Scientific Information Management in Collaborative Experimentation Environments

Kaletas, E.C.

Citation for published version (APA):
Chapter 5

Management of Information in the VLAM-G Experimentation Environment

Chapter 4 defined a framework and generic/reusable methodologies for information management in collaborative experimentation environments (CEEs). In addition to modelling the information related to scientific experiments in a CEE, Chapter 4 also provided description and modelling of the required CEE functionality for managing the experiment-related information. The models defined for data and functionality are both based on the experiment model, which was also introduced and defined in the previous chapter. Furthermore, a user environment was described to tackle the challenges of how to present the experiment, its related information, and the functionality provided by the CEE to its users.

The information management framework defined in Chapter 4 is generic and it can be implemented in different ways. The main subject of this chapter is to present one specific application of the information management framework described in the previous chapter and its implementation within the VLAM-G project; namely the Virtual Laboratory Information Management for Cooperation (VIMCO). VIMCO is the information management platform designed and implemented within the Grid-based Virtual Laboratory Amsterdam (VLAM-G) project, which was briefly introduced in Subsection 1.5.1.

The remaining of this chapter is organized as follows. The chapter first revisits the VLAM-G experimentation environment, describes its experiment model and user environment (Front-End), and provides an example user session that includes the typical user activities within the VLAM-G. Then the chapter focuses on the description of VIMCO first by providing an overview, and then describing its architecture, imple-
mentation, databases, and services. Since this chapter presents an implementation of the framework described in the previous chapter, several cross-references are provided throughout the text to the previous chapter.

5.1 VLAM-G Experimentation Environment – Revisited

The VLAM-G is a virtual laboratory environment supporting multiple-disciplines. It was described in more details in Subsection 1.5.1. In this section, the design and implementation of the VLAM-G experiment model and VLAM-G Front-End are provided. The experiment model adopted by the VLAM-G is an implementation of the experiment model introduced in the previous chapter, and the Front-End is an implementation of the user environment also described in the previous chapter.

5.1.1 Experiment Model of the VLAM-G

The experiment model presented in Section 4.1 is adopted by the VLAM-G with a different naming convention. Scientific experiments in VLAM-G consist of three components, namely Process Flow Template, Study, and Topology; respectively corresponding to the Experiment Procedure, Experiment Context and Experiment's Computational Processing components of the experiment model described in Section 4.1. This is summarized below:

\[
\text{Process Flow Template} \Leftrightarrow \text{Experiment Procedure} \\
\text{Study} \Leftrightarrow \text{Experiment Context} \\
\text{Topology} \Leftrightarrow \text{Experiment's Computational Processing}
\]

Since the experiment model was described in details in the previous chapter, this section outlines only the main aspects of the VLAM-G experiment model. A Process Flow Template (PFT) is a step-wise description of a certain type of experiments and contains the steps involved in a typical experiment of this type. PFTs are designed by domain experts and serve as a template for scientists during experimentation. A Study is an instance of a PFT. It provides information about a particular experiment, by describing the data elements and activities involved in the experiment. The computational aspects of scientific experiments are represented as a Topology in VLAM-G. A topology consists of a number of 'software modules' attached to each other to form a data flow, which is intended to solve a particular problem.

5.1.2 VLAM-G User Environment: Front-End

The Front-End is the user environment of the VLAM-G. Users interact with the VLAM-G only through the Front-End. It consists of a number of graphical user interfaces (GUIs), which present the experiment-related information managed by VIMCO and the functionalities provided by VLAM-G and VIMCO to users in a uniform way.
Recall here that although the author was involved in the design activities, the architectural design and the actual implementation of the Front-End were realized by other VLAM-G members (see ‘VLAM-G Development’ in Subsection 1.5.1).

GUIs of the Front-End include three editors: PFT Editor, PFT Viewer, and Topology Editor. Interfaces for log-in process, for displaying the available services, and for registration of new software modules to VLAM-G are the other GUIs in the Front-End.

The PFT Editor supports basic PFT manipulation functionality, such as designing new PFTs and browsing and/or updating existing PFTs. A PFT defines an experiment from a specific domain, and the PFT Editor is available only to domain experts who have extensive knowledge in their domains. When designing a new PFT, the PFT Editor displays the types in the database schema selected by the domain expert. The domain expert defines the PFT using the database schema types. During the PFT design process, the PFT Editor assists the expert by, for instance, inserting compulsory elements, displaying the relationships between elements, connecting elements to each other, and generating and holding information about the graphical representation of the PFT. Figure 5.1 shows the PFT Editor, used for designing the Material Analysis PFT. In the upper-left side of the editor window, a list of the data types in the material analysis database schema can be seen. Information about the selected PFT element (in this case corresponding to the MaterialAnalysis class) is displayed in the lower-left side of the editor window.

The PFT Viewer is an editor for instantiating PFTs (i.e. for making studies) and for browsing and modifying existing studies. When a user logs-in to the VLAM-G,
s/he is presented with a list of PFTs to make a new study and a list of existing studies. If the user selects to make a new study, the PFT corresponding to that type of study is loaded from VIMCO and displayed in the PFT Viewer. If the user selects an existing study, that study is loaded from VIMCO together with its PFT, and displayed in the PFT Viewer. The graphical information for the proper display of a study on screen as well as other necessary information related to the problem domain are held by the PFT, hence, a link is maintained between a study and the PFT that is used to make that study. Clicking on any step in the study displays the detailed information about that step in a form. Users make use of this form to create a new instance of a study step, or to browse and modify the information about a step. The form contains the attributes and their values for that step. Depending on the visibility policies specified for the attributes, the PFT Viewer may decide to filter out some of the attributes. The viewer assists the user to connect the steps in a study to each other. This is not done automatically, because, a study step may contain more than one instance at a time, and it is not possible to automatically determine which instance must be connected to the next/previous step instances. This occurs, for instance, when a study contains a parallel data analysis operation, where two analysis step instances generate two data step instances (as output files). In this case the user must manually link the analysis to its output. Furthermore, users can enter the query mode for retrieving already existing instances of a step from the database by clicking on the query button for that step. The query mode displays a form containing the attributes of that step, fill-in boxes to specify the condition values for attributes, and the operators for the condition. When the user enters the query conditions, PFT Viewer formulates the query in SQL, and submits it to VIMCO. The query results returned by VIMCO also include the PFT information, which is used by the viewer to properly display the results. Figure 5.2 shows the PFT Viewer used for creating a new material analysis study. The outlook of the viewer window is the same as the PFT Editor, except that the PFT Viewer does not include a list for showing the data types in the database schema. Information about the selected study element is however displayed on the left side of the viewer window (in this case the MaterialAnalysis instance).

