2009 International Conference on Advanced Information Networking and Applications

A Generic Database Web Service for the Venice Service Grid

Michael Koch, Markus Hillenbrand, Paul Müller
University of Kaiserslautern
PO Box 3049
D-67653 Kaiserslautern, Germany
{m_koch2, hillenbr, pmueller}@informatik.uni-kl.de

Abstract

This work describes a generic database service for the lightweight Venice Service Grid, which has been developed at the University of Kaiserslautern, Germany. By using Web services any SQL query can be processed and the corresponding result sets are sent back as strongly typed XML documents. The accurately defined interface allows an easy-to-use, platform and programming language independent access to all applications of a service federation. Transactions, prepared statements and parallel transfers of split result sets are supported, too. Because of a JDBC-like programming interface, the service can be used intuitively and existing software components can be quickly adjusted to the new system.

1. Introduction

In 1969 Leonard Kleinrock imagined "the spread of computer utilities, which, like present electric and telephone utilities, will service individual homes and offices across the country" [1]. Ian Foster and Carl Kesselmann in 1998 initially tried to define a Grid that could help implement such computer utilities [2]. After several iterations, Ian Foster finally published his three point checklist [1] that must be met by a system to be called a Grid.

Today, a distinction between *Compute Grids*, *Data Grids* and *Service Grids* can be made to distinguish between the Grid systems available. There is no common definition for these Grid types, but here they are understood as follows. A Compute Grid provides resources for high throughput computing and makes possible time consuming calculations over the Internet (e.g. by giving standardized access to clusters). A Data Grid makes available disk space to securely store large data sets. A Service Grid finally uses these Grid types as a commodity and adds more value to the end-user by providing special services like virtualized applications or accurate algorithms and offering intuitive graphical interfaces to access those services.

In today's Grid environments performant and reliant database management systems are required. Various loosely coupled applications need to get access to diverse data. Techniques allowing a uniform access (JDBC, ODBC, etc.) to mostly relational databases are existant but often they are very complex or lead to much overhead on the client side for management of different drivers for varying database managements systems. Firewalls and security policies are big problems in these overregional systems, too. Therefore new concepts have to be developed to provide access to heterogeneous databases while client systems can manage these data sources with a uniform standard, independent of the programming language while considering security issues of the distributed organizations.

Venice is a lightweight Service Grid [3], [4] which offers a framework for the easy and fast development, installation and management of distributed systems. On one hand it gives the opportunity to use a Single Sign-on mechanism (SSO) [5] which assures a secure authentication and authorization within a corporate service federation over the Internet. On the other hand an Information Broker (IB) supports the distribution of service information with modern and highly scalable Peer-to-Peer technology.

This work introduces a Generic Database Service (GDS) based on Venice, implemented with Web services and the central concepts of service-oriented architectures. The service is able to process almost every query of the SQL-standard and supports the transmission of large result sets by splitting it into several SOAP messages. These parts are passed in parallel to guarantee short latencies. Result sets are stored in a strongly typed SOAP-format specified by an accurately and securely designed XML schema. Therefore the interface can be easily used by any developer without knowledge of the internal logic and independent of the programming language. Furthermore a JDBC-like API has been developed which offers a intuitive and optimized access to the Generic Database Service. In this way the developer is familiar with the API and existing software components can be easily adapted to use the advantages of the new system. Besides it supports the pooled usage of precompiled statements, transactions without auto commit and other features of the JDBC API by saving state information on the client side. Optionally exists a right management structure which associates the underlying database with the Venice SSO mechanism.

1550-445X/09 $25.00 © 2009 IEEE
DOI 10.1109/AINA.2009.32

IEEE computer society

In section 2 this work is compared to related work. Section 3 explains the overall architecture and describes the different components of the Generic Database Service. Performance measurements in several scenarios are presented in section 4 while section 5 concludes the paper and gives some future prospects.

2. Related Work

New ideas and concepts for accessing remote data sources are an important topic in research and industry. In this section important related projects and their approach to accessing remote databases are discussed and compared to the solution presented in this paper.

The JDBC Web Service (JDBC-WS) [6] allows access to a DBMS per URL without using any driver. Likewise Web services are used to transfer queries and result sets. But they are not validated against an XML schema. This system can be seen as the basic idea of this implementation but the Generic Database Service as enhancement has more functionality, is more scalable and delivers a more performant system (see section 4).

