Available online at www.sciencedirect.com
SCIENCE DIRECT®
Nuclear Instruments and Methods in Physics Research A 559 (2006) 53–56
NUCLEAR INSTRUMENTS & METHODS IN PHYSICS RESEARCH Section A
www.elsevier.com/locate/nima

Metadata services on the Grid

Nuno Santos[a,b,*,1], Birger Koblitz[a,2]

[a]CERN, Geneva, Switzerland
[b]Department of Computer Science, University of Coimbra, Portugal

Available online 6 December 2005

Abstract

We present an interface for metadata access on the Grid, designed to support flexible schema management, efficient retrieval of large result sets and to allow a broad range of implementations. We also describe an implementation of this interface, which supports a wide range of storage back-ends and two access protocols: SOAP and a TCP-streaming-based protocol. This interface and implementation have been selected as the official metadata components of the gLite-EGEE middleware. Finally, we present the results of extensive performance studies, where the two front-ends are compared to evaluate the cost of using SOAP as metadata access protocol.

Keywords: Data Grid; Metadata; SOAP

1. Introduction

Data Grids often contain millions of files spread over several storage sites. To find the files of interest, users and applications need an efficient mechanism to discover and query information about their contents. This is provided by associating descriptive attributes (metadata) to files and by exposing this information in catalogues, which can then be queried to locate files based on the value of their attributes [1].

A metadata catalogue can also be regarded as a simplified database for jobs running on the Grid, which often need to retrieve or store non-file-related metadata too small or too volatile to be stored in data files. If this information can be modelled as metadata (i.e., in the form of `(key, value)` pairs with type information), then a metadata catalogue is often a good alternative to using a relational DB, offering a simplified interface and greater integration with other Grid services (e.g., GSI security).

A metadata service for use in this environment should satisfy some specific requirements. It must expose a complete but simple interface, so non-technical users can easily use it. It should be flexible and support dynamic schemas, since there is no single schema that will serve all application-domains. The service must also allow metadata to be structured as an hierarchy of logical collections, so that related metadata can be grouped together and isolated from other metadata. To deal with the 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. Finally, security is required to provide different access levels to different users.

In this paper we describe an interface for Metadata Services[3] that we have developed based on our experience with testing a number of custom metadata services used by several LHC collaborations at CERN. Although these services have similar goals and are built around similar concepts, they have incompatible interfaces and static schemas designed for a specific application-domain, limiting their reuse in other domains. Our interface generalises the functionality of these Metadata Services in a coherent and generic interface, suitable for most application-domain.

*Corresponding author. CERN, Geneva, Switzerland.
E-mail address: Nuno.Santos@cern.ch (N. Santos).

[1]Partially funded by Grant SFRH/BD/17276/2004 of the Portuguese Foundation for Science and Technology (FCT).
[2]Partially funded by Bundesministerium für Bildung und Forschung, Berlin, Germany.

[3]This interface was initially proposed by the ARDA group and then evolved jointly with the gLite (EGEE) Data Management team. It has since then become the official EGEE metadata interface.

0168-9002/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.nima.2005.11.104

The remainder of this paper is organised as follows. Section 2 describes our interface, Section 3 presents our implementation of the interface, Section 4 presents the results of a benchmark study of this implementation and Section 5 presents the related work.

2. The metadata interface

In order to create an interface generic enough to be used by many types of Grid applications, we had not only to design a generic set of operations for users, but also to allow a broad range of different implementations. No single implementation is likely to satisfy the metadata needs of the whole range of Grid applications, which vary significantly in their size and access patterns. Having this in mind, we designed our interface to hide as many implementation details as possible, allowing those decisions to be taken by implementors in the way that better suits the needs of the target applications. In the following discussion we will point out the main aspects where implementations might differentiate themselves.

The basic concepts of the metadata interface[4] are entries, attributes and schemas. An `entry` is the name of the data item or resource being described, an `attribute` is a `(key, value)` pair with type information, and a `schema` is a logical group of attributes. Entries are associated with one or more schemas and inherit the attributes defined in those schemas. This is the only way of associating attributes to an entry, it is not possible to have attributes associated directly with entries. The interface defines operations to add and remove entries from a schema, and to list the schemas to which an entry belongs. Also, entries cannot exist on their own, they must be created with at least one associated schema.

