1 ATLAS Metadata Interface (AMI), a generic metadata framework

J Fulachier¹, J Odier¹, F Lambert¹, on behalf of the ATLAS Collaboration.

¹Laboratoire de Physique Subatomique et de Cosmologie, Université Grenoble-Alpes, CNRS/IN2P3, 53 rue des Martyrs, 38026, Grenoble Cedex, FRANCE

E-mail: jerome.fulachier@lpsc.in2p3.fr

Abstract. The ATLAS Metadata Interface (AMI) is a mature application of more than 15 years of existence. Mainly used by the ATLAS experiment at CERN, it consists of a very generic tool ecosystem for metadata aggregation and cataloguing. We briefly describe the architecture, the main services and the benefits of using AMI in big collaborations, especially for high energy physics. We focus on the recent improvements, for instance: the lightweight clients (Python, JavaScript, C++), the new smart task server system and the Web 2.0 AMI framework for simplifying the development of metadata-oriented web interfaces.

1. Introduction

The aim of this paper is to give an overview of the ATLAS Metadata Interface (AMI), a mature ecosystem dedicated to metadata. AMI basic architecture was maintained for over fifteen years, adapting both software and hardware infrastructures to take advantage of new data computing standards in order to keep ensuring a high quality of service. AMI makes it possible to aggregate distributed metadata and to retrieve data in modern and robust web applications. It is the basis of important tools which are part of the offline software of the ATLAS experiment [1] at the CERN Large Hadron Collider (LHC), for which it was initially developed. It is an official component of the ATLAS production system.

AMI [2-5] is mainly used by the collaboration for three different purposes. The first is to catalogue the datasets available for analysis, with the Dataset discovery tool. A second tool, Tag Collector, is a highly specialized application for the management of the various releases of the ATLAS software. A third tool, the AMI-Tags interface, is used to save the processing history of the datasets. The generic nature of AMI makes it possible to valorize it in other experiments. The framework comes with a set of tools that can be used to quickly design metadata-oriented applications. It provides facilities for aggregating and searching data with easy-to-use web interfaces and lightweight clients.
In the following sections the main components of AMI are described:

- The server, written in JAVA, provides a single HTTP endpoint for all clients (web applications, APIs), a command interpreter with a highly configurable authorization system, a single sign-on for applications supporting all standard authentication methods, a distributed transaction engine, connection pooling for data sources, a Metadata Query Language (MQL) to SQL generator and finally a task server, a component equivalent to a distributed cron that can perform metadata aggregation on heterogeneous datasources.

- The JavaScript web framework, which takes advantage of modern standards web technologies like HTML5, CSS3, jQuery and Twitter Bootstrap. It provides an application and sub-application Model-View-Controller (MVC) pattern, and comes with a customizable search engine modeler and advanced data displaying tools.

- AMI Clients: The main client provided by the AMI team is a Python client called pyAMI. As the command interface is very simple, using a single HTTP endpoint, it gives the possibility to easily develop new clients (C++, JAVA, …).

- Finally the benefit of deploying AMI in cloud environment, the technical choices and the consequences in terms of flexibility and scalability are discussed.

2. AMI server overview

This section describes the main characteristics of the core part of AMI. The core of the AMI server can be considered as a command executor over a high-level database engine, see figure 1. JAVA was chosen because it is well adapted for HTTP services and database handling. The server has a multi-tier architecture and provides a modular environment to implement specific commands with a highly configurable authorization system.

![AMI Server Diagram](image-url)

**Figure 1.** Overview of the AMI server.

The AMI ecosystem follows the Open-Closed Principle (OCP). Software entities should be opened for extension, but closed for modification. Software developments are object oriented, portable and well adapted for both web applications and database connectivity. Encapsulation limits the overall system complexity, and thus increases robustness, by allowing the developer to limit the interdependencies between software components. In order to ensure OCP within the AMI layered architecture, dependency injections were introduced, using some “inversion of control” design patterns, in particular the “bridge” or “plug-in” pattern for communication with external tools, especially with databases and version control engines. This pattern is an example of encapsulation; the plug-in architecture completely hides the internal methods and access is only allowed through the interface.
2.1. The HTTP service

The unique point of access for all AMI clients is the HTTP web service. It provides X509 authentication over HTTPS protocol. For production, an Apache front-end cluster is used together with a load balancing system to redirect authenticated clients to a pool of Tomcat [6] servers. The Apache cluster is in charge of authentication. It hosts the service certificate for the AMI end point and takes care of keeping up to date the certification authorities and the revocation lists.

