 2, 3 |
| PositionAccuracy | tinyint | Yes | See [2] | 1, 2, 3, 18, 19 |
| NavigationalStatus | tinyint | Yes | See [2] | 1, 2, 3 |
| AISMessageId | tinyint | No | | All |
| Sentence | varchar (512) | No | Complete AIS sentences, comma separated | N.A. |

**Table 1:** AIS contacts table in the CMRE MSA database.

The table contains also the time (in epoch time format) which isn't included in the original VDM message. This timestamp is usually assigned at the time of AIS message reception. Additionally, the full raw NMEA sentence is added to the table to keep track of the original data. The primary key consists of Id using the Maritime Mobile Service Identity (MMSI), and time. Latitude and longitude are used as a non-clustered indexes in order to accelerate the spatial queries.

The AIS static table is described in Table 2:.

| Field name | Field Type | Allow null | Notes | Mapping to AIS message ID |
|---|---|---|---|---|
| Time | int | No | Primary Key. Epoch time | N.A. |
| MMSI | int | No | Primary Key. | all |
| Name | varchar(64) | Yes | | 5, 19, 24A |
| IMO | int | Yes | | 5 |
| CallSign | varchar(12) | Yes | | 5, 24 |
| Width | smallint | Yes | [m] | 5, 19, 24B |
| Length | smallint | Yes | [m] | 5, 19, 24B |
| AntennaFromBow | smallint | Yes | [m] | 5, 19, 24B |
| AntennaFromPort | smallint | Yes | [m] | 5, 19, 24B |
| ETAMonth | tinyint | Yes | | 5 |
| ETADay | tinyint | Yes | | 5 |
| ETAHour | tinyint | Yes | | 5 |
| ETAMinute | tinyint | Yes | | 5 |
| Draught | float | Yes | [m] | 5 |
| Destination | varchar (48) | Yes | | 5 |
| ShipTypeId | smallint | Yes | See [2] | 5, 19, 24B |
| CargoId | smallint | Yes | See [2] | 5, 19, 24B |
| AISMessageId | tinyint | No | | All |
| Sentence | varchar (512) | No | Complete AIS sentences, comma separated | N.A. |

**Table 2:** AIS Static table in CMRE MSA database.

Timestamp and raw VDM sentence are added as in the contacts table. Primary key is again made by time and Id (MMSI). For this table, there are no secondary indexes. A detailed description of AIS fields is available in [2].

Not all of the AIS static/kinematic messages contain all the fields reported in the database tables: a mapping is available in the last column of Table 1: and Table 2:. If a message does not contain some fields, the table value is set to NULL.

### 2.2.2 Radar data

The CMRE MSA project collects data also through ground and ship borne radars. The radar data is post-processed and only the high confidence detections are saved into the database. The format of these contacts is heavily dependent on the sensor type and the post processing algorithm. At this preliminary stage two radar types are used: R/V Alliance ship borne radar, providing the data in NMEA 0183 Tracked Target Message (TTM) format (see [3]), and WERA High Frequency surface wave radar[5] (see [17] [18]), providing the data in custom text files. For the radar data, only a contacts table is created since there is no equivalent of static data as found in AIS.

The table structure for R/V Alliance radar is shown in Table 3:. Note that the contact latitude and longitude are calculated by the data loader using the closest R/V Alliance GPS position and the range and bearing from the TTM message. Time stamp and the whole TMM sentence are also stored in each record.

The table structure for HF WERA surface wave radar is shown in Table 4:. A detailed description and units of measure of the parameters can be found in [4].

In both radar tables the primary key is time and Id/progId. A non-clustered index is generated on Latitude and Longitude fields for faster spatial queries. Note that for the WERA Radar, the progId (progressive Id) field does not associate contacts from two consecutive radar scans.

[5] CMRE deployed two WERA HF Radar stations: on Palmaria Island, La Spezia, Italy and on San Rossore coast, Pisa, Italy.

| Field name | Field Type | Allow null | Notes |
|---|---|---|---|
| Time | int | No | Primary Key. Epoch time |
| ID | int | No | Primary Key. Contact progressive ID |
| RangeFromSensor | float | Yes | [km] |
| BearingFromSensor | float | Yes | Decimal Degree relative to north |
| Speed | float | Yes | [km/h] |
| Heading | float | Yes | Index. Decimal Degree relative to north |
| Longitude | float | Yes | Index. Decimal Degree |
| Latitude | float | Yes | Decimal Degree relative to north |
| SensorLongitude | float | Yes | Decimal Degree |
| SensorLatitude | float | Yes | Decimal Degree relative to north |
| Sentence | varchar (512) | No | TTM sentence |

**Table 3:** R/V Alliance ship borne radar table in CMRE MSA database.

| Field name | Field Type | Allow null | Notes |
|---|---|---|---|
| Time | int | No | Primary Key. Epoch time |
| progId | int | No | Primary Key. Contact progressive ID |
| Longitude | float | Yes | Index. Decimal Degree |
| Latitude | float | Yes | Index. Decimal Degree relative to north |
| SensorLongitude | float | Yes | Decimal Degree |
| SensorLatitude | float | Yes | Decimal Degree relative to north |
| RangeFromSensor | float | Yes | [km] |
| BearingFromSensor | float | Yes | Decimal Degree relative to north |
| contactRadialSpeedReversed | float | Yes | [m/s] |
| stdevDistance | float | Yes | Stdev of RangeFromSensor |
| stdevAngle | float | Yes | Stdev of BearingFromSensor |
| stdevSpeed | float | Yes | Stdev of contactRadialSpeedReversed |
| SNRdbCFAR | float | Yes | See [4] |
| SNRdb | float | Yes | See [4] |

**Table 4:** WERA High Frequency ground radar contacts table in CMRE MSA database.

## 2.3 Data volumes

The database system described in the paragraphs above has been running at NATO STO CMRE since 2009. Figure 3: presents the number of records per month in the OLAP database for all the AIS sensors.