The Topology Editor allows users to compose their topologies using available software modules, and save and execute topologies. The Topology Editor is launched by the Front-End when the user clicks on a study step in the PFT Viewer which is marked as an RTS process (i.e. as a processing step). Upon initiation, the Topology Editor loads the descriptions of the software modules from VIMCO that this user is authorized to use. A module description contains information about its functionality, its run-time requirements (e.g. CPU requirements), brief description and detailed manuals, input and output ports, etc. Users create instances of the modules by dragging and dropping them into the working area of the Topology Editor, and set the actual parameter values. By connecting the input/output ports of the modules, users define a data flow, which can be saved and executed. The Topology Editor also allows users to display execution results, as well as to retrieve status information and change execution parameters at run-time. The Topology Editor is shown in Figure 5.3, used for designing a topology for material analysis. In the upper-left side of the editor window, a list of available software modules can be seen. Information about
the selected module (in this case *apodization*) is displayed in the lower-left side of the editor window.

### 5.1.3 Experimentation in the VLAM-G

This section provides an example user session to illustrate the typical activities performed by a user within the VLAM-G, as depicted in Figure 5.4. A user logs-in to the VLAM-G by presenting her/his certificate using the Front-End log-in dialog. If the user presents a valid certificate, a new Session Manager instance is created, which then contacts VIMCO for the authentication of this user as a valid VLAM-G user. If the user is authenticated after this two-level authentication, VIMCO generates a list of available services that this user is authorized to use. The list contains the types of studies that this user is allowed to make, existing studies that this user is allowed to browse/modify, and active sessions at the time of login. Different types of studies are represented as PFTs. Users make new studies by creating instances of these PFTs within the VLAM-G. If the user is an expert in her/his domain, the list also includes PFTs that this expert is authorized to manipulate. Each item in the list contains a name and a brief description, which helps the user to find and select the service that s/he needs.

The user then chooses either to make a new study by selecting its template, to work on an existing study by selecting that study from the list, or to join an active session by selecting that session from the list. For the latter case, this user will be registered as a participant of the collaborative session that s/he joined (implementa-

![Figure 5.2: Snapshot of the PFT Viewer](image-url)
tion of collaborative sessions is planned for future work). For the first two cases, a new session is created and this user is assigned as the owner.

This example assumes that the user selects a PFT to make a new study. As soon as the user selects the PFT, it is loaded and presented to the user in the Front-End. The user goes over the PFT steps and fills-in the required information for each step. This way, s/he fills-in the PFT and saves the study.

Some of the steps in the PFT actually correspond to computational processes; for instance, analysis of a data set generated by a laboratory instrument. In this case, after filling in the required information for this analysis step, the Topology Editor is launched, with a list of software modules that s/he is allowed to use. S/he defines a topology by dragging-dropping modules from the list and connecting them to each other, and executes it by sending the topology to the VLAM-G RTS. At the same time, the topology is saved in VIMCO, and a link is created from the analysis step to this topology.

When finished with the current study, the user either logs-out from VLAM-G or continues by re-displaying the list of available services. In the former case, the session is terminated and VIMCO releases all resources allocated for this session (e.g. closes all database connections). In the latter case, Session Manager updates the session information and persists it in VIMCO.

Figure 5.3: Snapshot of the Topology Editor
VIMCO: Virtual Laboratory Information Management for Cooperation

VIMCO is the information management platform of the VLAM-G. In addition to data modelling for diverse types of experiment-related information handled in the VLAM-G, VIMCO provides several mechanisms for platform independent database access, distributed and multi-threaded manipulation of these diverse types of experiment-related information, and XML-based information exchange facilities. VIMCO is an implementation of the information management framework presented in Chapter 4. Therefore, VIMCO provides the functionalities described in that chapter to its users.

VIMCO is designed and developed as an integral part of the VLAM-G architecture. The main goal behind the VIMCO development is to provide a scalable and flexible platform for managing the wide variety of experiment-related information, also considering the requirements listed in Section 2.6. Below, a summary of the main requirements for VIMCO is given:

- Modelling wide variety of information related to scientific experiments.
- Providing mechanisms for the management of such information.
• Designing and developing a scalable and flexible platform that is open for adding new users and resources, and open for adding new types of information and components to support the management of new information.

• Designing and developing an open platform that do not depend on any third party products.

In this direction, the design principles set for the design and development of VIMCO can be summarized as follows:

2. Using standards as much as possible.
4. Distributed, multi-threaded, and modular architecture.
5. Clear, consistent, and generic interfaces that each module declares for its functionality.

The rest of this chapter describes the VIMCO design and development in details, which address the requirements and follow the design principles as described above. Note here that all architecture and data model diagrams provided in this chapter are object-oriented and use UML notation.

5.3 The VIMCO Architecture

The VIMCO architecture has been designed to support information management in the distributed VLAM-G environment. The diagram in Figure 5.5 provides an overview of the VIMCO architecture. The figure shows the VIMCO architectural components, the interactions among them, and their distribution over several nodes (on which the components run).

The VIMCO architecture is open for extensions and improvements in the future. This is achieved through the modular architecture, the standardized and generic interfaces defined by components, and usage of driver-components for third-party software packages. One advantage of standardized and generic interfaces and usage of driver-components is that they reduce/remove dependencies among components as well as they remove dependencies to third party solutions. For instance, developing driver-classes for different XML generator/parser packages that comply with the base XML Manager allows for plugging-in any XML package. The driver-components approach is similar to the mediator approach.

Components of the VIMCO architecture are grouped into three different types of servers that are dispersed over three different types of nodes (see Figure 5.5):

• VIMCO Communication Servers
• VIMCO Core Functionality Server
5.3. The VIMCO Architecture

In the rest of this section, VIMCO components will be described in details.

