Grid Enabled Relational Database Middleware

Informational Document
Global Grid Forum
Frascati, Italy, 7-10 October, 2001

Wolfgang Hoschek
wolfgang.hoschek@cern.ch
CERN - European Organization for Nuclear Research
1211 Geneva 23

Gavin McCance
g.mccance@physics.gla.ac.uk
Department of Physics and Astronomy
University of Glasgow, Scotland

Introduction

In DataGrid projects many small and large applications, spread over multiple loosely coupled organizations, work together to provide access to and mangement of massive amounts of distributed data. Many of these applications, such as a replica catalog, service registry, run catalog tools, cluster configuration management tools, job schedulers, logging and performance monitors need to take advantage of reliable and performant database technology to maintain metadata. This technology should involve little complexity, deliver high performance and encourage interoperability. Although a wealth of strong database technology exists, complexity is typically substantial, interoperability weak, and performance not always satisfactory. Until these points are addressed more thoroughly, DataGrid applications will continue to use dozens of varying proprietary approaches towards metadata storage and retrieval.

The *Spitfire* project of the Data Management Work Package (http://cern.ch/grid-data-management) within the European Data Grid Project tries to address these issues. It provides a set of grid enabled middleware services for access to relational databases. The key idea is to decouple client and RDBMS by having a mediator in between them, enabling ease-of-use, interoperability, performance and plugability. Custom coded clients (Java, C/C++, Perl et al), browsers (netscape, iexplorer et al) and command line tools can read and write over HTTP(S) into any RDBMS (MySQL, Oracle, DB/2, Postgres et al). Currently Spitfire consists of the SQLDatabaseService and the `jwget` tool for client side command line usage. In the near future the ServiceIndex and a fine grained authorisation mechanism will be added. This work is embedded into the overall Data Management Architecture, available at http://cern.ch/grid-data-management/docs/DataGrid-02-D2.2-0103-1_2.pdf.

The SQLDatabaseService is a grid enabled front end for reading and writing from/to a relational database. The architecture can be summarised as follows:

```
          XML/HTTP(S)                              JDBC
Client       <-->        SQLDatabaseService      <-->     RDBMS
```

Client and middleware service communicate relational data in canonical XML format over GSI enabled HTTP(S). Canonical XML defines a 1:1 mapping from relational tables to XML format and vice versa (see below). Client, middleware service and RDBMS can

run on the very same box but can also run on three different boxes (and architectures) separated by the LAN or WAN. The middleware is implemented as a Java servlet and designed to be callable from command line tools (`jwget`, `wget`, `curl` et al) and custom client applications, for example using the xerces XML parser (Java, C/C++, Perl) and httpclient (Java), libwww (C/C++), libwww-perl (Perl) or httplib (Python). However, the middleware is also designed to be used by browsers as clients: Resultsets are delivered in nicely formatted HTML tables (instead of XML) if indicated by a request parameter.

The HTTPS configuration is fully GSI compatible by using CoG, the Java port of the Globus toolkit developed at Argonne National Laboratories (http://www.globus.org/cog).

Effort has been invested into providing high performance and low latency: Java servlets are highly efficient and scalable solutions. To minimize the number of connection setups and tear-downs, persistent HTTP(S) 1.1 connections are used (but not required). For similar reasons, the services use thread and JDBC connection pooling, and advanced caching. The net effect is that requests and responses are passed through layers with very little delay.

Interoperability

Industry standards foster cross-organisational interoperability and reuse. In addition, one gains a large variety of mature, easy to use and performant open source implementations to build upon.

With HTTP command line clients, GUI browsers, client programming language APIs and server frameworks available for virtually every language and architecture, HTTP and its SSL based secure variant HTTPS have been firmly established as the industry standard protocol for interoperable networking (and not just as vehicle to deliver HTML). Once SOAP (in practice a small formalism on top of HTTP) has reached similar ubiquity, it may be interesting to support it too.

Similarly, with XML parsers now being widespread, performant, easy to use and available for virtually every programming language, XML has been established as the industry standard to represent flexible interoperable data.