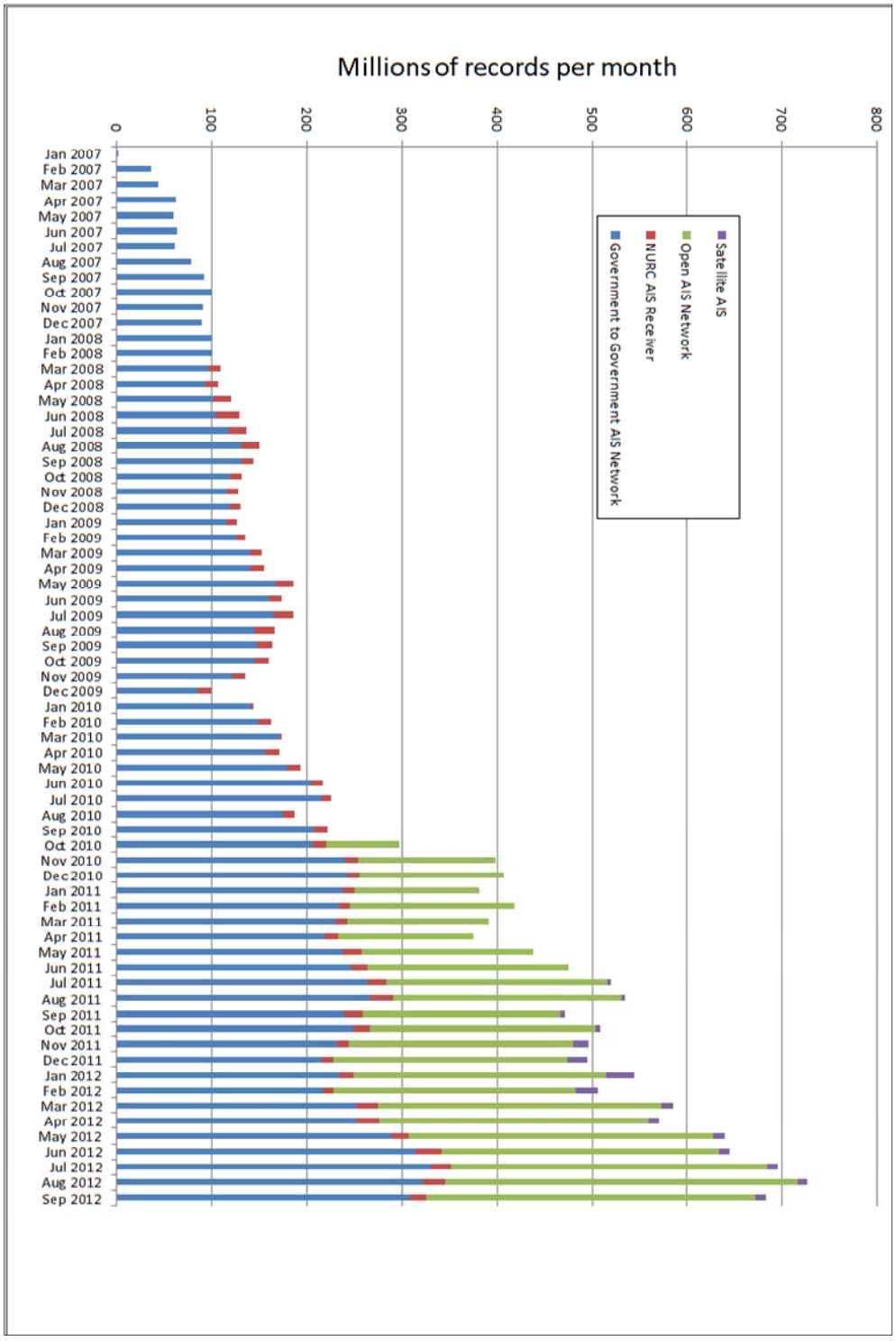


**Figure 3:** Data volumes for CMRE OLAP database.

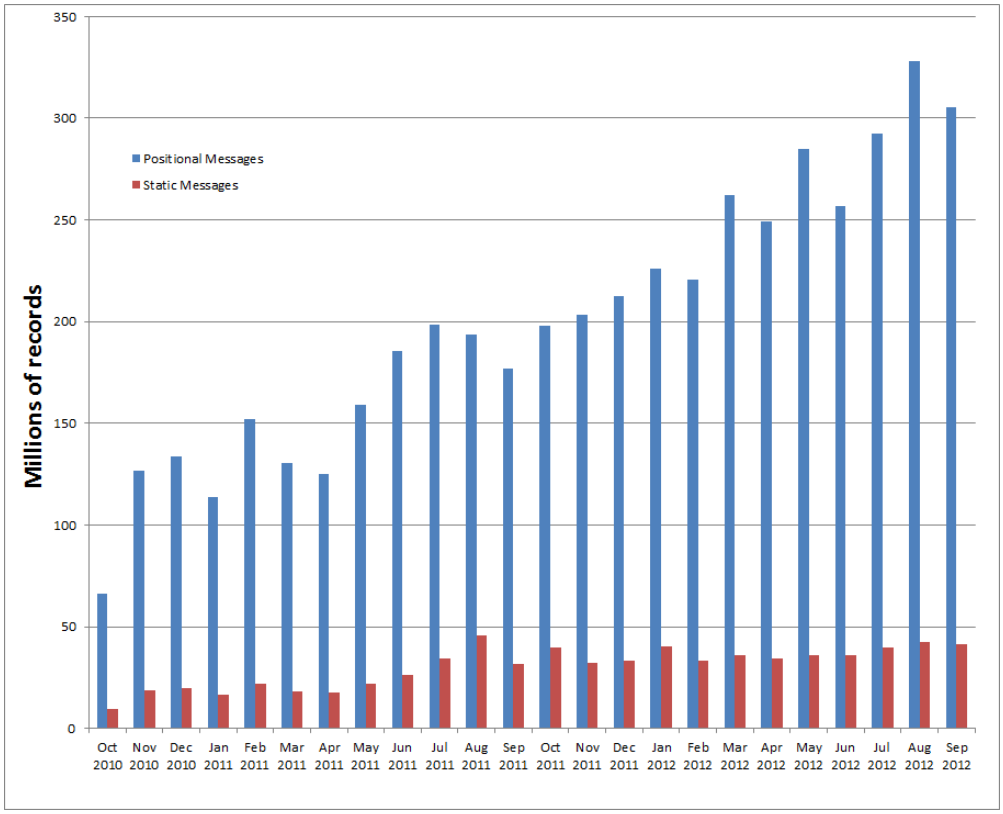


**Figure 4:** Data volumes in CMRE OLAP database for the Open AIS Network data provider divided into static and positional messages.

The sensors are:

- CMRE AIS receiver[6], local coverage
- Open AIS Network, world coverage
- Government to Government AIS network, world coverage

Satellite AIS network, non-continuous world coverage

The chart shows that the number of records, hence the number of AIS messages, increases over time as expected due to the increasing number of AIS receivers connected to the AIS network and by the increasing number of ships using AIS devices. It is also observed that the AIS message volume follows a seasonal pattern with more records during the summer period.

[6] The CMRE AIS receiver is set up in 44.06752° N, 9.816185° E, in Castellana locality, La Spezia, Italy

The number of records per month in Figure 3: comprises both AIS static and position messages. In the database, these messages are split over two corresponding tables (AIS_Static and AIS_Contacts), hence the number of records per database table is less than depicted in Figure 3: . For clarity, Figure 4: shows the number of messages in the OLAP database for the open AIS network divided into static and position messages. In this case, the numbers from the chart are equal to the number of records per table (split on a monthly basis). Figure 4: shows an extremely large number of records per table. This number is larger than recommended guidelines for database administration best practices. However, the data extraction queries performance resulted in acceptable performance in terms of latency (see 2.4), mainly due to the simplicity of the data extraction query. The OLTP database does not suffer of this large table problem, since it is split on weekly basis.

## 2.4 Performance

This performance analysis focuses on the CMRE MSA OLAP database for extraction queries (SQL selects). The OLTP database is smaller, and therefore results are less affected by performance degradation for select queries. The goal of the performance test was to calculate the extraction query time with an increasing number of concurrent users. To this end, the following SQL query was used:

```
SELECT Time, ID, Longitude, Latitude
FROM AIS_Contacts_S04_2011_<month>
WHERE
        Time BETWEEN <t1>+<randomOffset> AND <t1>+<randomOffset>+300 AND
        latitude BETWEEN 20.0 AND 40.0 AND
        longitude BETWEEN 0.0 AND 40.0
```

where:

- <month> is the number of month that identify the table
- <t1> is the time stamp of the first day of the <month> at 00:00:00
- <randomOffset> is a random time span across the <month>

Note that the constant 300 in the where clause is expressed in seconds, and it is the length of the time interval queried (five minutes). To make the test more realistic there is a fixed spatial where clause.

The test was done simulating six concurrent queries (as the one described above), referring to the first 6 months of 2011 for the AIS open network sensor, using 5, 10, 20, 30 and 40 concurrent users. Each of these six queries is run 72 times (72 queries, each one extracting 300 seconds of data, to simulate an incremental extraction of 6 hours of data). Note that the <randomOffset> parameter invalidates the effect of the DBMS cache: each single query executed during the test uses a different value of <randomOffset> (randomly generated), extracting different block of data in the table, trying to deactivate the effect of caching. The results with the average response time, and the system throughput is shown in Figure 5: . The average response time for up to 40 concurrent users doesn't exceed the value of 100 milliseconds. This shows that the overall architecture used is extremely effective. Also, the throughput of the system doesn't show

any significant degradation for the range of concurrent users chosen for the test, showing that the system is not saturated for these query volumes.

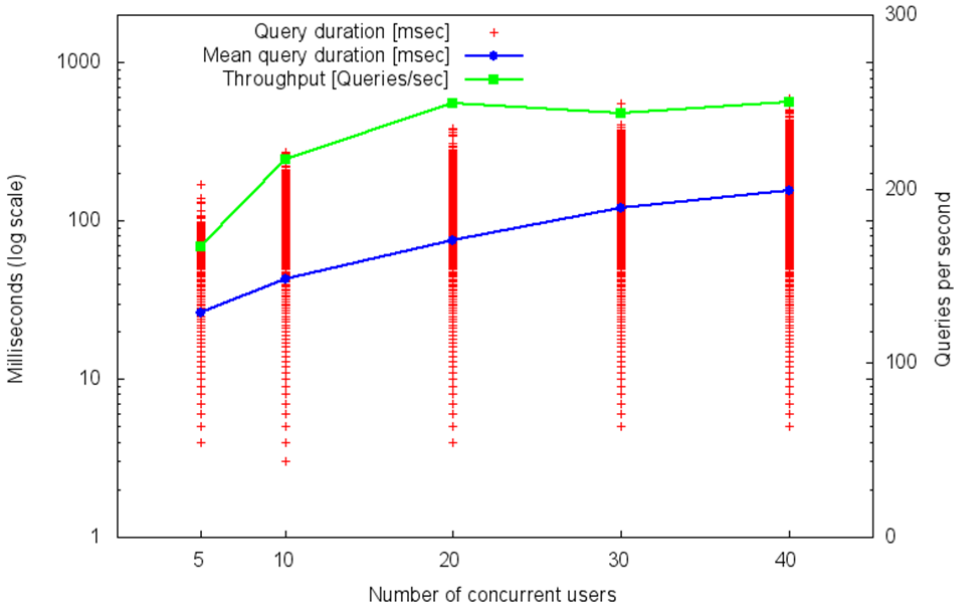


**Figure 5:** CMRE MSA OLAP database performance analysis.

# 3 Data Assimilation

## 3.1 Introduction

This chapter describes a software framework developed to support the CMRE MSA project implementing the sensor data extraction, transformation, and delivery functionalities. This framework has been completely implemented at CMRE, using Microsoft .NET and it is based on the Situational Awareness Core (SACore) described in [6]. The main functionalities are included in the *CMRE.IO.Reports* namespace. These functionalities enable a three steps process:

- Acquire data from a data source (e.g. file(s), network, databases, web services, etc.)
- Transform the data into a common format
- Send the data to a set of destinations (e.g. file(s), network, databases, web services, etc.)