Schemas are defined dynamically by the user. There are operations to create and delete schemas, as well as to add and remove attributes from a schema. Since the schemas are dynamic, we provide methods to discover their attributes at runtime. Schemas are the basic blocks used to structure and organise metadata as logical groups. But much of the details of schema management are left open to implementations. For instance, implementations may either allow an entry to belong to multiple schemas or only to a single schema. Also, they may either organise the schemas in a flat namespace or in a hierarchy.

To design the query operations we considered several issues, starting by the query language. Since most implementations will use relational databases as back-ends, an SQL-based language was a natural option, with advantages both for users, most of which are familiar with SQL, and for implementors, who can delegate most of the query processing to the RDMS engine. Nevertheless, the Metadata Interface does not restrict the type of storage back-end and other implementations may not use relational databases. For instance, an implementation based on an XML datastore would probably use XQuery as query language. Since, no single query language is suitable for all possible implementations, we decided not to specify any at the interface, leaving this as an implementation detail. For the interface, queries are simply two text strings: the query itself and the name of query language being used.

The second issue is how to deal with large result sets. A naive implementation will read from the back-end all the results of a query in a single operation and send them to the client in one message. This does not scale for large result sets or for many clients due to the memory requirements. To address this issue, the interface uses iterators to retrieve responses in small chunks. This is implemented by the methods `query()` (initiates a query and retrieves the first bunch of results), `nextQuery()` (gets the next chunk of results) and `abortQuery()` (cancels a query). Queries are identified by an opaque token, obtained in the initial `query()` invocation, that must then be provided to `nextQuery()` and `abort()` methods. Implementations are free to choose either to use a stateful or stateless model to implement these methods. A stateful implementation backed by a relational database can use a database cursor to read the results from the back-end. This is efficient and ensures consistency of the results, since the query is executed only once, but requires the server to keep state between invocations from the client, increasing its complexity. A stateless implementation can use the `LIMIT` clause of SQL to return a specific range of results and use the opaque token sent to the client to store the current position in the results. This option is simpler to implement, but is less efficient (multiple queries) and has consistency problems since the database may be updated between two invocations from the client, changing the result set.

3. A prototype implementation

Together with the interface we have developed a prototype implementation called ARDA Metadata Grid Application (AMGA),[5] to validate the interface and receive feedback from users.

The AMGA implementation uses a file-system model for structuring metadata. Schemas play the role of directories: they may contain entries and other schemas, allowing users to create an hierarchical structure. Our experience with users shows this to have been a good option, since many of them are making heavy use of hierarchies for better organising their metadata. From now on we will refer to schemas as directories, since this better reflects the model of AMGA.

Access control is on a per directory basis, with all entries in a directory sharing the same ACL list. Having a per-item ACL would impose a large performance penalty for little added value. The implementation also supports groups of

[4]The complete specification of the interface is at https://edms.cern.ch/file/573725/1.2.

[5]The AMGA implementation was recently chosen to be the official metadata catalogue of the gLite middleware.

users. Other security features are authentication based on certificates, grid-certificates or password, and secure connections using SSL.

Entries can only be in a single directory. This simplifies access control, since allowing an entry to be in multiple directories could result in conflicts between the possibly contradictory security policies of the different directories.

AMGA is designed to use a relational database as storage. Each directory is a table, entries are rows and attributes are columns. Attributes are added or removed from directories by adding or removing columns from the directory's table. A master table keeps the index of all directories, together with some per-directory properties (e.g., ACLs). This structure is flexible and efficient. Most operations require only two accesses to the database: one to the index table and another to the table of the directory.

The prototype was implemented as a multi-threaded C++ server (Fig. 1). The back-end is modular, supporting several storage systems by way of modules. Most of storage modules we have developed are for relational databases, including PostgreSQL, Oracle, MySQL and SQLLite. We also created a stand-alone implementation that stores the metadata directly on the filesystem.