With mature open source and commercial implementations available from the low to high end, relational database systems are established as the storage backbone of IT industry. Although SQL implementations vary slightly among vendors, SQL is comfortably standardized for the vast majority of applications. The JDBC interface defines an implementation independent mechanism to communicate with a RDBMS. It is a thoroughly accepted industry standard and optimized drivers exists for every serious RDBMS. Note that the JDBC interface is a Java API and does not specify a network protocol. This design decision allows for high performance driver implementations specially geared towards any given RDBMS, taking full advantage of its capabilities. On the other hand this design decision also excludes JDBC as a standard grid protocol. A network protocol such as GSI enabled HTTP(S) can fill this gap.

Client and middleware services communicate relational data in canonical XML format over HTTP(S). Canonical XML defines a 1:1 mapping from relational tables to XML format and vice versa: A relational table corresponds to a XML ROWSET element, a row corresponds to a nested XML ROW element, and a column is mapped to a nested element with the same name, filled with the value of the row's column. Let us introduce canonical XML by example.
Assume we have a relational table with Logical File Name (LFN) and Physical File Name (PFN) columns:

LFN	PFN
`lfn://cms.org/file1`	`ftp://host1.cern.ch/myfile1`
`lfn://whatever.org/file2`	`ftp://host3.anl.gov/data/file.tar.gz`

The corresponding canonical XML representation of the table looks as follows:

```
<ROWSET>
   <ROW>
      <lfn> lfn://cms.org/file1 </lfn>
      <pfn> ftp://host1.cern.ch/myfile1 </pfn>
   </ROW>
   <ROW>
      <lfn> lfn://whatever.org/file2 </lfn>
      <pfn> ftp://host3.anl.gov/data/file.tar.gz </pfn>
   </ROW>
</ROWSET>
```

Whitespaces are ignored. There is a straightforward 1:1 mapping between relational tables and XML format. For further details consult http://hep-proj-spitfire.web.cern.ch/hep-proj-spitfire/share/spitfire/doc/xsql/readme.html.

Example Use Cases

Example 1: Logical File Name to Physical File Name Lookup

Say we have an efficient and highly available file sharing system where a file can be replicated on many hosts. To keep track of the files we have a catalog table holding all physical locations of any logical file (1:N mapping). We would like to provide clients with a function that looks up and returns all physical file names for a given logical file name. A client should not be concerned how results are being computed so that we can change the implementation at some later point in time without breaking the clients (maybe we use SQL database or a flat file, or an LDAP database). We need a function such as

```
PhysicalFileName[] getPhysicalFileNames(LogicalFileName)
```

This can be implemented at the sqldb server side by writing a small text file that contains one or more SQL template statements with substitution variables as necessary. Here we

use a simple SELECT FROM WHERE statement with variables {@table} and {@lfn}, embedded into a <xsql:query> command.

```
<?xml version="1.0"?>
<xsql:query connection="demo" xmlns:xsql="urn:oracle-xsql">
        select pfn from {@table} where lfn="{@lfn}"
</xsql:query>
```

The SQL statement is extended with XML syntax to define xmlns:xsql as namespace. The connection tag refers to JDBC parameters in a configuration file, indicating DB backend URL, JDBC driver and tuning options.
Next, copy the template text file somewhere into a directory readable by the middleware server (e.g. tomcat-work/webapps/sqldb/demo/getPhysicalFileNames.xsql). The server will automatically pick up the new function and any changes that may be applied later (just like a webserver automatically picks up HTML files in its web path without explicit configuration).

To get all physical files for a logical file one can use a web browser, command line client or HTTP API to issue an HTTP request like

```
http://myserver.org/sqldb/demo/getPhysicalFileNames.xsql?table=repcat&l
fn=lfn://cms.org/myfile1
```

Here is how the result may look

```
<?xml version = '1.0'?>
<ROWSET>
   <ROW num="1">
      <pfn>ftp://host1.cern.ch/myfile1</pfn>
   </ROW>
</ROWSET>
```

A client can parse the result with any XML parser and do further processing as necessary. Note that the middleware serves as an effective way to shield clients from any backend implementation details. In effect it provides an abstract function invocation interface.