At any level it is possible to add filtering capabilities to exclude data items that are of little interest for the desired application. If the final destination is a database, a complete Extract Transform Load (ETL) process is achieved, as previously indicated in Figure 1: . The common data format is the *IReport* interface and its subclasses, as described in [6]. This format is able to represent heterogeneous data sets (AIS messages, radar contacts, GPS sentences, etc.) acting as a dynamic type.

## 3.2 Class Structure

The main high-level (abstract) classes of the *CMRE.IO.Reports* namespace are shown as UML class diagram in Figure 6: . For readability, only the most significant public methods are shown in the diagram. The central class is *ReportSource*, whose subclasses encapsulate the physical source of data: File, TCP, database, etc. The class hierarchy underneath *ReportSource* is described in the class diagram of Figure 7: . *ReportSourceASCII* handles all the sources providing data in ASCII format (e.g. text file). *ReportSource* has a *Start()* (blocking) and *Stop()* method to control the production of reports.

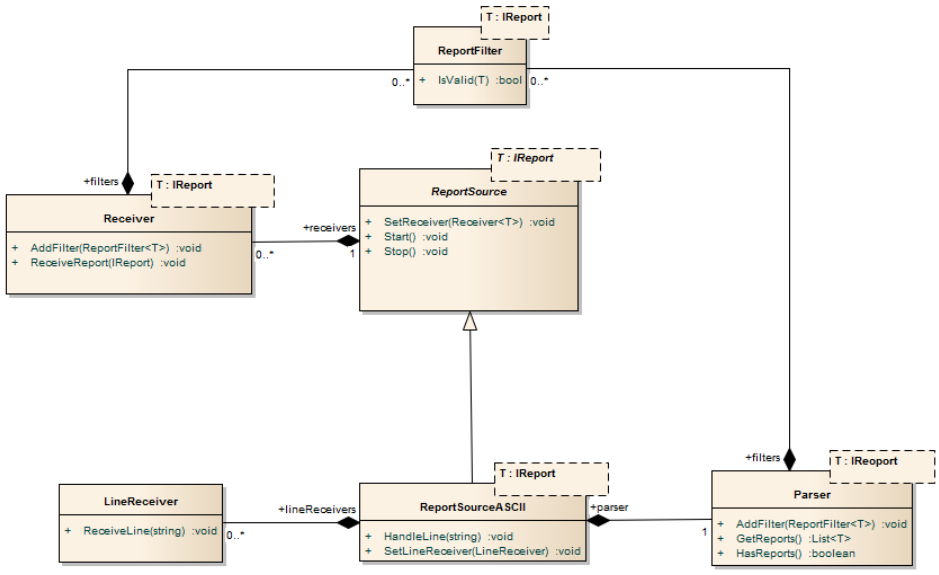


**Figure 6:** High level UML class diagram of CMRE.IO.Reports.

The *Parser* class is associated with *ReportSourceASCII*, and it has the responsibility of converting the text lines in subclasses of *IReport*, according to the actual type of the generic type parameter T. Note that some conversions might not be available, in this case, during object construction, a run-time exception will be thrown. The whole class hierarchy under *Parser* is further described in the class diagram in Figure 8: .

The *Parser* also has filtering capabilities using concrete *ReportFilter* implementations (see UML class diagram in Figure 9: ). In order to add a filter to a *Parser*, it is necessary to call the *AddFilter()* method on the parser object. If more than one filter is added, all of the filter's conditions must be satisfied for the parsed object to pass. In this way, filters are combined with the logic operator AND. Although basic filtering capabilities are available in the framework, adding new filters is very easy for the users by just adding sub-classes of *ReportFilter* as will be explained in Chapter 4.

Converted reports are delivered to the concrete implementations of the class *Receiver* (see the UML class diagram in Figure 11: ). Multiple receivers can be added to *ReportSource* using the *SetReceiver()* method. In the case of multiple receivers, all reports are sequentially delivered to each *Receiver* with the blocking method *ReceiveReports()*. A receiver may use a *ReportFilter()* in order to locally discard reports of little interest.

*ReportSourceASCII* delivers the input text lines, without any processing action, to a *LineReceiver* (see UML class diagram in Figure 10: ), using the method *ReceiveLine()*.