Authenticated clients are redirected to a Tomcat node using the mod-proxy module, a dedicated plugin for Apache server that transmits client identification. This allows fine tuning of both server pools according to the specific load on the system. On each node, we use a Java servlet which is able to trigger HTTP POST and GET requests to execute server side actions using the other components of the AMI framework. The result of an internal AMI operation, which is XML-formatted by default, can be transformed using AMI eXtensible Stylesheet Language Transformations (XSLT) engine and a set of transformations. At present, the default transformation generates a JSON output, which fits very well to modern JavaScript frameworks and it is easily mapped in Python to native directory objects.

In the past, the AMI service used to have two endpoints: one was a standard servlet and the other was a SOAP web service. SOAP technology was interesting in terms of interoperability between middleware services, but with the expansion of Asynchronous JavaScript and XML (AJAX), JSON and restful HTTP services, the SOAP web service was dropped in favor of a single endpoint. A direct consequence of this is the reduction of the maintenance cost (one code, fewer external libraries, only standard HTTPS JAVA libraries). The main function of the AMI servlet is to extract the command name and the arguments and to pass them to the AMI command executor. Once the command is finished the result is sent in the HTTP response to the client.

2.2. Authentication

AMI supports standard authentication mechanisms. It can use HTTP connection but the default is HTTPS. Authentication can be made directly on an AMI Tomcat node or on a front web server which redirects client identification to AMI nodes. In both cases, the authentication is based on the user information stored in the AMI central user database. This user database is shared by all applications using an instance of AMI, providing a single sign-on feature. Even if login/password can be used, the best way to connect is using X509 certificate.

To restrict usage of AMI to the members of the ATLAS collaboration, the accounts validation is based on a GRID authorization service: VOMS [7]. To be able to have his AMI account valid, a user must register its grid certificate in VOMS to obtain some specific ATLAS roles. Some dedicated AMI task is responsible for the synchronization of AMI role database. For browser clients, the session is made with a cookie stored in the navigator and for the APIs, the session belongs to a cookie stored in a dedicated authentication file.

2.3. The command engine

After being authenticated, a client can execute the AMI command with the parameters passed from the front-end servlet. As described in the next section, the execution of a command is allowed if you pass the authorization algorithms. AMI command engine can be considered as the heart of the AMI framework. It represents the Model in the MVC pattern implemented in AMI software.

By reflection it instantiates the proper object for a given command using the AMI authorization layer and transaction engine to ensure atomicity of the command execution. The command either ends properly with an appropriate result, or raises an exception that is passed to the client. An AMI command has access to all general utilities provided by the AMI framework, as well as all specific tools implemented for a given experiment. As all interactions between a command and the data sources are constrained by the use of the AMI transaction engine, it confers atomic properties to the command. Even if low-level access to a database system can be performed, the general canvas pro-
vided by the command system ensures distributed transaction management. In addition, a command
can trigger the execution of other commands. It can be in the same transaction or not. This extends
atomicity to a tree of commands. This is especially used in commands dedicated to external data ag-
gregation and running in some specific server thread. All specific application functionalities can be
implemented by enhancing the parent command class or any child class providing general features that
can be extended for some specific operations.

The main generic commands provided are: i) administration commands: users, roles, data
sources, tasks, monitoring, … ii) search commands: SQL, Metadata Query Language (MQL),
browsers, details, … iii) data manipulation commands: add, update, clone, remove, …

For selection commands a high-level request language was implemented. MQL requests are
transformed to catalogue-specific SQL queries, using database schema reflection. The results are ag-
gregated to produce a single output. This mechanism can be used as an abstraction layer in case of
database schema evolution.

2.4. Authorization system
An important thing when one develops an application is to rely on a robust user/role/action mapping.
Often, the simpler the role handling system is, the less configurable it is. In AMI, the datasource cre-
dentials are decoupled from the AMI users. Each AMI user is mapped to AMI roles. AMI roles can be
specific to AMI applications or can be imported from other authorization databases, as is the case in
ATLAS for VOMS roles associated with users. Those roles are then mapped to a set of AMI com-
mmands.