5.3.1 VIMCO Communication Servers

VIMCO Communication Servers are responsible for the communication between VIMCO and the outside world (i.e. other VLAM-G components). Within the general VLAM-G architecture, no direct user access to VIMCO is permitted. The only client of VIMCO is the VLAM-G Session Manager, which acts as a broker in the VLAM-G
The communication servers present the VIMCO functionality to the outside world through interfaces based on different communication mechanisms. Three different communication servers have been developed for VIMCO: RMI Server, Activatable RMI Server, and Servlets.

Although all VIMCO communication servers use the VIMCO Core Functionality Server as the back-end, having separate communication servers enables future modifications to the VIMCO Core Functionality Server by decoupling it totally from the client programs. Hence, effects of possible future changes in the VIMCO Core Functionality Server to clients are minimized.

**VIMCO RMI Server** is the RMI interface of VIMCO. The current design of VLAM-G Session Manager uses the VIMCO RMI Server to communicate with VIMCO. It forwards the arriving requests as RMI calls to the VIMCO Core Functionality Server, serializes the results into XML documents, and returns the XML documents back to the requester.

**VIMCO Activatable RMI Server** provides an RMI-based communication between VIMCO and the outside world. The difference between this server and the VIMCO RMI Server is the following: In order to keep the VIMCO RMI Server process alive, the Java Virtual Machine process must also be kept alive. That is, the VIMCO Server must be continuously active and running in the foreground. However, Activatable RMI Server works as a daemon in the background, and is activated only when there is a request. Thus, the Activatable RMI Server enables the VIMCO Server to be activated on demand.

**VIMCO HTTP Server** provides an HTTP/HTTPS-based communication. The VIMCO HTTP Server has the capability of understanding HTTP requests, converting the message-based requests into calls to VIMCO Core Functionality Server, and sending the results back using HTTP. The requests are formulated as messages over HTTP, while the actual data is transferred as XML. A protocol has been defined for the HTTP-based communication between VIMCO and the Session Manager [70]. The protocol messages have the following generic structure:

```
<sessionId>, <type>, <messagename>, <attributes>
```

In this protocol, *sessionId* indicates to which session the message belongs. Possible values for *type* are the following: ‘request’, ‘response’, ‘update’ and ‘inform’. *messagename* is the name of the message. *attributes* contain the relevant information for the message in XML format.

The reason for using XML-based message content is that intermediate communication layers will be isolated from the possible changes to the content or format of XML messages.

### 5.3.2 VIMCO Core Functionality Server

VIMCO Core Functionality Server is the main component in VIMCO. It provides the functionality to support a VLAM-G user throughout her/his experimentation, start-
5.3. The VIMCO Architecture

ing from presenting the available VLAM-G services until log-out. The components in the VIMCO Core Functionality Server work together to process the user requests that may arrive during a session.

VIMCO stores the session information in a persistent storage, while the VLAM-G Session Manager is responsible for the management of sessions. A session contains information about the active user in the session at any time, the study that the active user is working on, PFT of the study, the list of topologies that were submitted to the RTS during this session, and other users collaborating with the active user for the study. VIMCO Core Functionality Server uses the session information to process user requests. For instance, the user information in a session object is used to enforce the access rights defined for the active user.

Following describes the components in the VIMCO Core Functionality Server.

VIMCO Server Manager

The VIMCO Server Manager component is used by the VIMCO Administrator only during system initialization and system shutdown. It provides two functionalities: Startup and shutdown. During startup, VIMCO Server Manager connects to the VIMCO DB and reads the necessary information needed for the proper operation of the VIMCO Core Functionality Server, and initializes the internal data structures maintained by the Lookup Server. This information includes the available databases and the mappings between the databases and their contents. The Connection Manager and RMI Server are also initialized by the VIMCO Server Manager at startup time. During shutdown, several operations are performed for a clean shutdown of the system, e.g. all open database connections are closed.

The VIMCO Server Manager consists of the following classes and interfaces: The IVimcoServerManager interface defines the two methods that VimcoServerManager class implements, which performs the actual startup and shutdown operations. The VimcoServerLauncher is a password-protected graphical interface for administrators to startup and shutdown the system.

Lookup Server

Lookup Server maintains lookup lists in the memory for fast access at run-time, containing information about the databases, active sessions, users and access rights, and the mapping between a database and its contents. The Lookup Server maintains a number of lists at run-time. The database list and the mapping between the database ID and the content ID are initialized by the VIMCO Server Manager at startup time. Session information, however, is inserted into the session list at run-time by the VLAM-G Session Manager using VIMCO services. The methods provided by the Lookup Server enable the other VIMCO Core Functionality Server components to lookup for information based on keys, such as the ID of a database. For example, Lookup Server is used to obtain a connection to a VIMCO database as follows: A request to read a study contains the session ID. VIMCO looks up for the session information for the given ID, from which it obtains the database ID holding the
study. The database ID is then used when obtaining a connection to the database from the Connection Manager.

**VIMCO Server**

The main component in the VIMCO Core Functionality Server is the VIMCO Server. All VIMCO services representing the overall VIMCO functionality are implemented by the VIMCO Server. It also coordinates the activities of other components in the VIMCO Core Functionality Server in order to process a user request. For instance, when a request to save a study arrives to VIMCO Core Functionality Server, the VIMCO Server will first map the session information into actual database information using the session information and the other internal information provided by the Lookup Server, and contact the Connection Manager to get the correct data manager. Then it starts a transaction in the corresponding application database, saves the study, and commits the transaction using the Connection Manager. Every VIMCO service is executed within a transaction. Transactions are closed after the completion of the service execution.

`VimcoServer` is the main class in the VIMCO Server component that implements the VIMCO functionality. It contains read, write, delete, and write-as methods for different types of information managed by VIMCO, including for instance, PFTs, studies, and session information. Among other classes in the VIMCO Server, `VimcoServerConstants` defines the constant values used by all components in VIMCO, such as the reuse policy values and the attribute properties. The `Session` class for representing the session information is also defined as part of VIMCO Server. Finally, VIMCO Server contains a number of ‘info’ classes, each of which provides a summary of the actual information in the VIMCO databases. VIMCO services are described in details in Section 5.5.