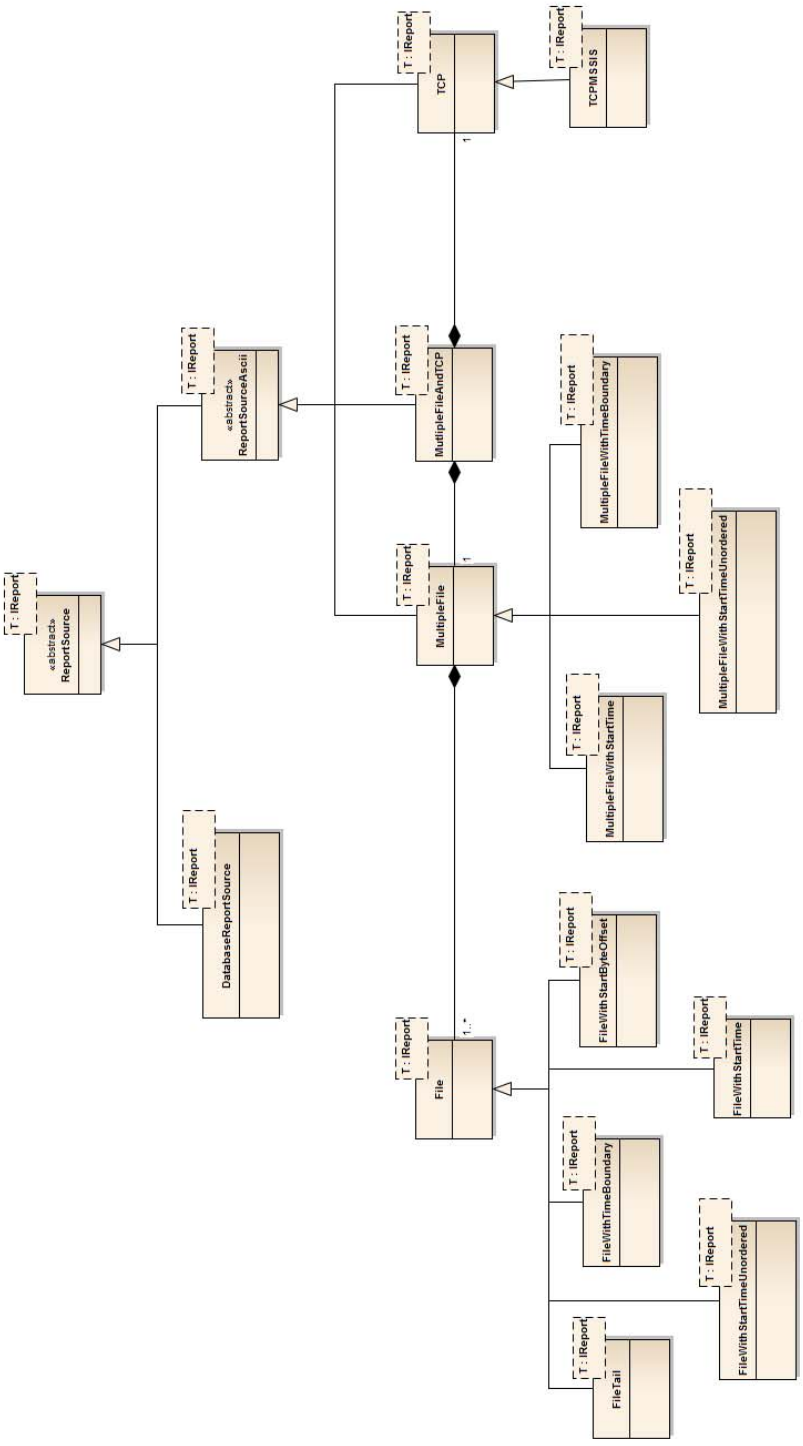


**Figure 7:** UML class diagram of ReportSource class hierarchy.

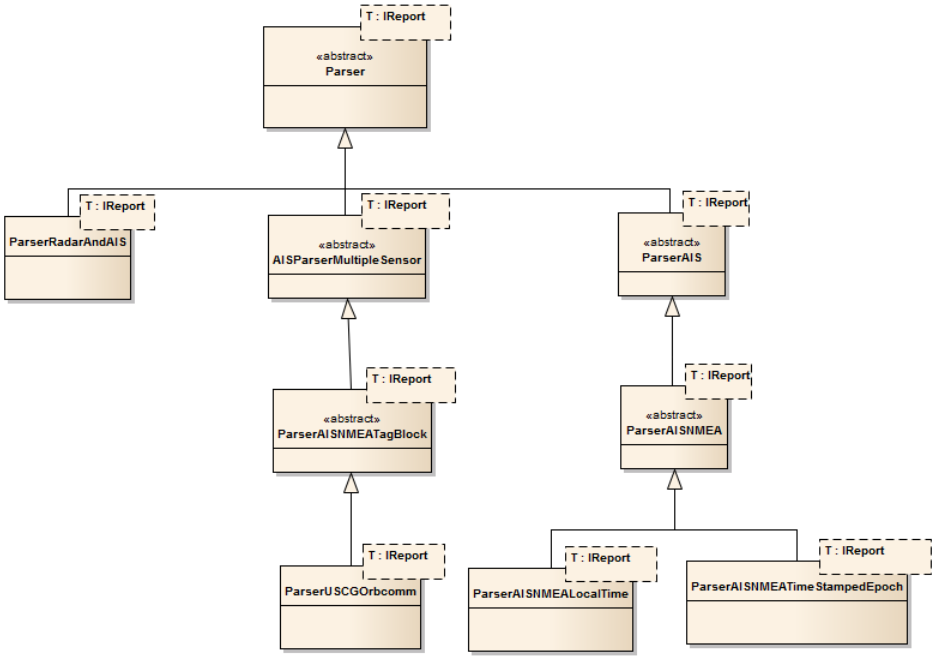


**Figure 8:** UML class diagram of the Parser class hierarchy.

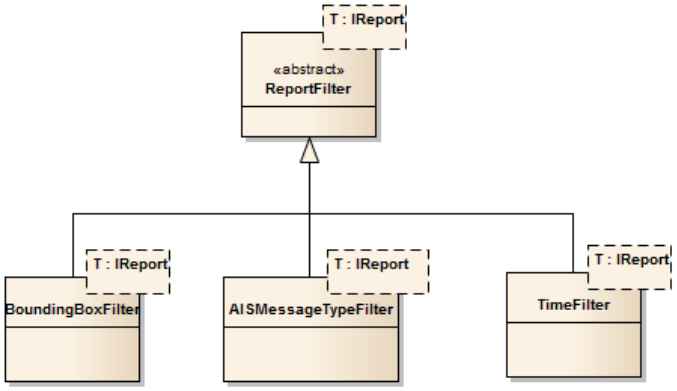


**Figure 9:** UML class diagram of ReportFilter class hierarchy.

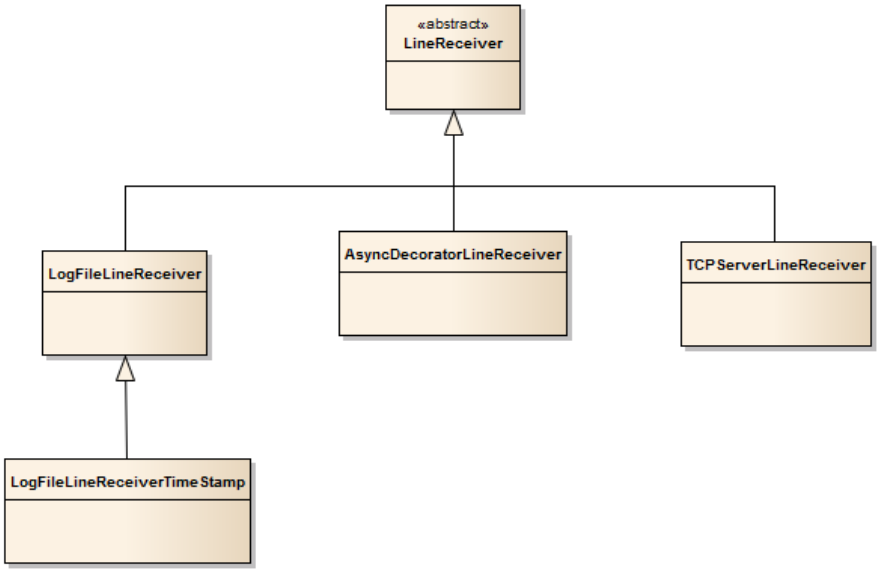


**Figure 10:** UML class diagram of LineReceiver class hierarchy.

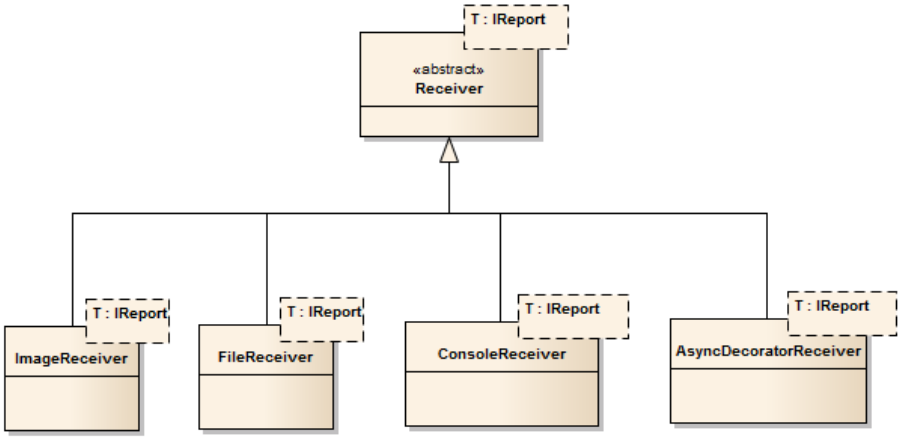


**Figure 11:** UML class diagram of Receiver class hierarchy.

# 4
# Data Assimilation Guide

## 4.1 Introduction

This chapter describes how to create new applications based on the class framework described in chapter 3. This is done by guiding the reader through the creation of a simple data logger application that uses the features available in the *CMRE.IO.Reports* namespace. This chapter assumes a basic knowledge of the C# programming language [8] and the availability of a set of assemblies containing the framework and some extra functionality.

## 4.2 How to create a simple logger

### 4.2.1 A minimal logger

The first step is to create a new Visual Studio 2010[7] console application project using .Net framework 4.0. Once the project has been created add to the project references the assemblies: *SACore.Information.Basic.dll, CMRE.Utility.dll CMRE.IO.Report.dll, SACore.dll.*

We assume a live AIS stream is available to the user and the data is transmitted as encoded text using the classic NMEA 0183 VDM message format as described in [2][3]. The AIS data transmission is usually based on a raw TCP connection initiated by the client to a server that streams the AIS data. Once the connection is open, the server starts to send the data in the format below (ASCII text) and the client only receives.

```
!AIVDM,1,1,,B,13coMH3P000e0EpI?:00oOvV0@;V,0*4C
!AIVDM,1,1,,B,14i1Rr?1@0Pe9qpI>gSewop`2400,0*03
!AIVDM,1,1,,A,13cbA`0P1c0a;:0I07l9M7fT08<0,0*01
!AIVDM,1,1,,B,13cf4=00000g9UHHsFWHAF4b0400,0*6F
!AIVDM,1,1,,B,181:Jm`w@00g:tPHsbb:2ULT0<0U,0*39
!AIVDM,2,1,0,A,55ANwt0221cDS<Q0000m<>0Hhu8T4p4000000017D@I,0*6C
!AIVDM,2,2,0,A,>>4I60@S0H4l1FR@@000000000000,2*49
!AIVDM,1,1,,B,13cfMV0P000g:3VHs?<0vgvb2400,0*5A
!AIVDM,1,1,,A,402;bK1u`M00E0bmd8HVqu7000S:,0*39
!AIVDM,2,1,1,B,55ANwt0221cDS<Q0000m<>0Hhu8T4p4000000017D@I,0*6E
!AIVDM,2,2,1,B,>>4I60@S0H4l1FR@@000000000000,2*4B
```

[7] Note that is possible to develop the application employing the free Visual Studio Express 2010 IDE. It is also possible to use the command line C# compiler csc.exe

```
!AIVDM,1,1,,A,13cwh`?P000g9l4Hs@C@0?vV0400,0*32
!AIVDM,1,1,,B,34h>OE5v010e:=8I>fmqMGpdP000,0*25
!AIVDM,1,1,,B,13cg:r003E0camjHsQ?mpT``0@=E,0*3C
!AIVDM,1,1,,B,15SN00002R0aHcFI1DoaPG`f0<0e,0*1A
!AIVDM,1,1,,B,13cjbg0wP00foW0I66`8NQnf0D0I,0*59
```