For generic data manipulation commands (add, update, remove), it is possible to parametrize
some restriction on the command action field. Thus it is possible to limit the addition of a row in a
given table of a given database. This is handy but it is insufficient for granting users privileges for very
specific actions on distributed database and middleware. In order to extend the role mechanism to any
potential application need, a reflection mechanism was introduced to easily develop some specific test
algorithm executed when checking if a command can be launched. An advantage of having a central-
ized user database for AMI applications is that it provides a single sign-on and the roles can be shared
between applications.

2.5. The transaction engine
This component is essential because it ensures the atomicity of operations in the AMI framework. It is
a singleton with a pool of transaction objects shared by an instance of AMI. For each transaction iden-
tified by a key, a set of datasource connections are stored and the transactions of each external data-
base are synchronized by the AMI transaction object.

Each time a command belonging to a given transaction calls a database, it adds a connection in
the transaction or uses one already present. In this way it is possible to implement a hierarchy of com-
mmands in the same transaction: nothing is committed on databases before the AMI transaction is com-
mmitted. All calls to databases in the transaction are performed on the same database transaction and so
all changes are seen by the transaction actors.

2.6. Connection pooling
In database applications, connection pooling is very important, this is the way to minimize the number
of connections to databases and to speed up applications. The first version of AMI core layer had its
own connection pooling system, based on JDBC [8] connections. It was robust enough to support
database interactions for the last decade, but the new version of the framework uses an open source so-
lution to benefit from the active development effort of the Tomcat team [9]. In both cases, the connec-
tion pool stores active connections that are used by the AMI transaction engine, and some free connec-
tions, ready to be used. The connection pool configuration can be tuned to minimize opened connec-
tions and maximize availability for AMI commands using them through the transaction pool.
2.7. Task server

One of the main uses the ATLAS collaboration makes of AMI is for retrieving data for analysis. It keeps physics metadata associated with datasets (group of files) and stores configuration parameters for offline reprocessing of data. To load this information in AMI databases, information is extracted from many external sources. AMI framework provides a distributed task execution system. Basically it is a thread using AMI classes which reads information in a centralized database. For each node running this thread, some tasks are affected.

Each task can run an AMI instance with a specific operation, or any external program. With this task infrastructure it is possible to change the allocation of a given task. Each task has a priority, a step, and logs its execution in the database and in a log file. An exclusion mechanism is present in the task engine, therefore it is possible to construct complex task execution graphs. Some dedicated monitoring was set up to be able to act quickly if the metadata flow changes. With the recent usage of the cloud technology for production services, it is now possible to easily scale the number of nodes running task servers.

In its very latest version, the task server includes a lottery scheduling (random task selection) which solves the problem of starvation. This is not the case for priority round-robin-like scheduling algorithms.

3. AMI web framework

The first AMI web interfaces were written in PHP, but this was rapidly abandoned for a more structured approach. A Java class hierarchy for the generation of HTML pages via XSLT was written and this was used until 2013, when it was partially replaced by a structure based on JavaScript. This work was extended further in 2014. Web development was considerably simplified by the development of a framework for AMI based on jQuery [10] and Twitter Bootstrap [11]. This section is dedicated to this new AMI web framework.

3.1. Web framework capabilities

The AMI web framework was developed in JavaScript (JS) and jQuery. It separates the HTML (the view) and the application itself (the model). The web framework implements a Model-View-Controller (MVC) pattern. For this, a JavaScript version of TWIG [12] was implemented. It is a scripting language introduced in HTML fragments that makes it possible to dynamically generate HTML tags. In this way, HTML code is totally banned from JavaScript code. Each application relies on a core responsible for resource and event management (dynamic JavaScript and Style Sheets loading, HTML fragment loading, application loading/switching, common event dispatching, ...).

In addition to the core subsystem, there is an AMI client and another subsystem, based on the client, responsible for authentication with credentials or X509 certificates.