For operations that return large result sets, the server uses a stateful model. When the user sends a `query()` request, the server creates a cursor on the database to read the result set. It then sends the partial results to the client asynchronously: when the client is processing a chunk of the results, the server is already reading the next chunk into a local buffer, so it can answer immediately to the next request. Since database connections are kept open between calls from the client, there is the risk of running out of resources due to buggy or malicious clients. The server implements two mechanisms to prevent this situation: it kills sessions that are left unused for a long time and limits the maximum number of sessions a single user can open. A stateless server would not require these mechanisms, but it would have significantly worse performance (queries have to be repeated for each chunk of results sent to the client) and would also require complex mechanisms to ensure that results are kept consistent between calls of the same query.

The front-end supports two access protocols: SOAP and TCP-Streaming (TCP-S). The SOAP front-end is based on the gSoap toolkit [2]. The TCP-S front-end is based on a text protocol similar to SMTP or TELNET, where commands and answers are sent as plain text. Since this is a stream-oriented protocol, it is not possible to implement the interface as it is, since it is designed for a message-based protocol. Nevertheless, the commands supported by the TCP-S protocol mirror closely the operations defined on the interface. The main difference is in how large results are sent to the client, where we take advantage of the stream-oriented nature of the protocol, by sending the results back in a single stream of bytes. This is efficient, since it does not require several round-trips between the client and the server. For the TCP-S protocol, we have created an iterative command line interface to the server and client libraries in C++, Java, Python, Perl and Ruby.

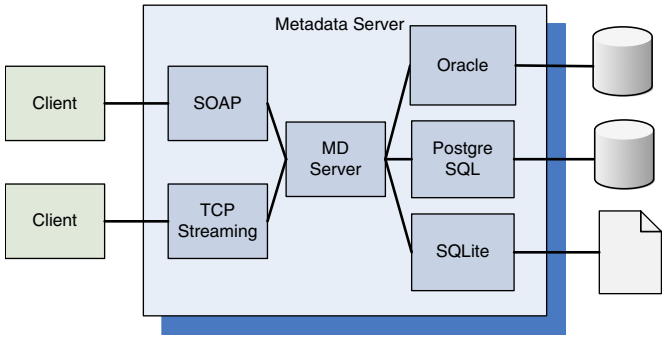

Fig. 1. Main components of the AMGA implementation.

Several applications have used or are using the AMGA implementation, either for evaluation or production. The LHCb collaboration has been evaluating the AMGA implementation using their bookkeeping information (20 million entries, 15 GB). They have uncovered many bugs and some limitations on the initial versions, which have since then been fixed. Another user with very different access patterns is Ganga [3], an user interface to submit jobs to the Grid being developed by LHCb and ATLAS. Ganga uses the AMGA implementation to store metadata describing the status of the jobs, consisting in a small but highly dynamic set of metadata.

4. Benchmark study

In this section, we present the results of a benchmark study comparing the SOAP and the TCP-S front-ends.[6] The tests were performed on two Linux desktop computers connected via switched fast Ethernet with a network latency of ≈ 0.1 ms.[7] The metadata server was pre-loaded with 100 collections, each containing 1000 entries. Each entry has 60 attributes, amounting to about 700 bytes of data.

Fig. 2 presents the results of reading 1000 entries from the server using two access methods: in `single` entries are read one at time, while in `bulk` they are read using a single query. The `bulk` method shows the benefits of the iterators defined in the interface, that allows large responses to be read in a single query. Results were taken for different numbers of concurrent clients. Each client was reading from its own directory.

Reading in bulk is a factor of 10 faster than reading single entries for both protocols, showing the importance

[6] Due to space limitations, we can only present part of the results. Please consult the ACAT'05 presentation at http://www-zeuthen.desy.de/acat05/talks/Santos.Nuno.1/Metadata.ppt for the complete study.

[7] The client PC was a dual 2.4 GHz Xeon with 1 GB RAM. The server a dual Pentium III (800 MHz, 0.5 GB RAM). The more powerful client was used to simulate multiple clients. In all tests with multiple clients it was made sure that the client PC was not limiting performance.