Another approach is the Spitfire-Project [7] within the *European Data Grid Project*. Tools like *jwget*, *wget*, *curl* or normal browsers are supported by usage of HTTP connections to a middleware. It connects to a RDBMS with the normal JDBC API, too. The result sets are transferred back in a canonical XML format. In this way it offers good interoperability but does not provide strongly typed datatypes. On the other side the interface is not accurately designed so that there must be another type of contract between client and provider. The transfer of large data amounts is also not discussed in the work.

Based on the WSRF-standard OGSA-DAI [8], [9], [10] offers different port types for stateful Web services of various data sources. Therefore this approach is a totally different concept because the Generic Database Service normally does not store state information about an SQL connection. There is no persistent connection between client and service provider at any time. However the *Grid Data Service* offers access to RDBMS and is comparable to this work. It is designed for the Globus Toolkit [11] and thereby it is developed to handle large amounts of data and different query-languages (SQL, XPATH, etc.) in huge heterogeneous systems are supported by the usage of special perform documents. In this context OGSA-DAI has to handle more complex functionality while the Generic Database Service only provides access to a single DBMS at this time. On the other hand the Generic Database Service offers an easy to use JDBC like API while providing nearly the same performance (see 4). Also new concepts like the split data transfer of SOAP messages can be used in larger scenarios, too. OGSA-DAI uses GridFTP to transfer the result sets. The system in this paper is tailored for the needs of Venice. So it abdicates on the validation of large XML query documents and just uses simple SQL statements embedded as strings in a small SOAP document. Summarized, the advantages of the work presented in this paper are the intuitive usage, short latencies and new transfer concepts.

3. Architecture

The generic interface to databases is provided by a single service within the Venice Service Grid enabling access to a PostgreSQL [12] database management system. Figure 1 presents the overall architecture.

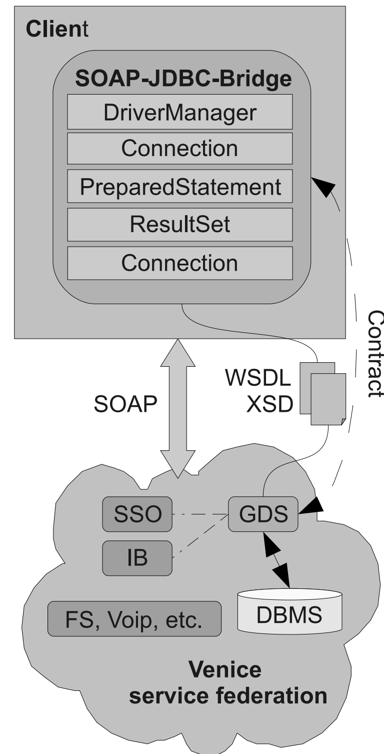

Figure 1. Architecture of the Generic Database Service: Clients use the optional JDBC-like wrapper system (SOAP-JDBC-Bridge) to gain access to the underlying PostgreSQL-DBMS within a Venice federation.

A client uses the service via a JDBC-like wrapper system called SOAP-JDBC-Bridge and achieves access to the underlying DBMS. Of course, it can be used without this bridging mechanism by any application independent of the programming language. All needed information are included in the WSDL file and the XML schema. The service itself is implemented in Java and uses connection pooling mechanisms for JDBC to connect to the database.

The following points describe the requirements of the architecture that are fullfilled in the implementation:

- Support of heterogeneous clients independent of the programming language by usage of Web services.

- Database abstraction. Till now only PostgreSQL is supported but it is possible to use other DBMS without changing the clients and small changes on the server side if required. Driver management is only required on the server side.
- Efficient datatype conversion with strong typing and transfer of large result sets.
- Intuitive and transparent usage of the system on the client side and easy adaption of existing applications. This is supported by a JDBC-like API (SOAP-JDBC-Bridge: SJB).
- Performance optimization of queries with caching and pooling mechanisms.
- Coverage of the JDBC API with operations of the GDS.
- Access control of databases and tables under control of the GDS in conjunction with the Venice Single-Sign-on (SSO) mechanisms.
- Adherence to the central SOA concepts, especially abstraction, autonomy and statelessness [13].

