The AMGA Metadata Service

B. Koblitz (birger.koblitz@cern.ch)
CERN, Switzerland

N. Santos (nuno.santos@cern.ch)
EPFL, Switzerland

V. Pose (vpose@cern.ch)
JINR, Russia

Abstract. We present the AMGA Metadata Catalogue, which was developed as part of the EGEE (Enabling Grids for EsciencE) project's gLite Grid Middleware. AMGA provides access to metadata for files stored on the Grid, as well as a simplified general access to relational data stored in database systems. Design and implementation of AMGA was done in close collaboration with the very diverse EGEE user community to make sure all functionality, performance and security requirements were met. In particular, AMGA targets the needs of the High Energy Physics community to rapidly access very large amounts of metadata, as well as the needs for security of the Biomedical community. AMGA therefore tightly integrates fine grained access control making use of a Virtual Organisation management system. In addition, it offers advanced federation and replication features to increase dependability, performance and data security.

Keywords: Data-Grids, Metadata, Replication

1. Introduction

Many advances in science today depend on the capability of researchers to share vast amounts of data and collaboratively use computing resources like networks, storage space or computer nodes. To give an example, in high-energy physics (HEP), the experiments at the Large Hadron Collider (LHC) will produce data in the order of tens of petabytes annually when the machine will be commissioned in 2007 [?]. Collaborations of thousands of researchers from institutes around the globe will need to access this data. Grid technologies [?] provide the means to efficiently share resources in such huge distributed environments.

As part of the Grid Middleware, scalable and fault-tolerant Metadata Services are essential for Data Grids, primarily as a means of describing and discovering data stored in files but also as a simplified, Grid-enabled, relational database service. In this paper we present the AMGA (ARDA[1] Metadata Grid Application) metadata catalogue,

[1] A Realisation of Distributed Analysis for LHC: http://lcg.web.cern.ch/lcg/activities/arda/arda.html

which was developed as part of the EGEE project's gLite Middleware. AMGA implements the EGEE metadata interface, which was defined in close collaboration with the different EGEE user communities and which has been adopted also by other Grid metadata services [?]. The main features of the AMGA catalogue are high performance, tight integration into the Virtual Organisation (VO) management system of the EGEE Grid, which works with X509 Grid certificates, fine-grained access control and advanced replication features. During the implementation of AMGA a continuous stream of feedback from early adopters assured that the performance needs of the users (in particular of the HEP community) were fulfilled while meeting the strict security needs of the biomedical community.

In this paper we will first present an overview of the requirements collected from the EGEE user community (Section 2), before describing the EGEE metadata interface (Section 3) and the AMGA implementation (Section 4). The emphasis is on the replication and distribution mechanisms of AMGA that were designed to provide the scalability and fault-tolerance required for operation in a Grid environment. Being part of the middleware, these mechanisms provide database independent replication, especially suited for heterogeneous Grids. We show how asynchronous replication is used for scalability on wide-area networks and for better fault tolerance. Our implementation supports updates on the primary copy, with replicas being read-only. For flexibility, AMGA supports partial replication and federation of independent catalogues.

Section 5 gives an overview of the performance characteristics of AMGA, which is one of the main requirements on a metadata catalogue. In Section 6 we describe several EGEE Grid applications that are currently using AMGA, and in Section 7 we discuss related work.

2. EGEE Requirements for a Metadata Catalogue

The work presented here is motivated by the use cases we have identified while working with the EGEE user community. This is a large and active community, as shown by the number of applications being ported to the EGEE grid Middleware. These applications vary widely in requirements, complexity, size and maturity, but it is possible to define general trends, and one of such trends is that Metadata Catalogues are essential for most of them. Here, we will describe the requirements of the two main EGEE user communities, HEP [?] and Biomed [?], which were the main motivation for our work. They are examples of two very different classes of use cases, with most of the use cases of other EGEE applications falling somewhere in between.

2.1. High-Energy Physics Community

HEP applications use a large number of files, in the order of hundreds of millions, with metadata associated to them. After production, the files and the metadata are usually read-only for users. The write rate is not expected to exceed a few entries per second [?]. The read rate is harder to predict, since it depends on the behaviour of thousands of physicists across many grid sites, but is expected to be one or two orders of magnitude larger than the write rate, with the possibility of usage spikes. Apart from file-related metadata HEP analysis often also needs additional structured relational data describing the calibration of the experiment or its condition. This application specific data is usually stored on relational databases that must be made available on the Grid.

HEP users are spread geographically across around two hundred sites, requiring special attention to deal with high-latency connections. Security is not a primary concern, as the metadata is not sensitive. Authentication is required to prevent denial of service attacks and for tracking users, but data is commonly sent as clear-text. For this class of applications, the main concerns are scalability, performance and fault-tolerance.

2.2. Biomed

Biomed applications manage a much smaller amount of metadata and have relatively low update and read rates. On the other hand, they have very strict requirements on the privacy and safety of data (e.g. medical images). The metadata itself may be even more sensitive, since it often includes patient names or specifics of their illnesses. This metadata is generated in different locations (hospitals or laboratories). Due to its sensitivity, it must be handled with extreme care.

The metadata service therefore should have fine-grained access restrictions based on Access Control Lists (ACLs) which allows to restrict e.g. reading patient data to only a few medical practitioners. While most other applications usually will be able to use only per-schema access permissions (which is easier to manage and has a much smaller performance impact), medical applications will need per-entry access permissions in a schema.