**XML Manager**

XML is used for the data transfer among the VLAM-G components. Since the VIMCO Server is object-oriented, XML Manager is used to serialize Java objects into XML documents and to de-serialize XML documents back into Java objects. The XML Manager provides a standardized interface, so that the underlying XML package can be replaced by another package at any time if need occurs. The same XML Manager is used also by the Front-End and the Run Time System. Currently, besides the Vimco-XML package, two third-party XML packages called JSX [166] and ElectricXML [167] are available for Java–XML conversion.

The XML Manager consists of the following classes: `XMLManager` class provides two methods for Java–XML conversion: `writeXML` for serializing Java objects into XML documents, and `readXML` for de-serializing XML documents into Java objects. The `XMLManager` class itself does not perform the actual conversion. It uses one of the available XML drivers for conversion, which is imported at compile time. For each of the available XML packages, a driver class is developed, which declares the same methods as the `XMLManager` class. The methods make the necessary calls to the under-
5.3. The VIMCO Architecture

### Comparison of ElectricXML (EXML) and VimcoXML

<table>
<thead>
<tr>
<th>Database name</th>
<th># of objects in the DB</th>
<th>DBMS backup size</th>
<th>Uncompressed EXML</th>
<th>Uncompressed VimcoXML</th>
<th>Compressed EXML</th>
<th>Compressed VimcoXML</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIMCO DB</td>
<td>1075</td>
<td>16632</td>
<td>356</td>
<td>218</td>
<td>11</td>
<td>7.5</td>
</tr>
<tr>
<td>RTS DB</td>
<td>869</td>
<td>14656</td>
<td>624</td>
<td>392</td>
<td>76</td>
<td>62</td>
</tr>
<tr>
<td>Project DB</td>
<td>30</td>
<td>1744</td>
<td>28</td>
<td>22</td>
<td>2.7</td>
<td>2.1</td>
</tr>
<tr>
<td>MACS DB</td>
<td>1136</td>
<td>9624</td>
<td>657</td>
<td>436</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>Expressive DB</td>
<td>157557</td>
<td>32304</td>
<td>140000</td>
<td>63000</td>
<td>6900</td>
<td>4400</td>
</tr>
</tbody>
</table>

Table 5.1: Comparison of the size of XML documents generated by ElectricXML (EXML) and VimcoXML (all file sizes are given in KB)

lying XML package for the serialization/de-serialization of Java objects. The `XML` data type is used to represent XML documents as Java objects; XML managers receive and generate XML documents as instances of this type. Its `value` attribute contains the contents of the XML document as a Java String. Following a mediator-like approach breaks the dependency of the VIMCO components to the used conversion package.

All three packages mentioned above have been used and tested in VIMCO. At the time of testing, JSX did not provide support for all the Java types that were used in VIMCO. ElectricXML is a much more robust tool, however, the XML document generated by ElectricXML contains too much overhead mainly due to the XML Schema used by the tool. To overcome this overhead, VimcoXML has been developed. The XML document of VimcoXML only contains the full class name, and tags corresponding to the attributes of the object being serialized and their values.

A test was made to compare the sizes of the XML documents generated by ElectricXML and VimcoXML. The test was to backup the VIMCO databases by serializing their contents into XML documents. Table 5.1 shows the test results. For each database, number of objects in the database, size of the backup file generated by the DBMS itself, and the uncompressed and compressed file sizes of the XML documents generated by the two packages are given. In all cases, VimcoXML generated smaller files than ElectricXML. When compressed, the VimcoXML files are even smaller than the DBMS backup files.
Connection Manager

Connection Manager uses the available database drivers to provide connections to other VIMCO Core Functionality Server components for accessing the VIMCO databases. A database driver is a library to manipulate the data stored in a specific DBMS. The Connection Manager opens a connection to the specified database and manipulates the data in that database using the corresponding database driver.

On the contrary to transactions, connections to the VIMCO databases are kept open during many user requests. Because the access rights are defined for individual users and their enforcement requires the user ID, connections to the databases must be opened by the users themselves. Therefore, performance improvement mechanisms requiring an anonymous access to databases, such as connection pooling, can not be applied. Instead, *connection caching* is used as an alternative mechanism for reducing the overhead introduced by opening and closing connections, where connections are kept open during the lifetime of a user session. Open connections are cached based on the user name, database name, and the current session ID. The first time a user requests a connection to a database within a session, a new connection is opened for this user. This connection is returned back to the requester (i.e. the VIMCO Server), and at the same time it is cached internally. When the same user requests a connection to the same database still in the same session, Connection Manager returns the cached connection. When a session ends, VIMCO Server issues a remove command to the Connection Manager to close and remove any cached connections opened to the specified database within that session.

The Connection Manager component consists of a manager class, a standardized interface to different DBMSs, and drivers implementing this interface. The *IConnection* interface defines the basic database connection and data manipulation methods, such as connect/disconnect, start/commit/abort transactions, read/write/delete objects, and execute queries. Two drivers have been implemented until now, both for the Matisse object-oriented database management system (ODBMS) [168]. The *JMatissse* driver is implemented on top of the Matisse C API, using Java Native Interface (JNI). JMatissse uses persistence by reachability for reading and writing database objects. That is, given an entry point object, all objects reachable from the entry point object (through navigating the relationships) are persisted to the database or retrieved from the database. JMatissse defines *depth* to overcome the object boundary problem. The depth specifies the number of recursions during the traversal of the object graph (i.e. the depth of the object graph to be traversed). *MtsJava* driver, on the hand, uses indirect object pointers to overcome the object boundary problem. MtsJava is implemented using the Matisse Java API.

*ConnectionManager* is the class that implements the connection management and caching functionality, as described above. It maintains a list of open connections based on the session ID and the database ID. Connections can be closed one by one upon request, or all connections in a session can be closed at session termination. *ConnectionManager* also provides an *init* method, which is used during VIMCO startup to open connections to the VIMCO DB to initialize the Lookup Server.
Log Manager

In order to log the errors occurred and trace the execution flow during run-time, a logging system is developed. Several exceptions have been defined to represent the errors that may occur during the VIMCO execution, and traces have been identified that correspond to the important points in the execution flow. These errors and traces are prioritized with a level number, specifying the importance of the error or trace. The level of required error or trace log is set at startup time by the administrator, and error messages and traces for which the level number is above the specified value are logged in two separate files at run-time. LogManager was especially useful during the software development phase.