Example 2: Adhoc SQL query

Say we would like to provide clients the means to execute arbitrary SQL queries. This breaks encapsulation and is a potential security hazard but is sometimes useful for privileged authorised clients. We need a function such as

```
ResultSet query(Query)
```

This can be implemented at the sqldb server side by writing a small text file that contains a single substitution variable {@query} for the entire SQL statement.

```
<?xml version="1.0"?>
<xsql:query connection="demo" xmlns:xsql="urn:oracle-xsql">
        {@query}
```

```
</xsql:query>
```

After the text file is copied into the server path an HTTP request invokes the function for selecting all entries of the replica catalog:

```
http://myserver.org/sqldb/demo/adhocquery.xsql?query='select * from
repcat'
```

Here is how the result may look

```
<?xml version = '1.0'?>
<ROWSET>
   <ROW num="1">
      <lfn>lfn://cms.org/file1</lfn>
      <pfn>ftp://host1.cern.ch/myfile1</pfn>
   </ROW>
   <ROW num="2">
      <lfn> lfn://whatever.org/file2 </lfn>
      <pfn> ftp://host3.anl.gov/data/file.tar.gz </pfn>
   </ROW>
</ROWSET>
```

Example 3: Insert data

Assume clients need to insert data into the replica catalog table without wanting to know how the catalog is implemented. Hence, we need a function such as

```
insertIntoReplicaCatalog(data)
```

This can be implemented at the sqldb server side by writing a small text file that contains a `<xsql:insert-param>` command with a substitution variable `data`.

```
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
<xsql:insert-param name="data" table=repcat"/>
</page>
```

After the file is copied into the server path an HTTP request invokes the function to insert all rows specified by variable `data` into table `repcat`. Input should conform to canonical XML format.

```
http://myserver.org/sqldb/demo/insertIntoReplicaCatalog.xsql?data=
"<ROWSET> <ROW> <lfn>lfn://cms.org/spitfile</lfn> <pfn>
ftp://myhost.cern.ch/myfile0</pfn> </ROW> </ROWSET>"
```

Implementation

The Spitfire implementation reuses existing technology where possible. To this end, several third party components are reused, including Java, the Apache Tomcat servlet engine (a dynamic HTTP server framework), XSQL, httpclient, COG and MySQL as the default database backend.

For convenience the software is completely self-contained and ships as a bundle containing all components necessary to run "out of the box". The bundle is designed to get users up and running with minimal installation and configuration problems. A single download and `untar` or `rpm` step will get a user operational (no extra configuration required). The bundle contains the Spitfire core, including source and binaries, plus a binary distribution of Java, Apache Tomcat and MySQL. Linux and Solaris flavours are included.

The packaging is designed both for ease of use and flexibility: Once a user is comfortable running the services and their defaults, alternative or more recent versions of Java, MySQL or other RDBMS systems can be plugged in easily and configuration options can be customized as necessary. For example, if a site is already running one or more of the bundled components or equivalent substitutes in a well managed and reliable production environment (e.g. running DB/2, Oracle or Postgres instead of MySQL) one can set environment variables to substitute the defaults with alternative components, paths and configuration files. A runtime auto detection mechanism automatically picks up new choices.

Conclusions & Future Work

Many DataGrid applications need to take advantage of reliable and performant database technology to maintain metadata. The technology should involve little complexity, deliver high performance and encourage interoperability. The Spitfire project tries to address this by providing grid-enabled relational database middleware with a focus on ease-of-use, interoperability and performance.
With a stable release available and in the near future deployed on the DataGrid Month 9 Testbed, future work will focus on performance and scalability studies, a proof-of-concept implementation of a hierachical distributed replica catalog, a fine grained authorization mechanism and a distributed service registry application.