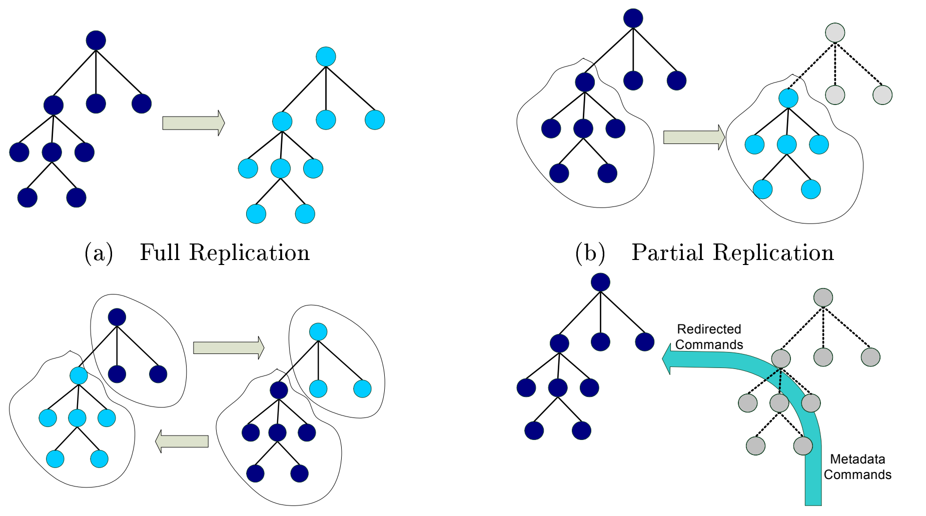
2.3. Requirement Analysis

The descriptions above suggest two classes of use cases: file metadata and application-specific structured data. The need for a catalogue for file metadata arises primarily from the vast number of files stored on a data grid, which users must be able to search for relevant files. A catalogue for structured relational application metadata becomes necessary on the Grid since direct access to database systems is not viable in such an environment, since traditional databases are not grid-aware, mainly in what concerns authentication and access control.

The two classes of applications are similar enough so that a single generic metadata catalogue can address then both. As basic requirements, the catalogue must support insertion, deletion, querying and updating of data records similar to the data manipulation in relational database systems. It should describe data in terms of a schema provided by users. And since no schema would serve the needs of all the application domains, the catalogue must support flexible and dynamic schemas, giving users the possibility of changing it at run-time. The service must also allow metadata to be structured as a hierarchy of logical collections, so that related metadata can be grouped together and isolated from other metadata. To deal with large number of entries (several millions), it must be designed with scalability in mind. The support for collections and hierarchies is a good first step in this direction, with replication being the next logical step.

Security is one of the major requirements, especially for the Biomed community. Authentication and access control are essential features. To integrate with the EGEE grid Middleware, the security of the catalogue should be based on X509 Grid certificates for authentication and on the Virtual Organisation Management System (VOMS)[**?**] for authorisation. Virtual Organisations (VO), i.e., groups of individuals and institutions collaborating on the Grid towards a common purpose, can set up membership or special roles for their members on their VOMS servers. Users within a VO ask the VOMS upon signing on to the Grid to create a short-term X509 certificate which contains a requested role or the group membership as additional attributes. The metadata service needs to extract such attributes from a certificate presented by a user upon login and use that information for authorisation purposes. Some communities like Biomed require fine-grained access control to their data, i.e. to individual items as opposed to full sets.

To cope with the heterogeneity of the Grid environment, a metadata catalogue service should support several different database back-ends, to better adapt to what is deployed on each grid site. The LHC computing grid (LCG) currently consists of about 200 participating sites and deployment of the middleware claims a large amount of the available manpower. Since many middleware services (e.g. the File Catalogue, batch system, file transfer system) anyway require database back-ends, it must be possible for a site to consolidate the number of different database systems deployed and use the one they have the most experi-



(a) Full Replication (b) Partial Replication

(c) Federation of Metadata Servers (d) Federation with Proxy Redirection

Figure 1. Replication/Distribution models.

ence with. On the other hand, the applications targeted by LCG have very different requirements on the availability and scalability of the metadata service, which may require only a file-based database linked into the service but might also demand a load-balanced multi-server installation of Oracle.

2.4. Replication Requirements

Replication is required by the HEP community to provide the scalability and fault-tolerance required to support over 200 geographically distributed sites. Still in the case of HEP applications, writes can easily be performed in one or a few central catalogues, but reads are more frequent and must be offloaded to read-only replicas that are closer to the users. Partial replication is also an important feature, as replicating only the data needed by local users can often result in an order of magnitude decrease in the amount of replicated data.

For the Biomed community replicating the data either to a central catalogue or to other replicas would increase the exposure to attacks. A better solution is the federation of individual catalogues into a single virtual catalogue, allowing data to remain secure at its site of origin, while providing transparent access to authorised users regardless of their location. While users see a global storage of metadata, access to the metadata stored on individual sites can be regulated by that site according to local privacy laws.

Figure 1 describes the main usage scenarios we have identified. *Full and partial replication* correspond to the HEP application case, where data is generated centrally and replicated for fault-tolerance and scalability. Partial replication is necessary for situations where remote users are only interested in part of the Metadata. This scenario can be implemented either by replicating part of the directory hierarchy, or by using a filter to specify an arbitrary subset of the data. The former requires a hierarchical structure of metadata schemas to replicate only the sub-trees required at the slave. Filtering is more generic and flexible, allowing the slave to specify arbitrary conditions that will be used by the master to select the logs shipped to the slave. In fact, partial replication of sub-trees is a special case of filtering, where the filter matches only a sub-tree.

Federation corresponds to the Biomed use case, where data generated in different Grid sites is federated as a single distributed catalogue consisting of several physical catalogues. In this case, the remote nodes can be either physical or virtual replicas. In the first case the data is copied to the replica, while in the second no data is copied; instead the metadata commands executed on the slave are redirected to the master. Federation is described in more detail later in this paper together with the replication architecture of AMGA.

3. The gLite Metadata Interface

In this section we provide an overview of the interface. The detailed description can be found in [?].

Metadata is defined within the gLite metadata interface in terms of schemas, entries and attributes. *Entry* are the most basic elements managed by the catalogue, representing the real world entities described in the catalogue. *Schemas are* collections of entries and may contain other schemas. The AMGA implementation organises schemas hierarchically, like directories in a file system, with entries playing the role of files. A schema has a list of *attributes*, which have a storage type and a unique name within each schema. Entries assign to each of the attributes of their schema a value, which can also be NULL. In a typical implementation of the interface, on the database back-end, schemas will correspond to tables, attributes to columns of these tables, and entries to rows.