In the next sections the main parts of the architecture and the key functionality are described in greater detail. At first it is shown how a SQL query can be triggered and controlled with a SOAP message and its corresponding operation. The next part presents the strongly typed XML schema and how result sets are composed in this way. After that the splitting mechanism is shown. Then the operations provided by the interface are described. In addition the SOAP-JDBC-Bridge and the parallel transmission of results is introduced. Finally the combination of Venice SSO and PostgreSQL rights management are presented.

3.1. Performing a Query with a SOAP message

This section shows how a query can be performed by using a simple SOAP message. Table 1 presents a schematic description of this document.

Name	Description
dbname	The database name to connect to.
queryString	A dynamic or static SQL query.
valueTypes	Declares which value types are used in a prepared statement.
values	Provides the different values for a prepared statement according to the value types.
isolation	Defines the isolation level for the query.
autoCommit	Configures whether explicit calls of commit and rollback operations for transactions are necessary.
persistentId	Turns on a hash based pooling mechanism.

Table 1. SOAP SQL Query Message

In this way all information required to connect to the database are provided by the client in every single request. No state information has to be stored by the service itself as services in service-oriented architectures are presumed stateless. Prepared statements can be performed by defining the appropriate datatypes and their values in the correct sequence. This design makes it possible to construct the JDBC-like API which is explained in 3.5. Also there are three more options which influence the behavior of the query and the service. *Isolation* (none, read_committed, read_uncommitted, repeatable_read, serializeable) affects the transaction level. This feature has to be supported by the underlying database management system. If *autoCommit* is turned off, several queries can be performed until a timeout occurs, the rollback or the commit operation of the GDS is called. They are assigned to the client by a unique identifier. The persistent mode functions in a similar way. The service now reserves one connection of the pool to register prepared statements and reuses them by comparison to hash values of the original query string. Therefore the statements of one single client don't have to be recompiled all the time. This leads to more performance while executing large INSERT operations in which many values have to be inserted into the database.

After the configuration of this document an according operation like *execute* or *query* of the GDS is called. The document is passed as a parameter. Normally the service analyzes the options, picks one connection object from the pool and then uses the corresponding JDBC functions to perform the query on the DBMS itself. After processing, the result set has to be converted into a SOAP message again.

3.2. Result Sets, Type Mapping and Strongly Typed Data

At this time the JDBC result set from the previous query is available and the service can work with these values. All the different datatypes have to be mapped to corresponding types in the SOAP message (XML document) and later on to the appropriate types of the specific programming language. In this work the client is also implemented in Java and therefore only this aspect is shown at this point. Table 2 presents the used datatypes and its mappings.

The datatypes and metadata of a single result set and the underlying database table are discovered dynamically by usage of the JDBC API. After that a SOAP/XML equivalent has to be found. For every row in table 2 the XML schema defines an adequate element with the related type which is shown in column two. The elements can be accessed with JavaBeans [14]. The XML schema can be found under address *http://www.v-grid.info/types/basic/sql.xsd* (*tns:SQLRow*). The XML container (*tns:SQLValueSet*) of the result set has to be very variable so that every combination of datatypes can be handled. The values are stored in arrays of the assigned type so that a datatype can be reused. Therefore the sequence of the accessed types is stored as metadata (*tns:colType*) which can be used by the client to reproduce the original result set. As an example, a small result set with two *int4* and one

varchar as datatypes would be represented as *array(xsd:int, xsd:int)* plus *array(basic:String)* while the datatype sequence would be stored in the *coltype* array. All other arrays would be nil (null). The row is then stored in a row array and can be transferred as a complete result set.

JDBC	SOAP	JAVA
int8	xsd:long	long
int4	xsd:int	int
bool	xsd:boolean	boolean
bpchar	basic:String	String
varchar		
text		
time **timetz**		Time
timestamp **timestamptz**		Timestamp
date	xsd:dateTime	Calendar
float8	xsd:double	double
float4	xsd:float	float
numeric	xsd:decimal	BigDecimal

Table 2. JDBC-XML-Mappings

For one prepared statement of the SOAP query message a similar conversion mechanism is used. But because the JDBC API has a very loose type coercion, many different types of the host language can be used to access the corresponding datatype in the database table via JDBC on service side. Type errors are transferred back to the client application as normal SQLException via the Axis [15] fault mechanism. This enables some kind of transparency. For Java there are several predefined datatypes for prepared statements which can also be used by any other programming language by appropriate mapping functions on the client side.