5.3.3 VIMCO DB Servers

VIMCO DB Servers correspond to the VIMCO databases. Currently, all VIMCO databases are developed using the Matisse ODBMS [168]. VIMCO databases are described in Section 5.4.

5.3.4 Implementation of VIMCO

Several technologies have been used for the development of the different VIMCO components, as shown in Figure 5.6. Entire VIMCO is implemented in Java, except the VIMCO RTS Module which is implemented in C/C++.

The RMI and RMI Activatable technologies and the RMI Registry tool of Sun Microsystems are used for the implementation of the VIMCO RMI Servers, while
Java Servlets and Apache Tomcat servlet container are used for the implementation of the VIMCO HTTP Server. As mentioned earlier, ElectricXML and VimcoXML tools are used for the serialization/deserialization of Java objects.

Matisse ODBMS is used for the implementation of all VIMCO databases. Currently, work on using a relational DBMS is ongoing. For database access, mainly the MtsJava library developed on top of the Matisse Java API is used. JDBC is also used for some operations, such as querying Matisse databases.

The VLAM-G RTS, its libraries and the provided support tools are used for the development of the VIMCO RTS module.

VIMCO is currently running on both a Linux cluster and a Sun server. Some reserved nodes of the Linux cluster host the VIMCO Communication Servers, VIMCO Core Functionality Server, and the VIMCO databases.

### 5.4 VIMCO Databases

VIMCO maintains several databases for storing different types of information generated and handled within the VLAM-G. VIMCO databases include the application databases storing experiment-related information, and three other databases: VIMCO DB for internal VIMCO information, Project DB for multi-disciplinary projects, and RTS DB for module descriptions and topologies. In the remaining of this section, VIMCO databases and their contents are briefly described.

**VIMCO DB**

*VIMCO DB* is the database for information that is used internally by the VLAM-G and VIMCO. Contents of the VIMCO DB include user information and access rights definitions, session information, information about available data sources, and other information that is only used by VIMCO for internal purposes such as a counter for unique identifiers.

The UML diagram for the VIMCO DB schema is given in Figure 5.7. VIMCO DB implements the user model defined in Subsection 4.3.3 for managing user information and access rights definitions.

In addition, VIMCO DB schema contains the *DataSource* class to maintain information about the available VIMCO data sources, such as the description of the data source, the host on which it resides, and the driver for accessing the data source and for manipulating its contents. Currently, all VIMCO data sources are database management systems, represented as instances of the *DB* class, which in turn inherits from the *DataSource*.

The *Session* class is defined in the VIMCO DB schema to provide persistence to session information. *Session* class was described in Subsection 4.3.4.

Remaining two classes in the VIMCO DB schema are *Password* and *Identifier*. *Password* is a singleton class with only one instance holding the *common* password for all VIMCO users. To access a database, a user must have an account on the underlying DBMS, with a unique username and password. However, in order to provide single
Figure 5.7: Class diagram for the VIMCO DB Schema

sign-on facility, VIMCO does not require the users to provide a password for accessing VIMCO databases. Since no direct access to VIMCO databases is allowed, defining a single password that is common to all VIMCO users is sufficient. This password is never published to users, but only used by the Connection Manager to connect to the VIMCO databases. Finally, Identifier is the other singleton class that is used to generate unique integer identifiers. There can be more than one type of identifier, each of which is unique only within its own context. For example, currently, only an identifier for session objects is generated and maintained. If needed, more identifiers can be defined in the Identifier class as new attributes.

Application Databases

One database is developed for each application in an experimental science domain. VIMCO currently maintains two application databases: MACS DB for material analysis experiments, and Expressive DB for microarray experiments.
Application databases contain PFTs, studies, and other information that are specific to the application.

For the representation of PFTs, application databases implement the Procedure Data Model, which was described in Subsection 4.3.2. However, as mentioned earlier in this chapter, VLAM-G uses the term ‘Process-Flow Template’ (PFT) to refer to experiment procedures. The PFT Data Model, which is appended to the schema of each application database, is given in Figure 5.8.

As also described in Subsection 4.3.2, application database schemas extend the Experimentation Environment Data Model for modelling the domain-specific experiment-related information. The Expressive DB schema will be described in
Another structure that is included in the application database schemas is the one for representing the link between studies and PFTs. The model of the structure was described in Subsection 4.4.1, however, Figure 5.9 shows the data model with the naming convention used in the implementation.

As described in Subsection 4.4.2 in the previous chapter, the OriginCopy class was defined for storing information about reused objects. For the same purpose, this class is appended to the application database schemas as it is.

**Project DB**

Multi-disciplinary projects consist of experiments from different domains. VIMCO adopts the approach for multi-disciplinary projects defined in Subsection 4.5.1, which uses the general structure of scientific experiments and the project concept as the facilitator.

In order to facilitate multi-disciplinary projects in VIMCO, a specific database for projects called Project DB is developed. All information about existing projects is stored in this database, while the actual experiment information (i.e. studies) is stored in multiple application databases. Representation of multi-disciplinary projects is illustrated in Figure 5.10.

The schema of the Project DB is given in Figure 5.11. The schema consists of two classes: Project class for representing (multi-disciplinary) projects, and ExpXRef class for representing the experiments in projects. A Project object contains high-level information about the project such as its description and start/end dates, and links to the experiments included in the project. The links are actually to the ExpXRef objects, which act as proxies to experiment objects in the application databases. An ExpXRef object contains the ID of the database storing the actual experiment object and the OID of that experiment object.

In the current implementation of VIMCO, application database schemas still contain the Project class, however, these are not instantiated. When writing a study, VIMCO writes all the objects in the study to the corresponding application database except the Project object. Since the project information is not stored in the same application database, the relationships from the Experiment object to its project are set to null. The Project object is then written into the Project DB, and an ExpXRef instance is created as a proxy to the Experiment object that was stored in
the application database. VIMCO then sets the attributes of the ExpXRef object to the correct values, and creates the links between the Project and ExpXRef objects. Similarly, when a read request for a study arrives to VIMCO, the project information for the study is retrieved from the Project DB; or, if a query involving the Project class arrives, VIMCO re-directs the query to the Project DB. Furthermore, ordering of experiments in a project is realized by ordering the ExpXRef objects in the Project DB rather than the actual Experiment objects in the application databases. All these operations are performed by VIMCO transparently to the user. In fact, users are not aware of the existence of the Project DB.