The view taken by the gLite Interface is that a Metadata Service should be generic, in order to allow its use in as many applications as possible. Therefore, it makes no assumptions on the classes of entries that are to be stored on the metadata service, and does not provide any predefined schemas. Instead, it provides the building blocks that allow users to define their own classes of objects: entries can represent any object with an unique name, and the schemas can be defined freely on a collection-basis. This is in contrast with other types of Grid Catalogues that are designed to store specific classes of entities, e.g., File Catalogues, and therefore have hard-coded support for those classes, including predefined schemas and specific operations for those types. This makes them easier to use, but limits them to the particular application domain they were designed to. In some cases, this specialisation is necessary due to the complexity of the task, like with File Catalogues. But for other types of applications, which are the target of the gLite Interface, a generic catalogue is enough.

The interface specifies operations to add and delete entries, and to set or update attributes of entries. To support dynamic schemas, there are operations to add and remove attributes from schemas and to list the attributes of schemas. In addition, it is possible to restrict updates so that they apply only to entries whose attributes fulfil a condition specified by the user. Values of attributes can be retrieved through the interface, which also allows to restrict retrieval to entries fulfilling a given condition. Most of the operations can be used in bulk mode, so that they work on several entries at once to reduce the number of network round-trips. In particular, entries can be registered in bulk (even across different directories) as well as updated or deleted in bulk based on a given condition. Transactions spanning different operations are also supported by creating transactions on the database backend. Both transactions and bulk operations are limited in the time they take to complete (by default 20 minutes), because they are bound to a server process and require a persistent TCP connection to the server. To make the interface more intuitive, patterns for entry names are supported, which is particularly important when used for file metadata.

For the retrieval of large datasets, the gLite metadata interface uses iterators, allowing users to retrieve the answer in chunks. One way of implementing iterators is by keeping the iterator state at the server. However, while a stateful implementation will typically provide better performance, it is also much more prone to resource exhaustion at the server if clients are not well behaved, making it harder to implement safely. Acknowledging this, the interface allows either stateless or stateful implementations of the service. A stateless implementation will need to perform the query for each iteration and limit its results based on the current position of the client, which is an optional argument to the method of the interface used for iterating.

The interface defines an SQL-like query language including string and mathematical functions which an implementation must translate to the respective SQL or XPath query on the backend. An implementation must therefore parse any query and validate it to prevent access control violations.

A crucial problem for scientific applications is the handling of the different data types of attributes. The interface must guarantee that data can be stored transparently on the backend and retrieved again as the same value. Several different data types on the backend must be supported, as this is required to be able to execute queries on that back end using data type aware functions. The interface therefore requires implementations to be able to store the following data types on the backend: *floats, integers, strings, timestamps* and *numeric* (a fixed point DB type). The interface itself is unaware of the actual data type of a value and knows only string representations thereof, but guarantees that applications will retrieve the same value they stored, within the limits of the data type's precision. The interface design leaves therefore the problem of type conversion to the client application (and the server implementation).

The metadata interface was designed to be modular, allowing a service to implement only parts of it. As an example AMGA running alongside the LFC file catalogue [?] does not need to implement the part of the interface which handles entries, as the file catalogue already has this capability.

4. Implementation of AMGA

AMGA follows closely the gLite Interface, providing all the features specified in Section 3. But the Interface leaves some freedom to the implementations in some aspects, which we clarify in this Section.

AMGA is implemented as a multi-process C++ server with a relational database back-end. It has two different front-ends, a web service front-end implemented using gSOAP, and a text-protocol based front-end, which is able to stream data back to the client. We will give here only a brief overview of the implementation of AMGA focusing on the features relevant for its performance or replication capabilities. A more detailed description can be found in [?].

AMGA is a service in front of a relational database system used as a storage back-end. Currently Oracle, PostgreSQL, MySQL and SQLite are supported. Requests received by either of the two front-ends are forwarded to the back-end, which streams data back into a buffer shared with the front-end. This data is returned to the user by the text-based front-end as a text stream whereas the front-end using SOAP provides the user with an iterator mechanism to browse the result. The text-based protocol significantly outperforms the SOAP protocol [?], but only the SOAP front-end provides a fully compliant implementation of the interface as defined in [?]. Both access protocols support bulk operations where several entries are inserted or read in bulk. In addition, AMGA also provides support for user controlled transactions spanning multiple operations.

Schemas are implemented hierarchically in AMGA. This model has the advantage of being natural to users as it resembles a file system, and of providing good scalability as metadata can be organised in sub-trees that can be queried independently. In addition this implementation choice permits replication of only parts of the schema hierarchy or to setup federated schemas where remote (sub-trees) of schemas are mounted into the local schema hierarchy. Schemas are dynamic and defined on a per-collection basis. In addition, child directories may optionally inherit the schema of the parent directory.

Considerable effort has been spent on providing native client APIs for C, C++, Python, Perl and Java, both for the SOAP and the text protocol. Interestingly, the text protocol implementations using standard socket techniques were considerably easier to implement due to many compatibility problems of the SOAP toolkits. A command line client is also provided, which is used by many applications.

Caching SSL security contexts in sessions on the server is very important for the server's scalability, as the public-private key cryptography to establish SSL connection contexts is very CPU intensive and limits the number of connections that can be established severely on custom hardware [?, ?]. AMGA caches the SSL contexts either in shared memory or in a file based database. The command line client caches the session in a file between calls, which speeds up the operations by about a factor of 10.

AMGA itself has few interaction points with other gLite middleware, because it is supposed to be used flexibly for various tasks involving relational data in the user applications. However, AMGA makes use of the gLite service lookup service, allowing AMGA clients to conveniently find the closest or otherwise most suitable AMGA server. Of course AMGA is also tightly integrated into the VO management system by making use of the VO specific information encoded in X509 certificates used to access AMGA as is being described in more detail in the following.

4.1. Security

AMGA features very flexible authentication and authorisation methods, based on passwords, X509 (Grid-)certificates and VOMS enabled certificates. Connections can be encrypted via SSL but this can be switched off for performance reasons. AMGA also supports user management internally if necessary. This allows simple deployment for VOs that only need a few Grid services and therefore may not need or want to manage resources centrally through a VOMS.

Access control is supported either on a per schema or per entry basis. The default is per schema access control, where all entries in a schema share the same ACL (Access Control List). Per entry access control is also supported, at the cost of some performance degradation. The implementation also supports groups of users, which can be defined within AMGA itself or via VOMS roles. ACLs are inherited from the parent directory when new entries are created, similar to sticky bits in a Unix filesystem.