The idea behind this concept is the usage of strong data typing whereas input and output message of operations are completely defined by an XML schema with additional constraints on the permitted values [16]. This enables tight control on message values by restriction of the length of string values, range of numerical values and permitted sequences of values. On one hand this influences security aspects and on the other hand the interoperability. The WSDL file and XML schema describe the usage of the interface without an additional contract. No additional manual negotiation between consumer and provider is necessary to establish the format of encapsulated data. As shown in the section about related work, several other systems are implementing a database interface with XML, too. But all of them are using a canonical format [17] which emulates the database structure but not the datatypes and the client has to get more metadata. Furthermore a loosely typed WSDL interface is required to transfer the XML result set within a generic data type (*String*, *Base64-Encoded*, *xsd:any*, *xsd:anyType*). So an extra negotiation between providers is required and the content can not be securely controlled. An understanding of the WSDL alone is not sufficient to invoke the service. So the "arbitrary" XML message as a result set has to be extracted and parsed. An understanding of raw XML manipulation is needed to extract the different rows. Besides in many cases the table structure is not important for the client because it is known from the beginning and only the contents of the rows matters. So there is much unnecessary expense. The concepts developed in this work enables a secure and transparent way of accessing databases and convey automatic processing of result sets on the client side. Section 3.5 shows that these structures can be used in an optimized way by implementing a client side wrapper system. This way the developer does not have to know anything about XML processing.

3.3. Splitting Result Sets

One disadvantage of converting a result set or data at all into an XML format is the large overhead coming from metadata. There is no big difference in an canonical format or the format with strong typing which is described here. Many system (like Axis) restrict the file size of SOAP messages, too. So a workaround has to be found because the GDS should not be limited in any way. Other system like OGSA-DAI use different techniques to solve this problem and enable the client to use different communication strategies for different circumstances (GridFTP, transfer all at once, streaming, asynchronous mechanisms, etc.) [8]. On one side these offer much more flexibility and very large data sets can be transferred but on the other side it is getting much more complicated and the whole system has to rely on other components.

At this time Venice as a Service Grid only needs short and fast queries to obtain and save data within its services. GDS just splits large amounts of data into two or more messages so no other components have to be used. To do so the size of the XML message is pre-calculated and several SQLValueSets are generated if a limit is reached. To guarantee short latencies the messages size is very low. The first message with some information about the message amount and an unique identifier is sent back to the client. After that the client can get the rest of the messages. The service caches the messages only a short time. This process is only restricted by ressources on client and server side.

3.4. Operations of the Generic Database Service

The GDS interface provides several operations to control the service. Table 3 gives an overview of the supported operations.

With knowledge of these operations and the SOAP messages it is possible to use the GDS. The next sections present optional enhancements and additional features.

Operation	Description
getDatabaseList()	Shows all permitted databases under control of the GDS.
databaseExists() tableExists()	Simple helper functions to check existence of databases and tables.
createDatabase() dropDatabase()	Administration of databases is only allowed with these operations, not within SQL queries.
query() execute()	Synchronous query operations.
nextResultSet()	Get split messages.
commit() rollback()	Used to control transactions.
setPermissions() getPermissions()	Get and set permissions on tables and databases within a Venice service federation.

Table 3. GDS - operations

3.5. The SOAP-JDBC-Bridge and Parallel Transmission

On one hand the developed service can be used programming language independent without any disadvantages. On the other hand there is one problem while working with XML documents. Especially accessing multi-array structures can be very confusing within a programming language like Java. In conjunction with goals of Venice the GDS should be easy to use. Therefore the SOAP-JDBC-Bridge (SJB) was implemented which can be seen as an enhancement for Java applications (although this library can be implemented for other programming languages, too). The concept of this wrapper system is to use the structures and metadata of the strong typed SOAP messages while hiding the underlying XML schema completely. Database programmers are used to access PostgreSQL with the JDBC API and a lot of applications are implemented with it. So the SJB provides a similar API. Until now the wrapper consists of the following Java classes: DatabaseManager, Connection, PreparedStatement, ResultSet, ResultSetMetaData and SQLException. There is no need for a driver manager. As one of the advantages of the GDS, client systems don't need any driver. DatabaseManager and Connection are needed to control and configure the SJB. There is no established persistent connection between client and server. In this way no state information are stored on the server side, the context is handled by the SJB and its corresponding SOAP message and the server can optimize and cache queries by using advanced connection pooling mechanisms.