In case the reader has no available live AIS source, for testing, is possible to use a free AIS stream provided by *hd-sf.com* for the San Francisco Bay Area on TCP port 9009[8]. The code to set up the TCP data source is shown below.

```
using CMRE.IO.Reports.InputStream;
using SACore.Information;
using CMRE.IO.Reports.MessageParser;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
            Parser<IReport> parser = new ParserNMEALocalTime<IReport>(1, "San
Francisco AIS");

            ReportSourceASCII<IReport> source = new TCP<IReport>("hd-sf.com",
9009, parser);
        }
    }
}
```

The parser is responsible for converting the incoming data (AIS NMEA VDM messages) into the common data format (described in [6]), that is specified with the generic parameter *T*. The common data format must be consistent with all the objects connected to the parser. In this case the common format is the most generic one: *IReport*. The user can specify any descendent of *IReport*, when specific data formats are required (this is discussed further in 4.2.6). Note that the concrete class for the parser is *ParserNMEALocalTime*, which will add the system time to each received AIS message. This timestamp is necessary for storing and archiving purposes since the incoming AIS data has no time stamp included due to the fact that the format is originally designed for real-time applications. The second step is to create the TCP report source using *hd-sf.com* as address and 9009 as TCP port. The third parameter is the just created parser object. Note that *ReportSource* has the same generic type parameter of *Parser*.

As final step a *FileReceiver* is created and added to the *ReportSource* using *SetReceiver()* method. The *FileReceiver* redirect the converted incoming data to the file

[8] Please note that hd-sf.com (San Francisco Bay Area AIS) feed data provided free for non-commercial use with best efforts service

*d:\tmp\ais_parsed.txt*. The acquisition begins when the blocking method *Start()* is called on the source object. See the code below (with the newly added code highlighted):

```
using CMRE.IO.Reports.InputStream;
using SACore.Information;
using CMRE.IO.Reports.MessageParser;
using CMRE.IO.Reports.OutputStream;
using CMRE.IO.Reports.OutputStream.ReportReceiver;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
            Parser<IReport> parser = new ParserNMEALocalTime<IReport>(1, "San
Francisco AIS");
            ReportSourceASCII<IReport> source = new TCP<IReport>("hd-sf.com",
9009, parser);

            Receiver<IReport> dataReceiver = new
FileReceiver<IReport>(@"d:\tmp\ais_parsed.txt");
            source.SetReceiver(dataReceiver);
            source.Start();
        }
    }
}
```

Now the file *d:\tmp\ais_parsed.txt* should receive the (parsed) incoming data. Assuming UNIX tools are available on the system, the output of the command "*tail -f d:\tmp\ais_parsed.txt*" is shown in Figure 12: .

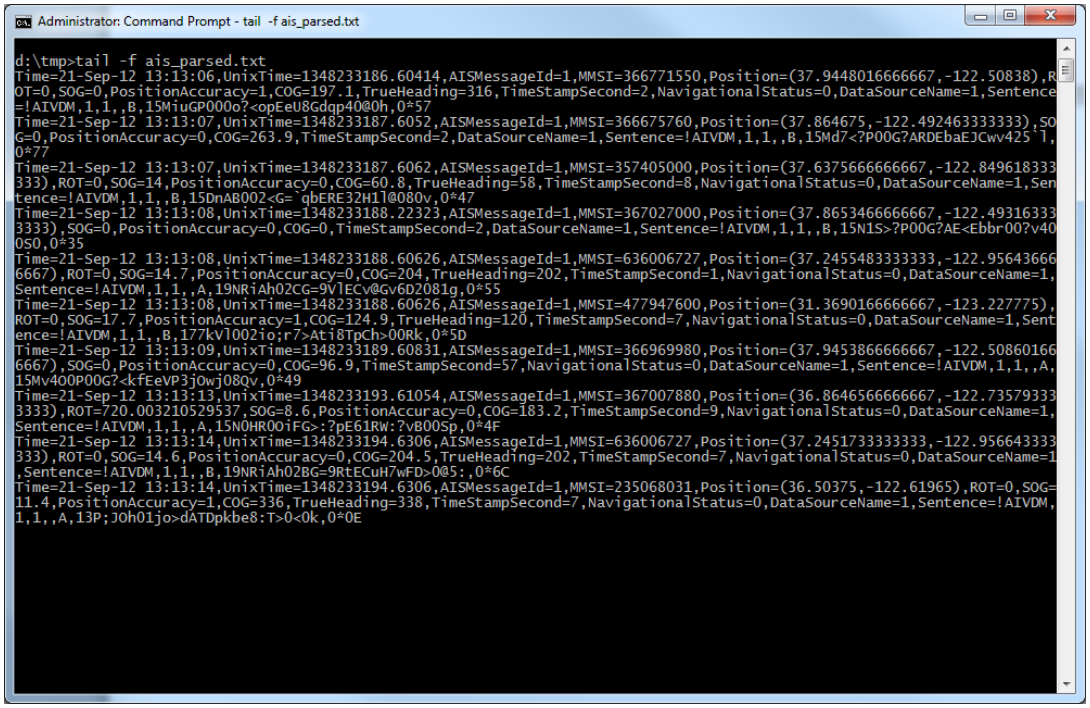


**Figure 12:** Log file generated by the simple logger.

### 4.2.2 Save the raw data

It is usually desirable to log the incoming data in a raw format to files, especially for backup or for future use. To this end, it is possible to add a *LogFileLineReceiverTimeStamp* to the source object. This Receiver inherits from *LineReceiver* (not from *Receiver*), and it works only with ASCII data sources. Its responsibility is to write the incoming data (AIS sentences in this case) to a destination without applying any parsing. In this case, the destination is a file. Each line will be decorated (at end of each line) with the receiving time stamp in epoch time format. This receiver has log file rolling functionalities (by default the rolling is daily based).

```
using CMRE.IO.Reports.InputStream;
using SACore.Information;
using CMRE.IO.Reports.MessageParser;
using CMRE.IO.Reports.OutputStream;
using CMRE.IO.Reports.OutputStream.ReportReceiver;
using CMRE.IO.Reports.OutputStream.LineReceiver;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
            Parser<IReport> parser = new ParserNMEALocalTime<IReport>(1, "San
Francisco AIS");
```

```
            ReportSourceASCII<IReport> source = new TCP<IReport>("hd-sf.com",
9009, parser);

            Receiver<IReport> dataReceiver = new
FileReceiver<IReport>(@"d:\tmp\ais_parsed.txt");
            source.SetReceiver(dataReceiver);

            LineReceiver logFile = new
LogFileLineReceiverTimeStamp(@"d:\tmp\ais_log.txt");
            source.SetLineReceiver(logFile);

            source.Start();
        }
    }
}
```

The incoming AIS data in NMEA VDM format should be saved in the file *d:\tmp\ais_log.txt*.

### 4.2.3 Add filtering capabilities

Assume one wants to limit the amount of parsed data sent to the *FileReceiver* (*ais_parsed.txt*), and log the raw data in native format (*ais_log.txt*). This is a realistic scenario, since the entire data is usually kept for archive and future-use, while for the real time processing one might be interested in a subset of the data set (for instance to a specific region). This can be achieved using the filtering capabilities. For instance, in the case of the San Francisco bay, let's say we want only the AIS messages with vessel position east of the Golden Gate bridge (longitude less than 122 28' 42'' W). In this case, one just needs to add a new *BoundingBoxFilter* to the dataReceiver object.

```
using CMRE.IO.Reports.InputStream;
using SACore.Information;
using CMRE.IO.Reports.MessageParser;
using CMRE.IO.Reports.OutputStream;
using CMRE.IO.Reports.OutputStream.ReportReceiver;
using CMRE.IO.Reports.OutputStream.LineReceiver;
using CMRE.IO.Reports.Filter;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
            Parser<IReport> parser = new ParserNMEALocalTime<IReport>(1, "San
Francisco AIS");
            ReportSourceASCII<IReport> source = new TCP<IReport>("hd-sf.com",
9009, parser);

            Receiver<IReport> dataReceiver = new
FileReceiver<IReport>(@"d:\tmp\ais_parsed.txt");

            //122 28' 42'' W = -122.47833333
            BoundingBoxFilter<IReport> eastOfGoldenGate =
                new BoundingBoxFilter<IReport>(-122.47833333, 0, -90, +90);
            dataReceiver.AddFilter(eastOfGoldenGate);
            source.SetReceiver(dataReceiver);
```

```
            LineReceiver logFile = new
LogFileLineReceiverTimeStamp(@"d:\tmp\ais_log.txt");
            source.SetLineReceiver(logFile);

            source.Start();
        }
    }
}
```

The output in *ais_parsed.txt* file should now contain only AIS messages with kinematic information and with the longitude between -122.4783333 and 0. Note that no limitation has been set on latitude (set between -90 and +90).