Column-wise (i.e. per attribute) access restrictions are supported by views in AMGA, an approach found commonly in database systems. AMGA allows users to create views combining selected attributes from different collections into new collections. These collections then can have different access rights as the primary ones. While this solution is not straightforward for the users, it is the only solution with good performance we are aware of, since column-level access control would impose an unacceptable overhead.

4.2. Replication

This section provides a brief overview of the replication mechanisms in AMGA. A complete description can be found in [?].

Replication allows a catalogue (slave) to create a replica of part or of the totality of the metadata contained in another catalogue (master). The system ensures that the replica is kept up-to-date, by receiving updates from the master. The clients of the slave catalogue can then access the replicated metadata locally. This has many benefits in terms of improved scalability and fault-tolerance.

AMGA uses an asynchronous, master-slave model for replication. The use of asynchronous replication is motivated mainly by the high latency of Wide-Area Networks, where synchronous replication does not scale properly [?]. Master-slave replication was chosen because it covers the needs of the majority of our target applications, all of them having simple write patterns. The master-slave model works well with applications where writes are infrequent or else originate from the same geographical location. Multi-master replication would significantly increase the complexity of the system by requiring inconsistent updates to be detected and resolved, and would not provide any significant benefit to the target applications.

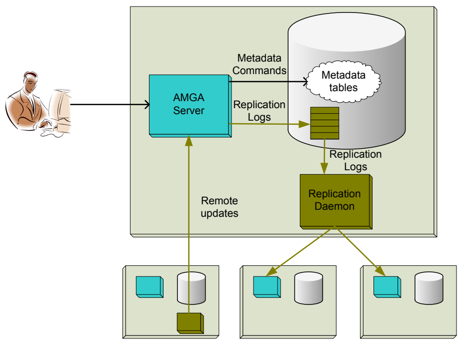


Figure 2. Replication internal architecture.

Figure 2 presents the replication architecture of AMGA. The basic mechanism is to have the master keep on its local database a log of all the updates it performed on its back-end. For each update, the master saves the metadata command that originated the update plus some contextual information required to replay the log on the slave. At a later time, these updates are shipped to slaves that replay the command locally. Since the metadata commands are independent of the database back-end, replication works even between AMGA servers using back-ends from different vendors.

The AMGA Server is only responsible for saving the updates into the replication log. The remainder of the functionality is implemented on the replication module, which is an independent daemon that can run on a different machine for better performance.

4.2.1. *Managing Subscribers*

To replicate a directory, a node must *subscribe* to that directory with the node that owns it. This informs the master that it should save the updates performed on that directory to its replication log, so they can later be sent to the subscribers. Subscriptions are hierarchical, i.e. they always include the metadata sub-tree rooted on the subscribed directory. They are also persistent, meaning they outlive crashes of the master and of the slave. If a slave disconnects without having first requested to be unsubscribed, the master continues saving the updates for the subscribed directories. When the slave reconnects, the subscription is resumed from the point it was interrupted. If the slave is disconnected for a long time, the master will eventually discard the subscription when the amount of pending logs exceeds a certain threshold. If the master fails, the slave tries to reestablish the connection automatically, while continuing to serve its local clients.

4.2.2. *Generating logs*

Nodes with at least one subscriber have to save to the replication log all the updates performed to directories subscribed by some other node. Each log entry is numbered with a unique sequence number - its ID - and contains all the contextual information required to execute the update on subscribers. To ensure consistency between the log table and the metadata tables, the replication log is written during the same database transaction used to perform the update.

4.2.3. *Sending the Initial Snapshot*

After subscribing to a set of directories, the slave must obtain an initial snapshot of their contents. To remain database independent, we cannot use the dump mechanisms available on most databases, since they are database specific. Instead, we have implemented a similar dump mechanism in AMGA that dumps the contents of a sub-tree of the metadata hierarchy as a sequence of metadata commands.

Sending a snapshot is a lengthy process, during which the master may receive updates from its clients. This can result in inconsistencies in the slave if these updates change the directories being copied. To prevent this situation, the reading of the snapshot from the database is isolated from any incoming update using a database transaction. These updates will be saved to the replication log and shipped to the slave when it finishes receiving the snapshot.

4.2.4. *Shipping Updates*

After obtaining the initial snapshot, the slave starts receiving and applying updates. To do so, it connects to the master using a TCP connection, sends the ID of the first update it needs, and waits for updates to be sent by the master. At the master, the replication module is responsible for shipping updates. It keeps track of all subscribers that are currently connected and of the ID of the last update they have acknowledged. Periodically, it polls the replication log table and sends any new updates to the subscribers interested on them. The replication module also deletes the updates from the log table when they are no longer needed by any subscriber. When all subscribers are connected, updates are deleted shortly after being generated. Only when a subscriber is off-line will an update be kept for a longer time.

4.2.5. *Federation of Catalogues*

The mechanisms described above support partial and full replication and provide the basis for federation. The federation mechanisms on AMGA are built on the hierarchical structure of metadata and on the support for partial replication. Mastership is granted not to a catalogue as a whole, but only to sub-trees of the catalogue. Therefore, node A can be the master for directory /a, which is replicated by node B, while node B can be the master for directory /b, which can also be replicated by A. The distributed catalogue contains both /a provided by A, and /b provided by B, corresponding to the situation depicted in Figure 1(c). This scheme permits different catalogues to have mastership of non-overlapping partitions of the metadata hierarchy, ensuring that for each directory there is a well known master.

Distributed queries are not supported, i.e., queries can only execute on a single node. However, it is possible to workaround this limitation in some cases by replicating all the directories needed by a certain query into a single node, where the query is then executed.

5. Server Analysis

Performance and resilience against failures are important attributes of a metadata service on the Grid. AMGA has undergone extensive performance studies and the replication capabilities are currently studied with respect to the recovery from various failure modes of a distributed system. In the following we present some of the results obtained, to give an overview of the characteristics of the AMGA metadata service.

5.1. Server Analysis

An important requirement of the HEP community is that a metadata service must not have a large performance overhead with respect to direct database access. We have therefore benchmarked AMGA and compared it with direct access to the same back-end database using JDBC. JDBC is a Java API for Databases and uses the native connection protocol of the respective database. In our test, we used JDBC to directly retrieve data from the database tables in which AMGA stores its data. During the tests no user-authentication was performed as in the HEP use-case where read-access should be unauthenticated and unencrypted. However the connection from AMGA to the database and from the JDBC client to the database was authenticated.