Commands are executed asynchronously using AJAX in order to increase both responsiveness and flexibility for applications. JSON [13] is used as data transfer format. An AMI application is a JavaScript class containing a minimal set of methods: onReady, onExit, onLogin, onLogout and onSessionExpired. As mentioned in the previous paragraph, applications can be loaded/switched dynamically. By default, they automatically benefit from the Twitter Bootstrap CSS framework. This framework contains HTML/CSS-based design templates for typography, forms, navigation and other interface components. Consequently, the AMI applications are resolutely Web 2.0 oriented.

3.2. Web framework components

The AMI web framework provides many components like hierarchical search engine, monitoring interface, configuration interface, basic Content Management System (CMS). The framework benefits from Web 2.0 functionalities. The technical complexity is hidden in some well-known libraries such as jQuery for JavaScript or Twitter Bootstrap for CSS and in the AMI web framework itself. The interfaces are based on reusable graphical components shared across applications.
Two families can be distinguished: i) the core components which are part of the framework and are available from anywhere; ii) the dedicated components which are application specific. An important core component is the "common login interface", see figure 2.

![AMI authentication](image)

**Figure 2.** The “common login interface” of the AMI web framework.

4. **Clients**
AMI comes with a powerful Python client called pyAMI, a JavaScript client, and an experimental C/C++ client. In this subsection, pyAMI is succinctly described.

pyAMI can run with both Python 2.6+ and Python 3.x. Based on HTTP/HTTPS, it allows credential and X509 certificate/proxy authentication. It is distributed as a Python library on the Python Package Index (PyPI) [14]. In the package, a Command-Line Interface (CLI) is also provided. It can easily be extended with experiment-specific features. All the information contained in AMI can be retrieved by pyAMI.

5. **Deployment on the cloud**
Using virtual machines guarantees uniformity and replacing a physical machine does not affect services. Furthermore, costs are lower for virtual machines than for physical machines, especially in a cloud environment. For these reasons, the AMI team has collaborated with the CC-IN2P3 in order to set up AMI as the first external service deployed on the cloud computing center.

![AMI deployed on the cloud](image)

**Figure 3.** AMI deployed on the cloud
5.1. OpenStack

The cloud technology available at CC-IN2P3 is based on OpenStack [15], a free and open-source cloud computing software platform. It makes it possible to manage large pools of computers, storage and networking resources in a data center. OpenStack has a modular architecture with various components. For example, NOVA is the cloud computing fabric controller. It is the main part of the Infrastructure as a Service (IaaS) model. It is designed to automatically manage a pool of resources and is able to work with most virtualization technologies. To interact with the cloud AMI uses the OpenStack REST API [16]. It gives full control to the AMI framework.

5.2. The AMI infrastructure

It was decided to migrate both development and production servers to the cloud, see figure 3. AMI currently uses a pool of twenty virtual machines that can be instantiated in a range of 30 fixed IP addresses. Each virtual machine is built from an image containing controlled versions of the operating system (SL6), of the required middleware and of AMI itself. This infrastructure makes it possible to test upgrades of systems as well as upgrades of applications in an automatic way. Those operations are controlled by the integration server which is itself in the cloud.

Currently, there are four Tomcat/AMI nodes with load balancing, three AMI task servers (used to fill databases), one Tomcat/Jenkins integration server and two development servers. In case of spike of important load on the AMI task servers, new nodes can be deployed on demand. AMI comes with an advanced monitoring service (CPU, memory, network), in order to better size server flavors (CPU, memory) and the number of server instances.

6. Outlook

The ATLAS Metadata Interface (AMI) is a mature application mainly used by the ATLAS experiment at CERN. It is a very generic tool ecosystem for metadata aggregation and cataloguing. Since 2013 another experiment, nEDM [17], is using AMI for run bookkeeping. Agreement was reached with the SuperNEMO [18] collaboration for the use of AMI, and at least one other experiment has expressed interest. We will work on making AMI much easier for experiments to set up databases with a minimum of support from the core team.

Acknowledgements

Over the years we have been helped and supported by many people in the ATLAS collaboration, and at the CC-IN2P3, in particular Philippe Cheynet, Benoît Delaunay, Noel Dawe, Markus Elsing, Pierre-Etienne Macchi, Yannick Perret, Emil Obreshkov, Mattieu Puel, David Quarrie and Jean-René Rouet.
References


[14] pyAMI sources: [https://pypi.python.org/pypi/pyAMI_core/](https://pypi.python.org/pypi/pyAMI_core/)