**RTS DB**

*RTS DB* is the VIMCO database for storing information related to the computations performed by the VLAM-G RTS. Module definitions provided by domain experts and topologies defined by scientists are stored in the RTS DB.

From one point of view, topologies can be seen as application data/information, and hence they should be stored in the application databases together with the studies. However, topologies are composed of modules, and a scientist can make use of any kind of module in her/his topology, without being restricted only to those modules that are developed specifically for her/his field of expertise. Also considering that one of the aims of the VLAM-G is to promote inter-disciplinary research, all modules must be available to all scientists, although specific security and access rights are still applicable, for instance to use a very expensive device. Thus, instead of replicating the module definitions in all application databases, a specific database to hold both the module definitions and the topologies is a better approach.

![Figure 5.10: Representing multi-disciplinary projects](image-url)
RTS DB stores the topologies. Since a topology is part of a study, that is, it corresponds to a (number of) processing step(s) in a study, a link is maintained from that/those step(s) in the study to the topology. Currently, this link is implemented as a pointer attribute in the study step, whose value is the topology ID. After the topology is saved in the RTS DB, the PFT Viewer sets the value of this attribute to the correct topology ID.

The RTS DB schema is given in Figure 5.12. This schema implements the data models developed for software entity descriptions and computational processing of an experiment, as described in Subsection 4.3.2. It also includes the data model for user information as defined in Subsection 4.3.3. Hence, the schema diagram in Figure 5.12 consists of three parts (represented using the UML *subsystem* symbol): Part A of the RTS DB schema is depicted in Figure 5.13. Part A includes module descriptions, their input/output ports and data types, required environment variables, parameters, etc. Part B of the RTS DB schema is given in Figure 5.14, which models the topologies with the modules in the topology and their connections to each other. Figure 5.15 shows the part C of the RTS DB schema, which includes the data types for user information.

### 5.5 Functionality/Services Provided by VIMCO

In order to maintain the security and consistency of information, and to provide higher-level functionality to scientists, no direct access to VIMCO databases is allowed. VIMCO provides a number of services to access the databases and manipulate their contents. In this section, first an overview of the VIMCO services is given, followed by their descriptions.
5.5.1 Overview of the VIMCO API

VIMCO API consists of the services declared by the four interfaces shown in Figure 5.16, each offering different levels of functionality. This figure shows the hierarchy of VIMCO interfaces which is based on the different types of users; namely application users (IVimcoUserServer), administrators and domain experts (IVimcoAdminServer), and VLAM-G Session Manager (IVimcoSesManServer). IVimcoServer is the top-level interface, supporting functionality that is available to all types of users, such as reading information about the current user and the type of current user. IVimcoSesManServer interface is presented to the VLAM-G Session Manager, and it extends IVimcoServer interface with methods for session information management and authentication. IVimcoUserServer interface defines methods for ordinary VIMCO users, providing the basic information management functionality to support experimentation in VLAM-G, such as reading and writing studies and topologies, reading PFTs, and executing queries. The IVimcoAdminServer interface is reserved for administrators and domain experts, extending standard user functionality among others with update functionality for PFTs, module descriptions, and user information.

The VimcoRMIServer and VimcoRMIServerActivatable classes both implement these interfaces, and the protocol defined for the VimcoHTTPServer messages covers all the functionality defined by these three interfaces. As mentioned earlier, all VIMCO Communication Servers utilize the VimcoServer as the back-end, which provide the actual functionality.

VIMCO services are briefly described in the following subsections. Note that the notation given in Figure 5.17 is used in the remaining of this section for all figures representing VIMCO services.
5.5.2 Services for Accessing VIMCO

Figure 5.18 shows the services for accessing VIMCO. Users can access VIMCO through one of the RMI Servers or through the HTTP Server. The following describes the steps involved in accessing VIMCO.

In order to access VIMCO, a user must first authenticate herself/himself. For this purpose, Session Manager extracts the distinguished name of the user from her/his Grid proxy, and passes this information to the RMI Server, which forwards the request to the VimcoServer. VimcoServer retrieves the user object with this distinguished
name. If there is a user, then the user ID is returned; otherwise a negative integer is returned.

Session Manager then uses this user ID to retrieve the user type. Depending on the user type, the RMI proxy for VIMCO is casted to the correct interface type (i.e. one of the three interfaces inheriting from the root interface shown in Figure 5.16) to enforce access control for services.

Next step is to retrieve the available services. Available services are presented to
the users as number of info objects (as described in Subsection 4.3.4). The info classes include ProjectInfo, StudyInfo, PFTInfo, and SessionInfo. To obtain the PFT, Project, and Study information, VimcoServer queries each application database and the Project DB to retrieve all instances of PFT, Project, and Experiment classes considering the current user’s access rights, and extracts the information necessary for the info classes. For Session information, the VIMCO DB is queried to retrieve all Session objects. Finally, in addition to the other info objects, one DBInfo object is created for each VIMCO database to be sent to the user, since the database ID is required for the calls to VIMCO (except the Project DB, which is totally transparent to users).

These steps are the same for all users and for all types of VIMCO access. The subsequent steps differ, however, depending on the service that the user chooses.

Note that some of these steps are performed by the Session Manager on behalf of the user, without her/his involvement. Steps for retrieving the user type and casting the interface, and for instantiating a session are not visible to the user.
5.5.3 Services for Session, User and Access Rights Management

This subsection describes the VIMCO services for session information management, user management, and access rights management.
**Session Information Management Services**

VIMCO provides the services depicted in Figure 5.19 for the management of session information. As mentioned earlier, VIMCO only maintains a persistent copy of the session object, and it never modifies the contents of the session object. Actual session management is performed by the VLAM-G Session Manager.