The test was performed on a LAN with about 2 ms round trip time and a bandwidth of 11.4 MB/s as measured with the *netcat* tool. Both server and client were dual Xeon machines with 1GB and 0.5 GB RAM respectively. The server ran the AMGA service as well as a PostgreSQL back-end database. The database was filled with 10 million entries with 5 attributes corresponding to an average payload size of 40 bytes. The entries were stored in 100 different schemas. Up to 100 concurrent

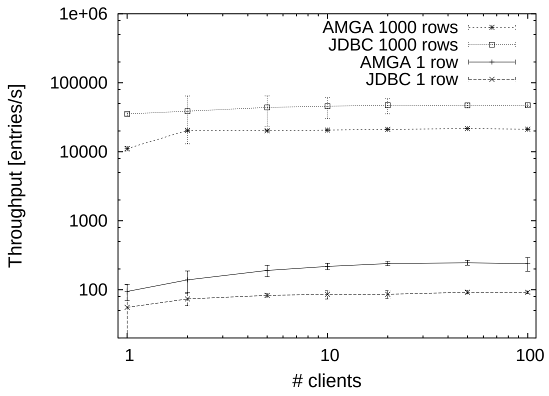


Figure 3. Comparison of the access speed to metadata stored in a PostgreSQL database through AMGA and directly through a JDBC client. Shown are the numbers of entries retrieved per second for single rows and 1000 rows at a time for both access methods depending on the number of concurrent clients.

clients were simulated on the client node while assuring that it is not a performance bottleneck for the test. The better performing streaming interface of AMGA was used during the tests.

Figure 3 shows the number of entries (= DB rows) retrieved through AMGA and JDBC for single entry requests and requests of 1000 rows each against the number of concurrent clients. For every request the clients open a new connection to AMGA or to the Database. While the overhead of processing the retrieved data makes AMGA about half as fast compared to direct access through JDBC for the requests returning 1000 entries, access through AMGA is actually faster when returning a single entry. Here, the access through AMGA benefits from the fact that the service has a pool of processes already connected to the database waiting for client connections, while JDBC establishes new connections for every query. This outweighs the additional database operations (likely cached by PostgreSQL) AMGA has to perform for access control checks. Our tests show the very good scalability and excellent performance of AMGA comparable to native database access.

A similar benchmark has been recently performed for the OGSA-DAI access layers for databases on a Grid (see Section 7 on related work) for a single client in [**?**], which showed an overhead of factors 7 to 40 over direct JDBC database access. Results of several other benchmarks of AMGA including comparison with the SOAP front-end, wide-area network access, and the impact of secure connections and authentication are given in [**?**].

5.2. Replication Performance

The replication in AMGA requires the master nodes to do a significant amount of work in addition to the normal operations of an AMGA server. This includes updating the replication log, managing subscribers and shipping the logs to several slaves. In contrast, the overhead on the slaves compared to stand-alone operation is minimal or even nonexistent, since receiving updates from a master requires the same computational resources as if they were being performed by a local client. Therefore, the master nodes are the potential bottleneck of the system.

We present here the results of a benchmark study assessing the scalability of AMGA replication at master nodes. The benchmark was performed on a 10 Mbit LAN. In spite of its low speed, the LAN was not the bottleneck as the data rate between the masters and each slave was about 10 KB/s[2]. The master AMGA server plus the associated replication daemon were running on the same computer, a P4 3 GHz with 1 GB of RAM. The slaves were running on two different computers, up to five in each, and were patched to discard incoming logs. This was done so that several slaves could be run in the same computer without becoming the bottleneck. The tests were performed with the slaves already connected to the master and waiting for logs. We then inserted 10,000 entries on the master at a rate of 90 per second, which corresponds to around 80% of the maximum rate when used stand-alone. We measured the overall rate at which updates were sent by the master to the slaves, and the CPU load at the master. Figure 4 shows the results.

The data point for 0 slaves corresponds to having the master save replication logs for slaves that are subscribed but disconnected. To provide a baseline, we measured the CPU usage with the AMGA server running stand-alone, i.e. no subscribers and not saving logs, which was around 20%. As can be seen, the scalability is close to linear with only a modest increase in CPU usage on the master.

5.3. Resilience of Replication

The replication mechanisms of AMGA are prepared to deal with faults resulting in the interruption of the connection between the slave and the master. If a master fails, the slaves attempt to reconnect automatically. If the slave fails, the master keeps the updates for the slave until it reconnects.

[2] Each update shipped to the slaves was around 100 bytes. For a rate of 100 updates a second, the network bandwidth required for each slave is approximately 10 KB/s.

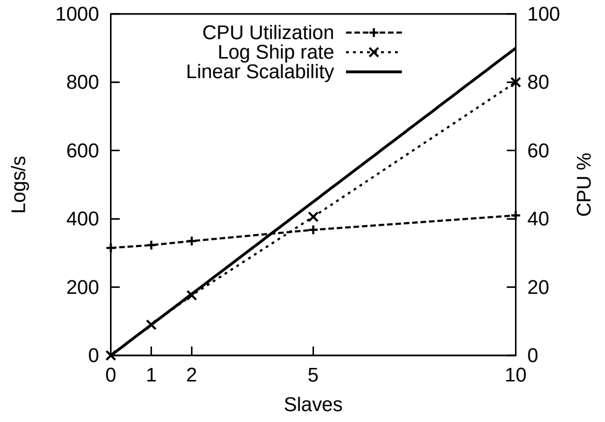


Figure 4. Scalability of a master serving up to 10 slaves. Insertion rate at master of 90 entries per second.