All the datatype conversion and splitting of SOAP messages is hidden completely. Even the original SQLExceptions are handed back. Extensive research is done for the development of composite Web services regarding fault handling [18]. In contribution to that the Venice Service Grid and of course the GDS use forward error recovery instead of backward error recovery. Thus the original error messages are forwarded to the client system so that the state of the component can be corrected. Analysis showed that this approach is more effective in these contexts. So the GDS can be used in an intuitive and transparent way. In addition optimizations can be done. For example split messages are transferred in parallel by starting several threads without blocking the application itself. Several features like bulk imports or asynchronous query operation are capable of being integrated, too, but not yet implemented.

3.6. Venice Single Sign-on and the DBMS

Every database that is created with the GDS is identified by a unique identifier (domain, user name, chosen database name) provided by the Venice SSO mechanism. User names in the Venice federation can be either client applications or services themselves that are trying to use the GDS. So it is simple to check if one user owns a database; but this database should be accessible by other users under the owner's terms, too. The granularity thereby is restricted to read/write permissions on complete databases and tables. Thus only a small additional access control system is needed.

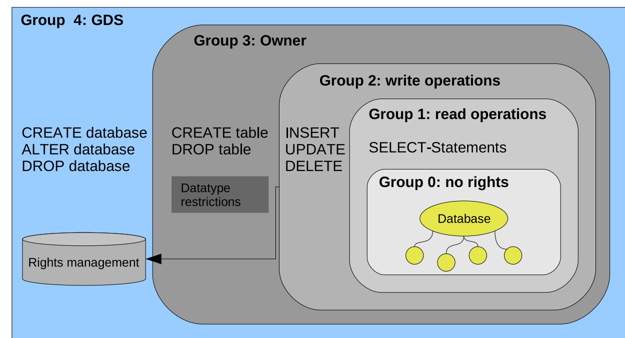

Figure 2. Rights management: Different access levels are associated to identifiers generated out of Venice SSO information.

The rights management of the PostgreSQL DBMS respectively of the SQL standard is under several aspects not ideal for this purpose. It is very complex and has much more features than needed. Furthermore the database may not be under the full control of the GDS. It can be used manually and from other applications. Unpredictable states can emerge this way. The GDS DBMS user may not have enough rights to hand permissions down to other newly created users. And at last the connection management and pooling mechanisms of the GDS would be too complicated because every user would need an own pool. Therefore figure 2 shows the own rights management system based on five categories.

This design is similar to Mandatory Access Control lists [13]. Databases can be created with the corresponding interface operations and can be only managed by the GDS

```
# write permissions
(?:insert\s+into|update|delete\s+from(?:\w+unique)
    ?)+\s+(\w+)
# read permissions
select[\w\s,\*\(\)]+from\s+(\w+)
\s*[\w_]+\s*\(
\s*(?:only|natural)?\s*(?:(?:left|right|full){1}\s
    *(?:outer)?\s+join|(?:inner)?\s*join|cross\s+
    join)?\s*(?:on\s+\w*|(\w+)){1}\s*\*?\s*(?:(?:
    as\s+\w+|using)?\s*(?:\([\s\w,]+\))?)?
```

Listing 1. Analyzing query strings with regular expressions

itselves (Group 4). Every user who creates a database with these operations is automatically its owner and can access it with the whole SQL standard (Group 3). Of course there are some restriction caused of data type restrictions and unsupported formats in the XML schema. The owner now can create new read/write permissions on his own databases and tables with the help of appropriate operations (Group 2 and 1). Every other user of a Venice federation has no permissions by default and can't even see a list of supported databases within the GDS.

All the SQL queries can only be categorized by analyzing the query string and associating it to Venice SSO information. This is done with regular expressions. Listing 1 shows two expressions to allocate read/write pulls to tables and databases.

After that the tables are compared to the internally managed control information. Depending on the result access is granted or denied.