![Session Information Management Services](image)

Figure 5.19: VIMCO Session Information Management Services

**User Management Services**

VIMCO users are the users that can access VIMCO and request one of the VIMCO services. VIMCO maintains information about the registered users, their roles, and restrictions on available data types.

VIMCO implements the user data model that was defined in Subsection 4.3.3. This data model is implemented by each VIMCO database to store the user related information in a homogenous way, excluding the roles and restrictions which are only defined in the VIMCO DB.

The user management services of VIMCO are given in Figure 5.20. Since every VIMCO user is assumed to be employed by an Organization, the organization information must already be in the database. XML document in the user and organization management calls also include the ContactInfo object for the user or the organization being manipulated.

![User Management Services](image)

Figure 5.20: VIMCO User Management Services
The main repository for user information is the VIMCO DB. All user management operations are primarily performed on this database. For instance, in order to create a new user for an application database, the user must be first created in the VIMCO DB. When writing user information in a database other than the VIMCO DB, VIMCO first checks whether this user is already defined in the VIMCO DB, and throws an exception otherwise. Then VIMCO checks whether the username specified for the new user is unique for insertions and whether it already exists for updates. Once the User object is inserted to the database, a user account must be created in the underlying DBMS. Currently, this is manually done by the administrator, though this can be automated by calling the DBMS command from within the writeUser method. If the user information is required in all VIMCO databases, it is the administrator's responsibility to make the necessary calls for each database.

On the other hand, when deleting a user from a database other than the VIMCO DB, the user information is not removed from the database, but only the user account on the underlying DBMS is deleted. If the target database is the VIMCO DB, then the user information is removed from the database, and the user account on all DBMSs are deleted. In the case that a user that was deleted from the VIMCO DB is being defined once more, VIMCO maintains consistency by ensuring that each real user has only one User object in any VIMCO database (since the User objects are not removed from the other databases).

Access Rights Management Services

Access rights information in VIMCO is stored in the VIMCO DB, which implements the model for roles and access rights that was shown in Figure 4.18. Roles that can be assumed by users were described in Subsection 4.5.3. Below, the specific roles defined for current VIMCO users are summarized:

- **Expressive User** as an Application User, that can access the Expressive DB, and read and write microarray information.

- **MACS User** as an Application User, that can access the MACS DB, and read and write material analysis information.

- **VLAM-G Admin** as a CEE Admin, that can create user accounts in VLAM-G and in VIMCO.

- **VIMCO Admin** as an Information Management Admin, responsible for maintaining the information stored in VIMCO DB.

- **Expressive Expert** as a Domain Expert, responsible for maintaining the microarray PFTs in Expressive DB and microarray related module descriptions in RTS DB.

- **MACS Expert** as a Domain Expert, responsible for maintaining the material analysis PFTs in MACS DB and material analysis related module descriptions in RTS DB.
Note here that an application database schema is defined by VIMCO Admin together with an expert from the domain of the application.

- *System*, which has all privileges, is used only during emergency cases.

As described in Subsections 4.3.3 and 4.5.3, access rights for each role are defined as a set of restrictions based on the role definitions given above.

Overview of the VIMCO functionality for access rights management is given in Figure 5.21.

| readRole(in sessionld : int, in dbld : int, in roleld : int) : XML |
| writeRole(in sessionld : int, in dbld : int, in role : XML) : XML |
| deleteRole(in sessionld : int, in dbld : int, in roleld : int) |
| readRestriction(in sessionld : int, in dbld : int, in restrictionld : int) : XML |
| writeRestriction(in sessionld : int, in dbld : int, in restriction : XML) : XML |
| deleteRestriction(in sessionld : int, in dbld : int, in restrictionld : int) |
| addUserRole(in sessionld : int, in dbld : int, in userid : int, in roleld : int) |
| removeUserRole(in sessionld : int, in dbld : int, in userid : int, in roleld : int) |
| addRoleRestriction(in sessionld : int, in dbld : int, in roleld : int, in restrictionld : int) |
| removeRoleRestriction(in sessionld : int, in dbld : int, in roleld : int, in restrictionld : int) |

Figure 5.21: VIMCO Access Rights Management Services

Access rights in VIMCO are defined on the VIMCO data types. In this version of VIMCO, the data types consist of the different types of information managed by VIMCO. Although it is called as data type, a data type can actually consist of several types in VIMCO databases. The VIMCO data types contain PFTs, studies, module definitions, topologies, session information, data source information (databases), user information, access rights (roles and restrictions), identifier, password, and info classes. VIMCO considers all these types as atomic, meaning that, a restriction is defined on the entire study, but not on the individual elements of the study. However, restrictions can also be defined on any data type in a database schema, which are only enforced during query execution.

These data types are used to define the restrictions that are applied to users when requesting a service that manipulates one of these data types. However, VIMCO provides some services which do not work on any of these data types. In order to support access rights on these services, similar mechanisms must be developed for those services. Ideally, this mechanism should be the *service* counterpart of the *data type*, where restrictions are defined for *all* VIMCO services. However, for practical reasons and for simplicity, the current VIMCO version provides access rights management for only a subset of all VIMCO services, and treats them as data types. These services are write objects, get user type, and authenticate user. For each of these services, one restriction is defined per role (if there should be any restrictions).
5.5.4 Services for Managing Experiment-Related Information

Services provided by VIMCO for managing experiment-related information (i.e. PFTs, studies, module definitions, topologies) are described in this subsection.

PFT Management Services

Figure 5.22 shows the VIMCO services for PFT management. A PFT consists of a PFT object, a number of PFTElements and PFTConnections, and one PFTGUI object for each PFTElement and PFTConnection object. PFT manipulation services consider a PFT as an atomic unit. When a PFT is passed to VIMCO, it traverses the PFT graph starting from the PFT object, and inserts all objects into the specified database. Similarly, when reading a PFT, first the PFT object with the specified ID is retrieved from the database, then other objects in the PFT are traversed and retrieved.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPFT</td>
<td>Read PFT from database</td>
</tr>
<tr>
<td>writePFT</td>
<td>Write PFT to database</td>
</tr>
<tr>
<td>writePFTAs</td>
<td>Write PFT with associated links</td>
</tr>
<tr>
<td>deletePFT</td>
<td>Delete PFT from database</td>
</tr>
</tbody>
</table>

Figure 5.22: VIMCO PFT Management Services

In VIMCO, different versions of the same PFT are grouped together. The available services list always contains the latest version of a PFT; hence, new studies can only be created using the latest versions of the PFTs. A PFT is not deleted nor updated as long as there are studies created using that PFT. Domain experts can retrieve older versions of a PFT by specifying the ID of the older version.