### 4.2.4 Add a second Receiver

A *ReportSourceASCII* object could have multiple *Receiver* and multiple *LineReceiver* objects. In this case, the converted data and the original data are sent sequentially to each receiver. Let's say we want to add a second *LineReceiver* that acts as TCP relay on port 19009, allowing other clients to connect and get the data.

```
using CMRE.IO.Reports.InputStream;
using SACore.Information;
using CMRE.IO.Reports.MessageParser;
using CMRE.IO.Reports.OutputStream;
using CMRE.IO.Reports.OutputStream.ReportReceiver;
using CMRE.IO.Reports.OutputStream.LineReceiver;
using CMRE.IO.Reports.Filter;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
            Parser<IReport> parser = new ParserNMEALocalTime<IReport>(1, "San
Francisco AIS");
            ReportSourceASCII<IReport> source = new TCP<IReport>("hd-sf.com",
9009, parser);

            Receiver<IReport> dataReceiver = new
FileReceiver<IReport>(@"d:\tmp\ais_parsed.txt");
            //122 28’ 42’’ W = -122.47833333
            BoundingBoxFilter<IReport> eastOfGoldenGate =
                new BoundingBoxFilter<IReport>(-122.47833333, 0, -90, +90);
            dataReceiver.AddFilter(eastOfGoldenGate);
            source.SetReceiver(dataReceiver);

            LineReceiver logFile = new
LogFileLineReceiverTimeStamp(@"d:\tmp\ais_log.txt");
            source.SetLineReceiver(logFile);

            TCPServerLineReceiver tcpRelay = new TCPServerLineReceiver(19009);
            source.SetLineReceiver(tcpRelay);

            source.Start();
        }
    }
}
```

Once the program is running, it should be possible to open a connection to the port 19009 (using raw TCP connection[9]) at *localhost* address and get back the data stream in the original format. Note that the time stamp in epoch time format will be added at the end of each line. The instances of all of the objects created in this source code are described in the UML diagram in Figure 13.

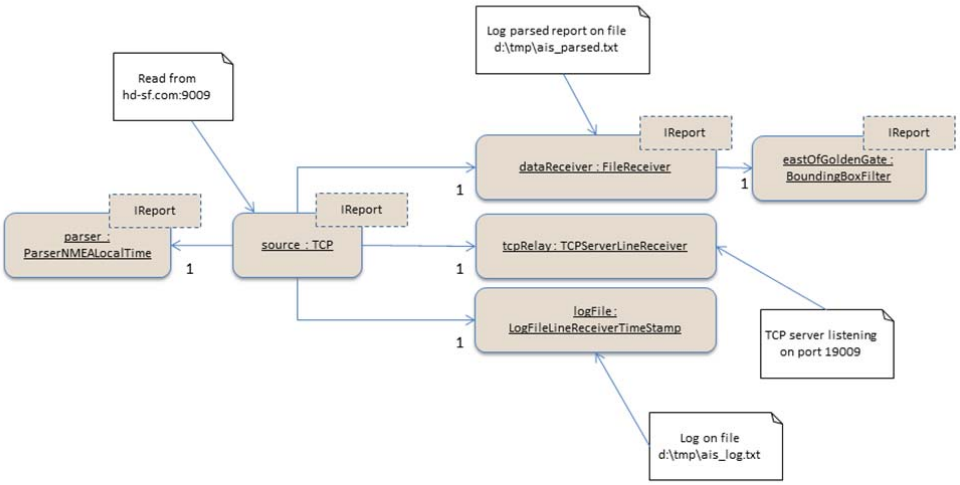


**Figure 13:** UML object diagram for source code in paragraphs 4.2.1, 4.2.2, 4.2.3 and 4.2.4.

### 4.2.5 Use a different input source

Assume now that one wishes to change the input source to use a text file instead of a TCP stream. This is possible by just changing the concrete implementation of *ReportSourceASCII* to *File*. The input file path is provided as a constructor input parameter in the *File* class. For instance it is possible to use the file generated in 4.2.2 (*ais_log.txt*). In this case, it is also necessary to change the type of *Parser* from *ParserNMEALocalTime* to *ParserAISNMEATimeStampedEpoch*. The latter uses the time stamp in epoch time format at the end of each line of text instead of the time of the system clock. The code below relays the data from the file to a TCP stream on port

[9] A simple free tool for open a raw TCP connection is PuTTY, http://www.putty.org/

19009. To view the output data use the same tool as in 4.2.4. Moreover, for simplicity, all the modifications to the code from the previous paragraphs have been cleaned up.

```
using CMRE.IO.Reports.InputStream;
using SACore.Information;
using CMRE.IO.Reports.MessageParser;
using CMRE.IO.Reports.OutputStream;
using CMRE.IO.Reports.OutputStream.ReportReceiver;
using CMRE.IO.Reports.OutputStream.LineReceiver;
using CMRE.IO.Reports.Filter;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
            Parser<IReport> parser = new ParserAISNMEATimeStampedEpoch
<IReport>(1, "AIS playback");

            ReportSourceASCII<IReport> source = new
File<IReport>(@"d:\tmp\ais_log.txt", parser);

            TCPServerLineReceiver tcpRelay = new TCPServerLineReceiver(19009);
            source.SetLineReceiver(tcpRelay);

            source.Start();
        }
    }
}
```

### 4.2.6 Change the common data format

It is possible to change the common data format generated by the *ReportSource* objects by just changing the actual value of the generic type parameter T of all the classes involved. One could be interested in changing the common data format mainly for two reasons:

- Generate specific data types: one might want a specific implementation of *IReport*
- Performance: some types use less resources than others

For instance, it is possible to generate a *KineticReport* [6] instead of *IReport*. This implies that all the messages without kinematic information are discarded. Note also that *KineticReport* carries a minimal information set (just position, time and ID). The code below changes the example of 4.2.1 to generate a *KineticReport.*

```
using CMRE.IO.Reports.InputStream;
using CMRE.IO.Reports.MessageParser;
using CMRE.IO.Reports.OutputStream;
using CMRE.IO.Reports.OutputStream.ReportReceiver;
using CMRE.MSA.Reports;

namespace SimpleFileLogger
{
    public class Program
    {
        static void Main(string[] args)
        {
```

```
            Parser<KineticReport> parser =
               new ParserNMEALocalTime<KineticReport>(1, "San Francisco AIS");
            ReportSourceASCII<KineticReport> source =
               new TCP<KineticReport>("hd-sf.com", 9009, parser);

            Receiver<KineticReport> dataReceiver =
               new FileReceiver<KineticReport>(@"d:\tmp\ais_parsed.txt");
            source.SetReceiver(dataReceiver);
            source.Start();
        }
    }
}
```

Note that the *Parser* may raise a run-time exception when it is created, because there is no report converter for the actual value of the generic type parameter T.

# 5 Database Access Layer

## 5.1 Introduction

This chapter describes the Database Access Layer (DAL), which encapsulates all of the database access logic inside an assembly (namespace *CMRE.Data.Sql*) in order to keep the database structure independent from the business logic, as recommended by software design best practices. Software engineering literature contains numerous examples of methods and tools to achieve this result, ranging from Object Relational Mapping (ORM) tools (see [9] and [10]) to more modern methodologies (as described in [11]). The CMRE MSA team decided to implement a simple custom solution, avoiding ORM (as suggested in [12]), mainly to be able to write direct SQL queries with the goal of achieving better performance, and to make the transition from OLTP to OLAP completely transparent, depending on the queried time range.

## 5.2 Class structure

The base class of the hierarchy is *Statement*, which encapsulates the creation of the *SqlConnection* to the OLAP and OLTP databases. Database information, such as the connection string, is stored in *DatabaseInfo* objects. This class is also responsible to create the *SqlCommand*, even if the SQL statement construction is delegated to the subclasses via the virtual method *Sql()*. *Statement* implements *IDisposable* [13] and it is responsible for releasing the *SqlCommand* object and the *SqlConnection* to the ADO.NET connection pool. The subclasses of *Statement* encapsulate three types of functionalities:

- Implementation of the SQL select statement (*Query*)
- Implementation of a simpler query that returns a scalar value (*ScalarQuery*)
- Implementation of SQL non-query instructions such as insert and update (*Command*)

These three functionalities are implemented respectively by *MultiTableQuery*, *MultiTableScalarQuery* and *MultiTableCommand*, which encapsulate the logic for handling the SQL operations on multiple tables split over time (as explained in Chapter 2). The knowledge of the splitting granularity is set in the *DatabaseInfo* object by using *TimeSplittingPolicy* objects. These high-level classes are graphically described in the UML diagram of Figure 14: .