4. Evaluation

In this section the Generic Database Service is evaluated within several test scenarios to analyze the behavior in extreme situations and under normal circumstances. All measurements are made with help of the Venice *TimeKeeperService* which provides functions for storing data sets, calculating average values and for visualization. The whole system is tested within a gigabit network whereas the client applications were running on one computer (8 CPUs, 2.33 GHz, 16 GB RAM) and the GDS in combination with the PostgreSQL DBMS on another one (4 CPUs, 3.30 GHz, 4 GB RAM). Furthermore only the two operations *query* and *execute* with several configurations are observed in this paper. In every scenario three values are measured: complete runtime of one query, message transfer and XML conversion time (on both sides) and the JDBC processing time. In the following list the discussed scenarios are introduced in detail:

- **INSERT.** Every prepared statement inserts 12 different datatypes (shown in table 2) into a table (testtable) whereas the *SQLQuery* document has to be configured with help of the SOAP-JDBC-Bridge. The request is transferred by invoking the execute operation of the GDS. Here it is converted and the appropriate JDBC methods are called. Because of the synchronous character the client waits on a positive affirmation and the affected rows.
- **SELECT.** In this scenario the SQL query "SELECT * FROM testtable LIMIT 100" is tested. The system is configured to transfer the result set without splitting it.
- **Split ResultSet.** Here the query is limited to 500 rows and the maximum value set file size is restricted to 800kB and 1200kB. This normally results in a SOAP message with around only 140 rows (800kB) and 211 rows (1200kB).

UPDATE and DELETE statements are not measured because they only affect the JDBC and PostgreSQL components. Per scenario 5000 cycles have been recorded to get significant results. The measurements had to be made under normal load of the computers and network so that external effects could not be excluded completely. Table 4 presents the measurements of the first and second scenario.

	Runtime	Transfer	JDBC
INSERT	15 (27) max 401 ms	12 ms	3 ms
SELECT	287 (400) max 2096 ms	268 ms	19 ms

Table 4. Evaluation: Values of a SELECT and an INSERT statement (complete runtime, data transfer and XML conversion, JDBC processing)

Values in *brackets* characterize the standard deviation whereby *max* stands for the maximum measured value within the 5000 cycles. It can be seen that a significant part of the execution time is used to convert the data to XML and to transfer it over the network. But still the values are adequate for the usage in the Venice federation. Figure 3 presents the measurements and some calculated arithmetic values of the complete runtime (execution time) of queries.

The curve in figure 4 describes a cumulative distribution function (CDF) of the values and shows that 80% of the query are processed under 350 ms. Compared to other systems like JDBC-WS [6] the approach shown here is more performant but it has to be considered that the scenarios may not be comparable. The roundtrip time of queries in OGSA-DAI [10] therefore is nearly the same as the ones in GDS. So the new datatype conversion approach used offers the same performance while it has the advantage of strong data typing, usage of simple message splitting mechanisms and an intuitive usage on the client side. Furthermore small queries only need about 15 ms (INSERT) or 300 ms (SELECT) while OGSA-DAI always needs about 142 ms to parse the perform document which is used to configure a request.

Table 5 presents the measurement results of a SELECT statement with a split result set.

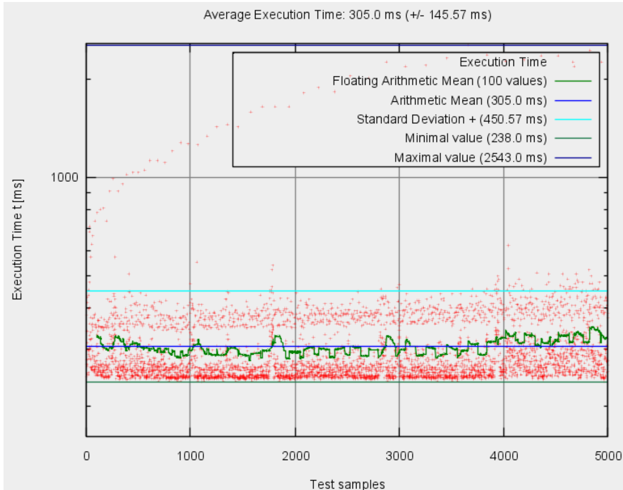

Figure 3. SELECT scenario: Curves with different result values of the 5000 cycles test