Study Management Services

Similar to PFTs, studies are also considered as atomic units. VIMCO services for study management are given in Figure 5.23.

When writing a study, a link is stored for each element in the study that points to the corresponding PFTElement. These link objects are also included in the study XML received from the Front-End as part of the study manipulation calls.

One of the major issues with study management is the boundary of a study. Since an object can be included in more than one study (e.g. a software instance), traversing the links starting from the Experiment object is not sufficient. Current version of VIMCO uses the study-PFT links to retrieve the study elements. Given a study, VIMCO first retrieves the link objects, and obtains the OIDs of the study elements from these objects. Then the study objects are retrieved one by one using the OID.

As mentioned in Subsection 4.4.2, the reuse policy values are used to determine which study elements must actually be removed from the database when deleting a
study. The same algorithm is used when an existing study is saved as a new one, to determine which objects must be copied and which must not.

The `execQuery` method is called by the Front-End, to retrieve the instances of a study step satisfying certain criteria. The query is executed by VIMCO as described in Subsection 4.4.2. Queries in the current VIMCO version are formulated in Matisse SQL.

The `getReuseObjects` method is called by the Front-End when a user issues a query and selects one object from the query result set. Upon this call, VIMCO retrieves the semantically related objects from the database based on the reuse policy values (as described in Subsection 4.4.2). If objects are copied, then one `OriginCopy` instance for each copied object is created and sent back to the user together with the related objects. If the user decides to use a copied object in her/his study, only then the corresponding `OriginCopy` object is persisted in the database.

### Module Information Management Services

VIMCO provides a number of services for module information management (see Figure 5.24). These services are only available to domain experts. Similar to PFTs and studies, a module is also considered as an atomic unit by VIMCO, i.e. a request for module information manipulation contains not only descriptive information about the module itself, but also information about its ports and data types of the ports, required parameters and environment variables, and executables.

### Topology Manipulation Services

VIMCO services for topology management are given in Figure 5.25. These services only work on the `instances` of the module descriptions. In other words, when reading a topology, VIMCO first retrieves the `Topology` object with the specified ID, then traverses all the `Module` objects in the topology following their connections to each other. During the traversal, all relationships of objects in the topology are followed, except those that link the topology objects to the module description objects (i.e. relationships having a name that starts with `hasMeta`). Similarly, when writing a topology, the Front-End does not include the module description objects in the topology XML.
Chapter 5. Management of Information in the VLAM-G Experimentation Environment

VIMCO RTS Module

Users can access VIMCO also through the VLAM-G Run Time System (RTS). The current version of VIMCO provides a single RTS module for querying a Matisse database, which is mainly used for testing purposes. This module accepts host name, database name, user name, password and query as parameters from the user, and directly accesses to the specified database. This is the only case where direct access to VIMCO databases occurs. This module will be replaced with a new one, which uses HTTP to access VIMCO and use XML as the data exchange format. This will allow both the enforcement of VIMCO access rights to requests from the RTS and the usage of more than one database in a request (in comparison to directly accessing a single database at a time).

Utility Services

In order to ease the development and testing of VIMCO and VLAM-G, several utility services and tools have been developed. Two of the utility services are instantiateSession and writeObjects. instantiateSession creates and initializes a Session object using the provided session information. writeObjects is used by administrators, mainly to insert large amounts of data into the application databases for testing the developed VIMCO services.

The utility tools include SchemaClassGenerator which is used to generate Java classes for the types defined in a Matisse database. Another Utility class implements several methods, such as tokenizing a string value, generating keys to be used by the
5.5. Functionality/Services Provided by VIMCO

Although it supports more functionality than the above mentioned utility services and tools, attribute properties can be considered as a utility for the Front-End. Attribute properties are used to direct the Front-End about how to treat attributes of the objects displayed in the Front-End editors. For instance, the `pftGroupId` attribute of the PFT class is set by the VIMCO, and should not be modified by users. Another example is the `oid` attribute, which can be set only by the VIMCO or by the Front-End itself. To handle such situations, the `AttProperty` class is defined (see Figure 5.26), which defines the visibility level for each attribute of a class, handling type of the attribute, and whether the value of the attribute can be set to null. It also defines the name that should be used for display in the Front-End. The possible values for the handling type and access modes are defined in the `VimcoServerConstants` class (which are also shown in Figure 5.27). An attribute may correspond to a file location for which the file selection dialog needs to be displayed, or it can be a date or a list requiring special formatting. Also, a user can read and write an attribute, or may only view the value of the attribute. In some cases, users can not see the attribute at all.

![Notation:](image)

Figure 5.26: Class `AttProperty`
5.6 Conclusions

This chapter presented the architectural design and implementation of the information management framework that was described in the previous chapter. Specifically, first the implementation of the experiment model and the user environment within the VLAM-G experimentation environment is described. Then this chapter focused on the design and implementation of VIMCO, the information management platform of VLAM-G. The VIMCO architecture, databases maintained by VIMCO, and services provided by VIMCO are described, all of which implement the corresponding data or functionality models defined in the information management framework of Chapter 4.

The following can be mentioned among the key characteristics of the VIMCO design and implementation:

- **Modular architecture.** VIMCO has a modular architecture, which allows for easy maintenance and modification of the VIMCO software.

- **Well-defined interfaces.** Components in the VIMCO architecture present well-defined, uniform programming interfaces that are easy to understand. This makes the development of a new component easy, reduces the possibility of error, and improves the interoperability and reusability of components.

- **Independent components.** VIMCO components are independent from each other. Independence is achieved through interfaces and driver components, where driver components implement the interfaces. As such, any driver component can be replaced by another component without disturbing the system, as long as the new component complies with the interface.

- **Scalability.** VIMCO architecture is scalable, flexible, and open to support addition of new users and resources in time.

As a result, VIMCO is an evolvable information management platform that implements the generic and uniform data/functionality models of the information management framework.