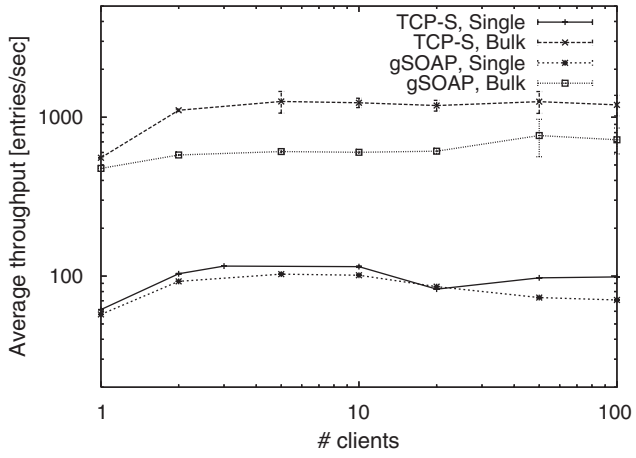

Fig. 2. Throughput of the metadata service while reading 1000 attributes using single and bulk operations.

of minimising the number of interactions with the server. Comparing the two protocols, we can see that SOAP is at least 2 times slower than TCP-S: 1000 queries per second for TCP-S against 500 for SOAP. In the other tests we performed the overhead of SOAP was even greater, varying between 2 and 5 times.

gSoap is considered one of the fastest SOAP toolkits [4], but most Grid applications being developed nowadays are written in Java and therefore use Apache Axis. We have evaluated the relative performance of these toolkits by comparing clients implemented in gSoap (2.7.0f) and Axis (1.2RC3). We have also tested a client in Python using ZSI (1.6.0). The test consisted in making 1000 null requests. gSoap took 4 s to complete the test, Axis 11 and ZSI 25. All these tests were made using the gSoap-based server. If we had also changed the toolkit on the server to match the one in the client, the results for Axis and ZSI would be even worse. These results show clearly that SOAP toolkits vary widely in performance. We have performed the same test with the TCP-S protocol using clients written in C++, Java and Python, and found no significant difference in performance (all clients took between 3 and 4 s), showing that the programming language by itself is not the limiting factor.

5. Related work

A metadata service with similar goals is the Metadata Catalog Service [1,5]. The model and structure used for metadata is similar to our own, supporting flexible schemas and an hierarchical organisation. The authors have provided two implementations, one based on Web Services and the other on OGSA-DAI [6] Grid Services. Our work differs in several aspects. First, our metadata interface was designed to give as much freedom as possible to implementations, so they can adapt to particular segments of applications. Another difference is that we address explicitly the problem of returning a large result set over stateless protocols like SOAP, by using iterators and sessions. There are also some important differences in our implementation. To allow easy deployment in the available infrastructure of any Grid site, it is designed to be DB independent without requiring external middleware like OGSA-DAI. And for applications with high-performance requirements, it supports an efficient TCP-s protocol, which we have shown to be 2–5 times faster than the alternative SOAP protocol.

6. Conclusions

We have presented the EGEE interface for metadata access on the Grid, which is designed to cover a broad range of applications. It supports dynamic schemas, an iterator pattern to retrieve large result sets using message-oriented protocols and a high level of abstraction that gives developers freedom to explore different types of implementations. We have also described our implementation, which was chosen as the official gLite-EGEE Metadata Catalogue. Its main features are the support for different relational databases as storage back-end and the support for two access protocols: SOAP and a custom-designed protocol based on TCP-s. Finally, we have presented the results of a performance study comparing these two protocols, where the TCP-s-based protocol is shown to be 2–5 times faster than SOAP.

Acknowledgements

This work was performed within the ARDA project and the authors would like to thank in particular V. Pose (Dubna, Russia) for his intensive studies and testing of the streaming protocol. Also we would like to thank the GridPP and EGEE-gLite teams for their collaboration on metadata ideas.

References

[1] E. Deelman, et al., 16th International Conference on Scientific and Statistical Database Management (SSDBM'04), 2004.
[2] R.A.V. Engelen, K.A. Gallivan, CCGRID '02: Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid, Washington, DC, USA, 2002, IEEE Computer Society, Silver Spring, MD, p. 128.
[3] Ganga—Gaudi/Athena and Grid Alliance, ⟨http://cern.ch/ganga/⟩.
[4] M. Govindaraju, et al., GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04), Washington, DC, USA, 2004, IEEE Computer Society, Silver Spring, MD, pp. 365–372.
[5] G. Singh, et al., SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, Washington, DC, USA, 2003, IEEE Computer Society, Silver Spring, MD.
[6] M. Antonioletti et al., Concurrency and Computation: Practice and Experience, 17 (2005) 357.