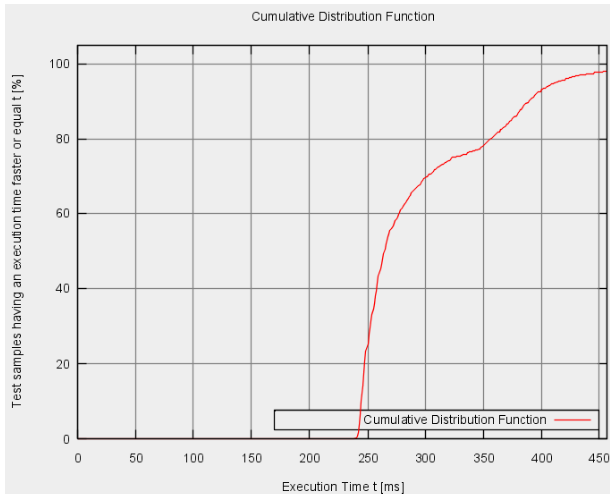

Figure 4. SELECT scenario: cumulative distribution function.

It is noticeable that smaller messages are influencing the latencies and round trip times; but it has to be considered that too many messages lead to a lot of parallel threads and service connections on the client side.

It has to be recognized that the presented values are not fully comparable to the other systems because the evaluation environments are not completely specified within the given papers and the performance depends on the computing power of the test systems, too. In case of OGSA-DAI only a small subset of features is implemented in GDS. OGSA-DAI can access more data sources like for example XML database and therefore supports more query languages than the SQL standard used here. Federated database functionalities are also implemented. On the other side the usage of

SOAP Messages			Runtime	Kind	Transfer	JDBC
File Size	Amount	Rows				
800kB	4	140	1051 ms	first call	375 ms	8 ms
				in parallel	382 ms	-
1200kB	3	211	1788 ms	first call	556 ms	8 ms
				in parallel	402 ms	-

Table 5. Evaluation: Values of select statement with split result set (complete runtime, data transfer and XML conversion, JDBC processing)

OGSA-DAI is not as intuitive as working with the SOAP-JDBC-Bridge accessing the GDS. A programmer has to learn the overall concept of OGSA-DAI to configure its perform document and process the result set. The GDS is optimizing and easy-to-use within the Venice Service Grid.

5. Conclusion

This paper described the development of a Generic Database Service (GDS) for the Venice Service Grid which is designed at the University of Kaiserslautern, Germany. Venice offers an open framework for distributed applications supporting implementation, installation, integration and the usage of services. These features can be used in several domains within a service federation by using a Single Sign-on mechanism and an Information Broker for creating loosely coupled service-based systems.

The GDS is a central component offering access to database management systems with an accurately defined interface (as WDSL) and datatypes (as XML schema). The interface can be used independent of programming language and platform from applications and services within the federation. The following list gives an overview of the supported features:

- **Performing a Query**. A query can be easily performed by configuration of a simple XML document. It offers support for prepared statements and additional features like transactions, manual commitment and advanced caching mechanisms on the server side. The state is stored within the context of that document. After configuration the synchronous *Query* or *Execute* operations of the GDS can be invoked.
- **Type Mapping**. In contrast to other systems like JDBC-WS and OGSA-DAI where a canonical XML document is used for result sets, the GDS is implemented with an accurately defined interface and a corresponding XML schema with strong data typing. In this way tight control over messages is enabled and no additional negotiation between providers is required. On the client side the data can be accessed with JavaBeans while the canonical format has to be transferred within generic types (for example “xsd:any”) and has to be processed as raw XML data.

- **Splitting**. Almost every SOAP engine (like Axis) restricts the SOAP message file size. The GDS precalculates the size and splits messages which then can be transferred in parallel. This way no additional resources have to be used and the latencies are very low.
- **SOAP-JDBC-Bridge**. The SJB as a client side wrapper system is provided as an enhancement to the GDS. It offers a JDBC-like API to control access to the GDS. Additionally it simplifies the parallel transmission of split result sets and optimizes the queries. In this way the service can be used very intuitively and transparently. Existing applications can be easily adapted, too.
- **Rights Management**. A small additional rights management system is provided to close the gap between the Venice SSO mechanism within the service federation and the DBMS. Regular expressions are used to analyze SQL queries and attach them to databases and tables.
- **Performance**. The measurements showed that the datatype conversion chosen in this approach offers the same or sometimes even better performance than other systems. But it still provides the advantage of strong data typing, usage of simple message splitting mechanisms and an intuitive usage on the client side.