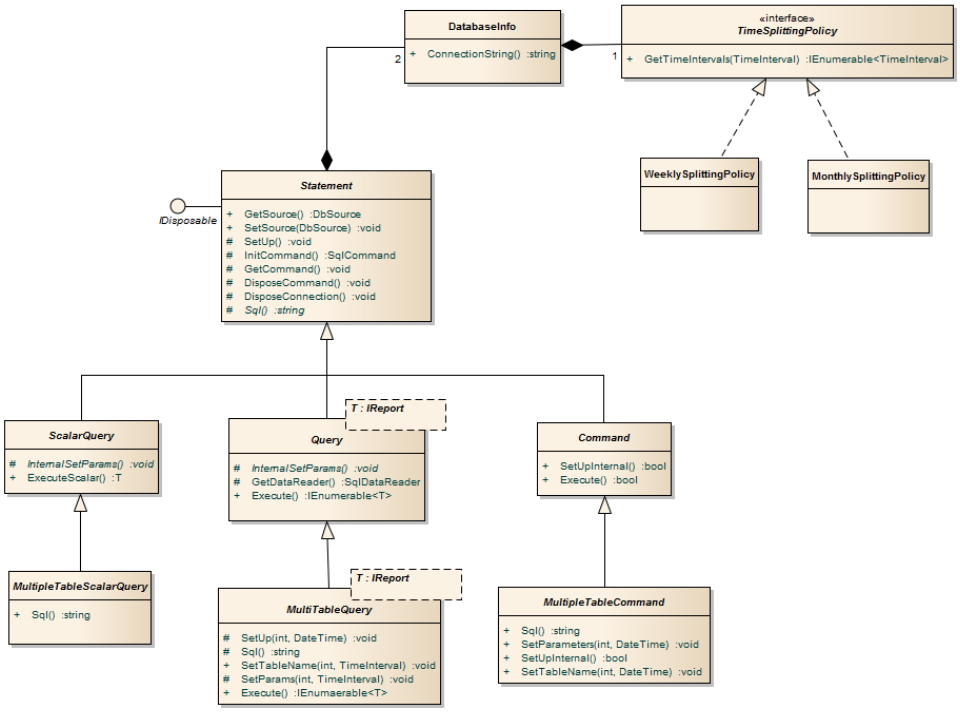


**Figure 14:** UML diagram of high-level DAL classes.

All of the classes described above are abstract, and in order to create a concrete database operation, one should derive from the appropriate super class. A set of predefined concrete classes are available in the assembly *CMRE.Data.Sql.dll*, implementing queries and commands for the CMRE MSA database (see Chapter 2).

Figure 15: shows an UML diagram for the *ExtractAISContactsByTimeAndBBox* class (available in *CMRE.Data.Sql.dll*), which implements the SQL select on the AIS_Contacts tables (see 2.2.1) using a time range and a bounding box restriction in the SQL where clause. Note that the actual SQL statement is inside each concrete class. When the user needs a new database operation, a new sub class needs to be implemented. In this way, users of the framework can extend the number of database functions to their will, without modifying other deployed operations, and thereby complying with the Open-Closed Principle [15].

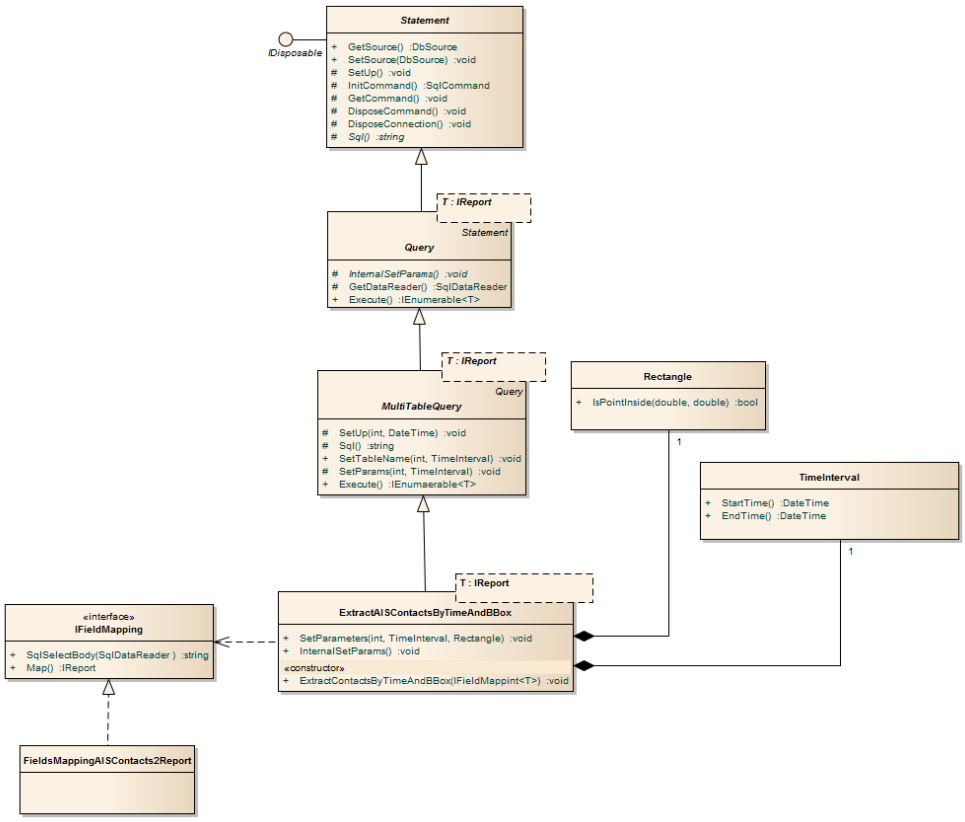


**Figure 15:** Detailed UML class diagram of a concrete query.

All the query operations (*Query* and *MultipleTableQuery*) deliver the results to the user via an *Execute()* method in form of *IReport* [6] or its subclasses. The concrete type of the generated *IReport* is established by the user via the actual value of the generic type parameter T. The mapping between the database record fields and the report object is performed by classes derived from the *IFieldsMapping* interface, which is passed by the user to the concrete *Query* object via a constructor parameter. In future releases, the *Query* objects might find a default *IFieldMapping* implementation using reflection and the actual type of the generic type parameter T. The reports are generated out of the database result-set records, following a lazy-initialization [14] pattern using the C# yield keyword. This approach avoids the allocation of the full result-set in memory, which would otherwise potentially cause out-of-memory run-time exceptions.

In the next two subsections, the use of the DAL is described from two distinct points of view. In 5.3, it is explained how to add a new query to the DAL for a user that wants to extend it. Basic knowledge of the database schema is required. In 5.4, it is explained how to use the queries and the commands for the DAL clients.

### 5.3 How to extend the database access layer

New queries and/or commands can be added to the database access layer by sub-classing the desired abstract class, depending on the query type. For instance let's extend the DAL with a new query that extracts AIS Contacts inside a given time range with speed greater than a given minimum value.

```
using CMRE.Utility.DataStructure;
using CMRE.Utility.EnhancedTypes;

public class ExtractAISContactsByTimeAndSpeed<T> : MultiTableQuery<T>
{
    private TimeInterval timeInterval;
    private double speed;

    public ExtractAISContactsByTimeAndSpeed(IFieldsMapping<T>
_record2ReportMapping)
        : base(
            @"SELECT $fields FROM AIS_Contacts$table_suffix AS c WHERE Time
BETWEEN @startTime AND @endTime AND speed > @speed",
            _record2ReportMapping
        )
    {
    }

    public void SetParameters(int _sid, TimeInterval _timeInterval, double _speed)
    {
        base.SetParams(_timeInterval, _sid);
        timeInterval = _timeInterval;
        speed = _speed;
    }

    protected override void InternalSetParams()
    {
        AddParameter("@startTime", interval.StartTime().ToEpoch());
        AddParameter("@endTime", interval.EndTime().ToEpoch());
        AddParameter("@speed", speed);
    }
}
```

A subclass of *MultiTableQuery* has been created, since AIS are stored in the CMRE database using multiple tables split over time (see Chapter 2). Only two methods are necessary: a public *SetParameters()*, where the query input variables will be passed by the users and will be store in data member variables; and a protected *InternalSetParams()* where the parameters are injected into the prepared SQL statement. Note the constructor where the client will provide the *IFieldMapping* implementation which is compatible with the actual type parameter. The actual SQL statement is passed to the super class constructor call.