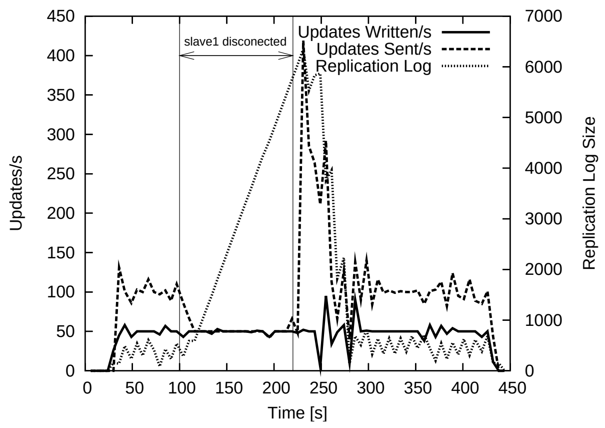


Figure 5. Behaviour of replication in AMGA measured in a master node while tolerating and recovering from a slave failure. The test starts with two slaves connected and receiving logs. At $t = 20$ s (20 seconds into the test), a client starts inserting 20.000 entries into the master at a rate of 50 per second. A slave disconnects at around $t = 100$ s and reconnects 120 seconds later.

Figure 5 shows how AMGA behaves when a slave fails. The tests were performed using the same hardware setup as the one used for the scalability tests described above. There were only two slaves running on separate machines. The slaves were unmodified, i.e. they were applying the received updates to their back ends. During the tests we monitored the performance of the slave nodes and verified that they were not the bottleneck.

From $t = 20$ s to $t = 100$s the system is in a steady state, with both slaves connected and receiving updates. The size of the replication log oscillates between 10 and 50. The spikes correspond to the intervals

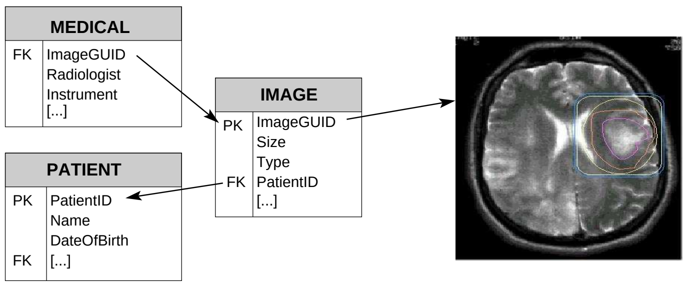


Figure 6. Table schema of the MDM biomedical application using AMGA.

between the periodic garbage collections of logs. The replication log is not smaller because the master can only delete an update from the log if it has received from all the slaves the acknowledgments confirming that the update was committed to the slave's back-end. This process takes some time, during which the updates must be kept on the log.

At $t = 100\,\mathrm{s}$ one of the slaves disconnects. Immediately, the size of the replication log starts increasing, as the master is keeping the updates for the slave that is disconnected. At $t = 220\,\mathrm{s}$ the slave reconnects, causing a large spike in the send rate, as the master starts sending all the pending updates to this slave. Recovery takes 60 seconds; at $t = 280\,\mathrm{s}$ the send rate and the size of the replication log has once again stabilised at levels similar to those before the disconnection of the slave.

6. Experiences

Several applications from different user communities are currently either evaluating AMGA or using it in a production environment. Close contact between the developers and these user communities has been very important to improve the design of the metadata interface and the AMGA implementation.

6.1. Medical Data Management

The MDM (Medical Data Manager) application [?] is a biomedical service that stores medical images on the grid and allows users to retrieve them again for post-processing or viewing. The service makes use of the fact that medical image data is highly standardised and that most medical imagers in hospitals, like X-ray machines, NMR scanners or computer tomographs all use the DICOM standard to store their files. Hospitals are often already equipped with central storage systems for this data. However, due to the high sensitivity of the personal metadata of these files and the lack of a global infrastructure to establish trust, these image files are not commonly shared between practitioners. The MDM service targets this need. A (simplified) schema of the table structure is given in Figure 6. Additional tables contain specific information on the DICOM description of the image and information about series of images as from a tomography unit. Finally, a logging table exists. The schema can be extended by the user for other image protocols.

The following is an example query actually performed by some of the used-cases:

```
selectattr /PATIENT:id '/PATIENT:name = family_name and
/PATIENT:dob = date_of_birth'
```

This query selects a patient's id based on his family name and the date of birth.

Access restrictions are very important to the MDM application because the stored data is highly confidential and the handling needs to meet the privacy requirements of all countries where this application is deployed. MDM therefore makes use of the fine-grained access restrictions of AMGA for the tables. The current scheme only uses per-table access restrictions so that e.g. doctors are allowed to modify the medical information while nurses can only read that table. There are currently no per entry access restrictions being used.

Since the MDM application also makes use of the storage middleware to store the images in an encrypted form (the keys are stored in a separate key-storage, which stores them in a distributed fashion) and the computing elements for post-processing of the images, the permission handling for the middleware components used needs to be tightly integrated. All the permission management is therefore centralised within the VOMS and AMGA only implements the access control as taken from the VOMS enabled certificate presented by the user at login.

The MDM application is still being developed. In the future MDM will make use of ACLs on a per-entry basis such that only medical staff which is concerned with a certain patient will be able to see all the related information. In addition, the use of AMGA's capabilities to create views are foreseen. This will allow to selectively restrict access for different groups to parts of an entry's metadata (column access restrictions) without hurting performance as these restrictions will be per-view. Finally AMGA's support for federating the metadata shall be used to keep metadata and access restrictions physically in the hospitals where they were created.

6.2. LHCb Bookkeeping

The LHCb experiment is one of the four HEP experiments at the LHC accelerator at CERN. Their bookkeeping service keeps count of the processing of the huge amount of data taken by the experiment and stored in files (their AMGA service contains currently 15 million entries, equivalent to a full year of data) on the Grid. This is a typical use case where file-related metadata is stored in AMGA. Per entry about 1KB of data is stored, currently amounting to 15GB of relational data in a complex schema documenting the characteristics of the data taken initially, the processing parameters and details of the processing process like the site and the time.

The Logging and Bookkeeping system system is for LHCb the central data inventory, that contains information about file contents and their processing steps. Production jobs store into this central system the information on what processing has been done with a file and user analysis jobs look up information about files based on e.g the physics process the user is interested in or the time of data taking. The Logging and Bookkeeping system can then provide the user job with the logical filenames of the files of interest as well as additional information like the data taking conditions.