It has to be considered that the whole architecture was designed to fulfill the needs of the Venice Service Grid. Therefore the Generic Database Service has to be fast (low latencies) while processing short and small queries, easy to use and existing application should be easily adaptable.

However, this approach – especially the datatype conversion with strong typing – can be used in other scenarios as well and can lead to several advantages (security, interoperability, etc.). The development of a wrapper system for other languages is advisable, too. The future development depends on the requirements of the Venice Service Grid but the interface and XML schema are already prepared to offer additional features like asynchronous query functions or bulk imports.

References

[1] W. Gentzsch, "Response to Ian Foster's 'What is the Grid?'," *Grid Today*, vol. 1, no. 8, Aug. 2002.

[2] C. Kesselman and I. Foster, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, November 1998.

[3] M. Hillenbrand, J. Götze, G. Zhang, and P. Müller, "A Lightweight Service Grid based on Web Services and Peer-to-Peer," in *Proceedings of Kommunikation in verteilten Systemen KiVS (Berne, Switzerland)*, 2007. [Online]. Available: http://dspace.icsy.de/handle/123456789/196

[4] M. Hillenbrand, J. Götze, and P. Müller, "Venice - A Lightweight Service Grid," in *32nd EUROMICRO Conference, Cavtat, Croatia, Aug/Sep 2006*, August 2006. [Online]. Available: http://dspace.icsy.de/handle/123456789/166

[5] M. Hillenbrand, J. Götze, J. Müller, and P. Müller, "A Single Sign-On Framework for Web Services-based Distributed Applications," in *Proceedings of the 8th International Conference on Telecommunications ConTEL (Zagreb, Kroatien)*, Jun. 2005. [Online]. Available: http://www.icsy.de/ archiv/DPArchiv.0134.pdf

[6] E. Dogdu, Y. Wang, and S. Desetty, "A Generic Database Web Service," in *SWWS*, 2006, pp. 117–121.

[7] W. Hoschek and G. McCance, "Grid enabled Relational Database Middleware." In Global Grid Forum 3, Frascati Italy, October 2001.

[8] M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, 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, "The design and implementation of Grid database services in OGSA-DAI: Research Articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, no. 2-4, pp. 357–376, 2005.

[9] M. Atkinson, K. Karasavvas, M. Antonioletti, R. Baxter, A. Borley, N. C. Hong, A. Hume, M. Jackson, A. Krause, S. Laws, N. Paton, J. Schopf, T. Sugden, K. Tourlas, and P. Watson, "A new architecture for OGSA-DAI," *AHM*, 2005.

[10] K. Wang, Y. Xie, S. Li, and X. Wang, "Performance Analysis of the OGSA-DAI 3.0 Software," in *ITNG*, 2008, pp. 15–20.

[11] Globus.org, "Globus Toolkit 4," Apr. 2005, http://www.globus.org/toolkit/.

[12] S. Simkovics, A. Yu, J. Chen, and Z. Fong, "PostgreSQL Webseite/Dokumentation," http://www.postgresql.org/. [Online]. Available: http://www.postgresql.org/

[13] T. Erl, *Serive-Oriented Architecture - Conepts, Technology, and Design*. Prentice Hall, 2005.

[14] L. DeMichiel, "Enterprise JavaBeans Specification, Version 2.1," Sun Microsystems, Tech. Rep., 2003.

[15] A. Foundation, "Apache Axis," 2000-2005, http://ws.apache.org/axis/. [Online]. Available: http://ws.apache.org/axis/

[16] A. Akram, R. Allan, and D. Meredith, "Best practices in web service style, data binding and validation for use in data-centric scientific applications," September 2006. [Online]. Available: http://pubs.doc.ic.ac.uk/web-services-binding-data-styles/

[17] J. Boyer, "Canonical XML Version 1.0," 2001. [Online]. Available: http://www.w3.org/TR/xml-c14n

[18] F. Tartanoglu, V. Issarny, A. B. Romanovsky, and N. Lévy, "Coordinated Forward Error Recovery for Composite Web Services," in *SRDS*, 2003, pp. 167–176.