### 5.4 How to use the database access layer

This subsection describes the database access layer from the framework user point of view. It is worth noting the simplicity of the public interface of the DAL classes. The inner complexity of database access has been completely encapsulated inside the DAL, enabling the user to accomplish the database query with only a few lines of code. The code below shows the use of the class *ExtractAISContactsByTimeAndBBox*. Note that *CMRE.Data.Sql.dll*, *CMRE.Utility.dll* and *SACore.Information.Basic.dll* need to be linked.

```
using System;
using SACore.Information;
using CMRE.Data.Sql;
using CMRE.Utility.Logging;
using CMRE.Utility.DataStructure.TimeSplittingPolicies;
using CMRE.Utility.DataStructure;
using CMRE.Utility.Topology;


public class Program
{
    public static void Main(string[] args)
    {
        Statement.SetDbInfo(Statement.Source.OLTP, new DatabaseInfo("<your
database connection string here>", TimeSplittingPolicy.Create("Weekly")));
        Statement.SetDbInfo(Statement.Source.OLAP, new DatabaseInfo("<your
database connection string here>", TimeSplittingPolicy.Create("Monthly")));

        using (ExtractContactsByTimeAndBBox<IReport> query = new
ExtractContactsByTimeAndBBox<IReport>(new FieldsMappingAISContacts2Report(),
true))
        {

            query.SetParameters(1, TimeInterval.Last24Hours(),
Rectangle.AegeanSea());
            foreach (IReport r in query.Execute())
            {
                Console.WriteLine(r);
            }
        }
    }
}
```

The initial *Statement.SetDbInfo()* instructions are necessary to initialize the database connections. The C# *using* statement is used to create the query and to ensure that memory clean-up is properly performed calling the *Dispose()* method at the end of the scope, releasing also the database connection to the pool. In the constructor the *FieldsMappingAISContacts2Report* object is passed, coherently with the actual type of the generic type parameter T (*IReport*). Then the query's input parameters are set with *SetParameter()* call. The first *int* parameter identifies the data source ID inside the database. The query is finally executed using the *Execute()* method, as described in Paragraph 5.2. When this program is run the output records are displayed to the console standard output.

# 6 Conclusions

A complete system to support MSA sensor data handling has been presented. The system, designed and implemented at NATO STO CMRE for the MSA project, has been deployed since 2009 and is still under continuous development. This paper demonstrates in this context an effective approach to develop a system able to deliver relevant information from sensors to the information consumer, in timely manner. Furthermore, this implementation is also a real solution for the storage of historical queries which, combined with historical data mining, results to be a step towards Information Superiority in the Maritime domain. Moreover, the system represents an easy data access tool for scientists in the fields of data fusion, target tracking, sensor performance evaluation and anomaly detection.

There are many future developments envisioned:

- Enable the deployment of the system to other NATO bodies, Nations, or research centres. This will require the addition of easy administration capabilities and extension of the array of possible RDBS which can be used.
- Enable the database spatial capabilities and connect to an Open Geospatial Consortium Web Mapping Server and/or Web Feature Server to extend the number of possible clients.
- Integrate with tasks performing data mining and knowledge discovery such as [16] or [19][20]. These activities can benefit from this architecture and the integration of the DAL is envisioned in the near future.

# Acronyms

- SOA: Service Oriented Architecture
- MSA: Maritime Situational Awareness
- MetOc: Meteorological Oceanographical
- ETL: Extract Transform Load
- DAL: Database Access Layer
- AIS: Automatic Identification System
- SQL: Structured Query Language
- OLAP/OLTP: On Line Analytical Processing / On Line Transaction Processing
- VDM: VHF Data-link Message
- DBMS: Data Base Management System

# Acknowledgements

This work is a portion of the body of work being undertaken by the STO-CMRE MSA Project Team in 2013 and collaborators consisting of (in Alphabetical Order):

- Gianfranco Arcieri
- Paolo Braca
- Luigi Bruno (Visiting Researcher)
- Karna Bryan (MSA Project Leader)
- Giampaolo Cimino
- Domenico Ciuonzo (Visiting Researcher)
- Steven Horn
- Giuliana Pallotta (Visiting Researcher)
- Giuseppe Papa (Visiting Researcher)
- Renato Sirola (Consultant - MSA Project)
- Michele Vespe
- Ingrid Visentini (Consultant - NEREIDS Project)

# References

[1] Leavitt, N. "Will NoSQL Databases Live Up to Their Promise?", *IEEE Computer*, Vol. 43, pp. 12-14, Feb. 2010.

[2] "Technical characteristics for a universal shipborne automatic identification system using time division multiple access in the VHF maritime mobile band NMEA AIS", Recommendation M.1371-2, International Telecommunication Union.

[3] "Standard for interfacing marine electronic devices", National Marine Electronics Association, NMEA 0813, version 3.00.

[4] Braca, P., Grasso, R., Vespe, M., Maresca, S., Horstmann, J. "Application of the JPDA-UKF to HFSW radars for maritime situational awareness," in Proc. of the 15th International Conference on Information Fusion (FUSION), Singapore, 2012.

[5] Transaction Processing Performance Council. http://www.tpc.org

[6] Horn, S. Arcieri, G., Millefiori, L., Cimino, G. "Information interoperability: lessons learned on interoperability standards in fusion and tracking. CMRE Formal Report CMRE-FR-2014-01, 2014.

[7] Arcieri, G., Cimino, G., Horn, S., Bryan, K. "Web service interoperability in a network-enabled environment", CMRE Formal Report CMRE-FR-2014-003, 2014.

[8] Richter, J. "CLR via C#", Microsoft Press, Third Edition, Feb. 2010.

[9] Perkins, B. "Working with NHibernate 3.0", Wrox, First edition, Sep. 2011.

[10] Lerman, J. "Programming Entity Framework: Building Data Centric Apps with the ADO.NET Entity Framework", O'Reilly Media Second Edition, Aug 2010.

[11] Dapper.Net. http://code.google.com/p/dapper-dot-net/.

[12] Neward, T. "The Vietnam of Computer Science", Ted Neward's technical blog, 26 June, 2006.

[13] Duffy, J. "DG Update: Dispose, Finalization, and Resource Management" Joe Duffy Technical Blog, 8 April, 2005.

[14] Fowler, M. "Lazy Initialization", http://martinfowler.com/bliki/LazyInitialization.html.

[15] Meyer, B. "Object Oriented Software Construction", Prentice Hall, 1988.

[16] Baldacci, A., Fabiani,, A., Giannecchini, S. “Contact-based AIS coverage estimation and distribution”, NURC Memorandum Report NURC-MR-2008-001, March 2008.

[17] University of Hamburg HF-Radar Home Page. http://ifmaxp1.ifm.uni-hamburg.de.

[18] Helzel Messtechnik GmbH. http://www.helzel.com.

[19] Vespe, M., Pallotta, G., Visentini, I., Bryan, K., Braca, P. “Maritime Anomaly Detection based on Historical Trajectory Mining”, NATO Port and Regional Maritime Security Symp. Research and Technology Organization, Lerici (SP), 2012.

[20] Vespe, M., Visentini, I., Bryan, K., Braca, P. “Unsupervised Learning of Maritime Traffic Patterns for Anomaly Detection, 9th IET Data Fusion and Target Tracking Conference, London, 2012.

# Document Data Sheet

| Security Classification | | Project No. |
|---|---|---|
| NATO UNCLASSIFIED | | MSA |
| **Document Serial No.** | **Date of Issue** | **Total Pages** |
| CMRE-FR-2014-017 | October 2014 | 45 pp. |


**Author(s)**

Cimino, G., Arcieri, G., Horn, S., Bryan, B.


**Title**

Sensor data management to achieve information superiority in maritime situational awareness.


**Abstract**

This report describes the data handling process set up at the NATO Science & Technology Organization (STO) Centre for Maritime Research and Experimentation (CMRE) which includes sensor data acquisition, processing, storage and access in support of the Maritime Situational Awareness (MSA) project. The Database Management System (DBMS) and the way in which sensor data is acquired and loaded using a database access layer framework for client applications is described. The system has been designed and developed to cope with extremely large data volumes generated by sensors and it is the foundation for supporting the CMRE MSA Service Oriented Architecture and the Fusion on Demand concept. Many aspects of this system are then analyzed: data sensor parsing, real-time database loading, database structure, database data extraction (real-time and historical). This analysis is supported with performance figures for the use of the system with real data sets. This analysis demonstrates that the system is an effective way to deliver relevant information to MSA decision makers. The whole system is currently deployed at CMRE.



**Keywords**

MSA, Database, DBMS, SQL Server, AIS, ETL, OLAP, OLTP, Radar, C#, Big Data, Microsoft .Net, Information Superiority



**Issuing Organization**

Science and Technology Organization
Centre for Maritime Research and Experimentation

Viale San Bartolomeo 400, 19126 La Spezia, Italy

*[From N. America:*
STO CMRE
*Unit 31318, Box 19, APO AE 09613-1318]*

Tel: +39 0187 527 361
Fax:+39 0187 527 700

E-mail: library@cmre.nato.int