LHCb bookkeeping has put strong requirements on AMGA in particular due to the need to be able to retrieve very large result-sets (millions of entries) without straining the AMGA server, because hundreds of concurrent users are expected to query the service to select data files for their analyses. This requirement was taken up by implementing streaming and iterators, and the intensive stress-testing has lead to identifying and removing many bottle-necks and instabilities. The bookkeeping team uses an Oracle database as a backend and is currently performing tests with over 100 million simulated entries to study the scalability of the setup. In the future replication could be used to further increase the scalability should the need arise.

6.3. UNOSAT

The UNOSAT project[3] is a United Nations program that provides satellite images to help aid workers in the event of natural disasters. Users can access and, if necessary, process images on the Grid by specifying by geographical coordinates and image type. This image metadata is stored within AMGA which holds also the storage location of the images on the Grid. This use case is similar to the MDM use case apart from the fact that the access restrictions are much less detailed because

[3] `http://unosat.web.cern.ch`

authorised users always have access to all the images. On the other hand this use-case is special because AMGA allows the users to use the GIS (Geographical Information System) features of the backend database, which provides essential functionality like spatial indexing of the data to provide fast access.

6.4. GANGA

GANGA [?] is a user job monitoring and management application developed by the LHCb and ATLAS HEP experiments. The GANGA application uses AMGA to store the status information of jobs running on the Grid which are controlled by GANGA. Objects representing Jobs by their metadata are in fact stored centrally on the AMGA database, which allows to migrate the monitoring of the jobs from one user client to another user client. AMGA's simple relational database features are mainly used to ensure consistency when several GANGA clients of the same user are accessing the stored information remotely. The job metadata stored includes information about the Job's status, its input and output data as well as inter-job relations when jobs depend on each other.

7. Related Work

The importance of a metadata service middleware for the usability of large scale local or wide area storage infrastructures has induced many groups to investigate metadata access for the Grid and to implement such metadata services.

Perhaps the earliest metadata middleware is the Metadata Catalogue Service (MCAT), which is part of the Storage Resource Broker (SRB) [?] developed by the San Diego Supercomputing Centre. The project's goal is to provide an abstraction layer over heterogeneous storage devices and file systems at or even across computing centres. The metadata catalogue component of the system stores metadata on the resources managed by SRB including all file-related metadata. The file metadata consists of system metadata (like access permissions) and user defined metadata. Like the AMGA implementation of the gLite metadata interface, MCAT is hierarchically organised using a tree of collections. More recently the MCAT was extended with support for federation and replication mechanisms [?]. While MCAT has functionality quite similar to AMGA, it is tightly integrated into the SRB system, which limits its usability by other Grid Middlewares. The AMGA catalogue, on the other hand is designed to be a component in a modular architecture.

The Global Grid Forum is currently working on a standardisation of a generic database interface in the Database Access and Integration Services (DAIS) working group. The interface is part of the Open Grid Services Architecture (OGSA) framework [?]. In contrast to the gLite metadata interface, which tries to hide the backing DBMS, the DAIS specification exposes the underlying data storage to the user. While it gives users access to all the features of the respective DBMS, it requires the user also to cope with the different implementations. Consequently, no simplified query language is defined, instead the user discovers the specific backend, and then uses for example the respective SQL-dialect of this backend or XPath for an XML database. We believe that the gLite metadata interface is therefore better suited for a heterogeneous environment as is the EGEE computing Grid.

The OGSA-DAI project aims to implement the DAIS specification as part of the Globus Alliance's middleware [?].

The Metadata Catalogue Service MCS [?], which is being developed by the Globus Alliance, is based on the same paradigms as the interface presented here, as it does also not expose the underlying storage backend. Initially implemented as a standalone service using Apache, it is now being developed on top of OGSA-DAI [?]. MCS provides like the gLite interface a hierarchical organisation of the metadata and flexible schemas. While a combination of OGSA-DAI and MCS would provide much of the functionality that the AMGA catalogue provides, we believe that the gLite interface is currently easier to use and that it provides the user with a more homogeneous interface to general hierarchically organised metadata.

In High Energy Physics, several LHC experiments have implemented metadata catalogues specific to their needs and consist of a standard RDBMS backend and an adapter layer to allow access in a distributed computing environment. These catalogues were studied in detail during the design phase of the gLite interface so that it encompasses the functionality needed by the HEP experiments. The catalogues are the ATLAS Metadata Interface AMI [?], which is now implementing the gLite metadata interface, the CMS experiment's RefDB [?] and the Alien Metadata catalogue [?] from the Alice experiments. Benchmarks of these implementations have raised questions about their scalability, which were addressed in AMGA.

All major database systems have some kind of replication mechanism, like Oracle Streams or Slony-I for PostgreSQL, but they are all vendor specific and therefore do not address the heterogeneity of a Grid. The motivation for generating and shipping replication updates came from these systems. However, instead of replicating at the database level, we replicate metadata commands, which provides database independence.

The FroNtier [?] project aims to improve the performance of database read access over the Internet, by wrapping database queries in HTTP requests and using Internet caching mechanisms. The LCG's 3D project [?] is setting up a distributed database infrastructure for LHC experiments, using Oracle Streams as the main replication technology. Both Frontier and the 3D project are aimed at generic database applications while we are focusing on replication of Metadata Catalogues, which allows us to be database independent by moving the replication functionality from the database to the Metadata Catalogue.

8. Conclusions

We have presented the AMGA metadata service, which is being developed as part of the EGEE gLite middleware. AMGA is being used as a core middleware component by a number of very diverse EGEE applications. The spectrum ranges from bookkeeping applications for high energy physics data with very high demands on the performance of handling requests and returning large amounts of data, to medical applications with very high privacy requirements resulting in very fine-grained access permissions. We have shown in a comparison benchmark, that the access speed of AMGA is comparable to the direct access to a database.

An important feature of AMGA, and one that is to our knowledge unique for a metadata catalogue for Grids, is its replication capability. We believe that replication and the federation of metadata provide a high level of resilience against failures and performance of distributed metadata systems that will allow many new applications to be used on computing Grids. We have given the security needs of medical applications as an example for the application of federated metadata. Although the replication capabilities are still in an early stage of development, our benchmarks and stability tests have shown very promising behaviour. It is in this direction, therefore, that we will focus our future research.

An important aspect in the design and implementation of AMGA has been to closely collaborate with early users to provide the features necessary to fulfil the applications' needs. In several cases additional requirements on AMGA arose during the application development and wherever possible the needed features have been added in a manner, which should be of general use also for other applications. We believe that this co-evolution of AMGA and the diverse EGEE applications has lead to a metadata catalogue that will be useful for the large majority of Grid-applications making use of metadata.

Acknowledgements

AMGA was designed and developed within the LCG-ARDA project and the authors would like to thank the project team for the fruitful collaboration. Many people have helped in the design of the EGEE metadata interface, discussed use cases and collected requirements, among them are the EGEE project's technical forum, the Biomed working group, the GridPP Metadata working group, and in particular the gLite data management team. Several early users have tested AMGA and given invaluable feedback, they include the LHCb bookkeeping team, the GANGA project, and the gLibrary project. This work was funded by grant SFRH/BD/17276/2004 of the Portuguese Foundation for Science and Technology (FCT), Bundesministerium für Bildung und Forschung, Berlin, Germany and by the European Commission under Contract number INFSO-RI-508833.

References

1. Albrand, S. and J. Fulachier: 2004, 'ATLAS metadata interfaces (AMI) and ATLAS metadata catalogs'. In: *Computing in High Energy and Nuclear Physics (CHEP)*.
2. Alfieri, R., R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohner, A. Gianoli, K. Lorentey, and F. Spataro: 2003, 'VOMS, an Authorization System for Virtual Organizations'. In: *1st European Across Grids Conference*.
3. Antonioletti, M., M. Atkinson, R. Baxter, A. Borley, N. P. C. Hong, P. Dantressangle, A. C. Hume, M. Jackson, A. Krause, S. Laws, M. Parsons, N. P. Paton, J. M. Schopf, P. W. T. Sugden, and D. Vyvyan: September 2005a, 'OSA-DAI Status and Benchmarks'. In: *Proceedings of the UK e-Science All Hands Meeting 2005*.
4. Antonioletti, M., M. Atkinson, R. Baxter, A. B. N. P. C. Hong, B. Collins, N. Hardman, A. C. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N. W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead: 2005b, 'Design and implementation of Grid database services in OGSA-DAI'. *Concurrency and Computation: Practice and Experience* **17**, 357–376.
5. Apostolopoulos, Peris, and Saha: 1999, 'Transport Layer Security: How Much Does it Really Cost?'. In: *INFOCOM: The Conference on Computer Communications, joint conference of the IEEE Computer and Communications Societies*.
6. Barrass, T. and *et al.*: 2005, 'Unlucky for Some - The thirteen Core Use Cases of HEP Metadata'. preprint GLAS-PPE/2005-03, University of Glasgow.
7. Baru, C. K., R. W. Moore, A. Rajasekar, and M. Wan: 1998, 'The SDSC storage resource broker.'. In: *CASCON*. p. 5.

8. Baud, J.-P., J. Casey, S. Lemaitre, and C. Nicholson: 2005, 'Performance analysis of a file catalog for the LHC computing Grid'. In: *Proceedings of High Performance Distributed Computing 14*.
9. Beltran, V., J. Guitarat, D. Carrera, J. Torres, E. Ayaguade, and J. Labarta: 2004, 'Performance Impact of Using SSL on Dynamic Web Applications'. In: *Jornadas de Pralelismo*.
10. Blumenfeld, B. and *et al.*: 2004, 'FroNtier: High Performance Database Access Using Standard Web Components in a Scalable Multi-tier Architecture'. In: *Computing in High Energy and Nuclear Physics*.
11. Buncic, P., A. J. Peters, and P. Saiz: 2003, 'The AliEn system, status and perspectives'. In: *Computing in High Energy and Nuclear Physics (CHEP)*.
12. Deelman, E., G. Singh, M. P. Atkinson, A. Chervenak, N. P. C. Hong, C. Kesselman, S. Patil, L. Pearlman, and M.-H. Su: 2004, 'Grid-Based Metadata Services'. In: *16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*.
13. EGEE project: 2005, 'EGEE gLite Metadata Catalog User's Guide: gLite Metadata Catalogue Interface Description'. *EGEE-TECH-573725*.
14. Foster, I. and *et al.*: 2001, 'The Anatomy of the Grid: Enabling Scalable Virtual Organizations'. *International Journal of High Prformance Computing Applications* **15**.
15. Foster, I. T., C. Kesselman, J. M. Nick, and S. Tuecke: 2002, 'Grid Services for Distributed System Integration'. *IEEE Computer* **35**(6), 37–46.
16. Gray, J., P. Helland, P. O'Neil, and D. Shasha: 1996, 'The dangers of replication and a solution'. In: *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA, pp. 173–182.
17. Harrison, K. and *et al.*: September 2003, 'Ganga: a user-Grid interface for ATLAS and LHCb'. In: *Proceedings of UK e-Science All 'ands Conference, Nottingham*.
18. Knobloch, J.: 2005, 'LHC Computing Grid - Technical Design Report'. Technical Report CERN-LHCC-2005-024, CERN, `http://lcg.web.cern.ch/LCG/tdr/`.
19. LCG 3D Project - Distributed Deployment of Databases. `http://lcg3d.cern.ch/`.
20. Lefebure, V. and J. Andreeva: 2003, 'RefDB: The Reference Database for CMS Monte Carlo Production'. In: *Computing in High Energy and Nuclear Physics (CHEP)*.
21. Montagnat, J. and *et al.*: 2006, 'Bridging clinical information systems and grid middleware: a Medical Data Manager'. In: *HealthGrid 2006*.
22. Rajasekar, A., M. Wan, R. Moore, and W. Schroeder: 2004, 'Data Grid Federation'. In: *PDPTA - Special Session on New Trends in Distributed Data Access*. pp. 541–546.
23. Santos, N. and B. Koblitz: 2006a, 'Distributed Metadata with the AMGA Metadata Catalog'. In: *Workshop on Next-Generation Distributed Data Management - HPDC-15*.
24. Santos, N. and B. Koblitz: 2006b, 'Metadata Services on the Grid'. *Nuclear Instruments and Methods in Physics Research* **A 559**, 53–56.
25. Singh, G., S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman: 2003, 'A Metadata Catalogue Service for Data Intensive Applications'. In: *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*. Washington, DC, USA, p. 33.