WS-JDBC

(Web Service - Java DataBase Connectivity)

Remote database access made simple:
http://ws-jdbc.sourceforge.net/

1. Introduction

Databases have become part of the underlying infrastructure in many forms of information systems. Until recently, databases were part of the resource management layer in multi-tier architectures. For this purpose, simple interfaces such as stored procedures (which offer a RPC/RMI interface) and JDBC drivers (which provide direct access using the Java programming language) were enough to embed databases within larger infrastructures. With the advent of service oriented architectures and the need to offer flexible access through the Internet, it has become necessary to revisit the interfaces databases offer to clients. An example of this trend is the prompt support provided by many databases for Web service-enabled stored procedures. Since Web services are specifically designed to support RPC/RMI interactions and IDL-like interface languages, existing stored procedures in databases can be extended to become Web services with very little effort. Useful as it is, this is still not enough since stored procedures limit the interactions between clients and database to predefined queries. To provide support for a wider range of interactions, this paper describes **WS-JDBC**, a JDBC driver implemented using Web services, which allows clients to connect to remote databases using the standard JDBC interface.

The **WS-JDBC** driver has a simple architecture that encompasses (a) server-side classes that implement the necessary parts of the JDBC interface as Web services and (b) client-side classes that are used by applications to invoke those Web service operations. The **WS-JDBC** driver offers the full JDBC API and it is used – with just a few exceptions – as a standard JDBC driver. By simply replacing the JDBC driver with **WS-JDBC**, existing applications can transparently connect to a remote database without having to change their main code, though a few extra lines are necessary to configure the Web service endpoint, etc. The **WS-JDBC** driver manages all parts of the server interactions for clients, from setting up connections, contacting the Web service endpoint, dealing with all necessary XML translations, and all format mappings between objects, relations and XML formatted data.

The challenges behind the design of **WS-JDBC** arise mainly from the latency introduced by the Internet and the overhead of the Web service infrastructure. It is for this reason that **WS-JDBC** uses optimization strategies to encode the relational data so that the price for XML translation and parsing is reduced to a minimum. A progressive interface has also been tested, where large ResultSets can be partitioned and traversed incrementally, so that new data subsets are retrieved while the application processes a previous data subset.

WS-JDBC should be seen as the stepping stone between databases for conventional middleware and databases for more modern and open service oriented architectures. As such, **WS-JDBC** enables very interesting possibilities in terms of system architecture. A first option is to use **WS-JDBC** as the interface to a database application server where clients connect to a remote location, which provides database services. Given the cost and maintenance efforts associated with existing database management systems, this can be a very interesting option in a wide variety of settings. A second option is to use databases directly for *dynamic content delivery networks* (CDNs). CDNs typically replicate pages, but in many cases it is more efficient to replicate the data underneath. **WS-JDBC** could play a central role in such architectures by giving direct access to the database from a number of locations across the Internet. Finally, **WS-JDBC** can also be effectively used to grant database access to mobile devices so that they do not need to keep an open, low level connection (as in conventional JDBC drivers). With **WS-JDBC**, these devices can freely roam among locations and networks and still connect to the same database using a Web service infrastructure.

The design feasibility and performance of the **WS-JDBC** driver are demonstrated through an extensive set of experiments. The system was tested for response time using clients on different locations across the Internet. The behaviour of the system was also analysed with and without optimizations, to illustrate their advantages. Hence, this paper includes: a working design for a Web service-enabled JDBC interface, a detailed performance study of relational data access in Internet settings, and optimizations that make the system more usable in practice. It is expected that the use of **WS-JDBC** will also lead to further optimizations in query processing and optimizations to cope with the network latency.

The rest of this paper is organised as follows: Section 2 describes the background on JDBC and Web Services. The system architecture follows in Section 3 and benchmark performance results are presented in Section 4. Section 5 outlines some applied uses for the framework. Section 6 is the planned future work, before the paper is concluded in Section 7.

2. Background

2.1 JDBC

JDBC (see http://java.sun.com/products/jdbc/) is a Java framework for standardised access to relational data sources. The main package, *java.sql.**, contains classes and interfaces to create and maintain connections, submit queries and process the results. The interfaces show what methods need to be implemented for the API to be complete and usable. An actual implementation of the JDBC API is referred to as a *driver*. Drivers are provided by either database vendors or independent developers and are available for most major database products, including MicroSoft SQL Server, PostgreSQL (see http://jdbc.postgresql.org/), DB2 (see http://www.ibm.com/) and Oracle (see http://otn.oracle.com/tech/java/sqlj_jdbc/). The benchmarks in this paper were performed using the open source driver *jTDS* (see http://jtds.sourceforge.net/).

In the JDBC model, the appropriate driver software needs to be available to clients for them to be able to access a database. The first step in using a driver is to register it with the *DriverManager*. The DriverManager then works as an object factory for *Connections*. A Connection is created by calling one of the DriverManager's *getConnection*(…) methods. The minimum input required for the getConnection(…) call is a database URL in the form: jdbc:*subprotocol*:*subname*://*hostname*/*catalog*, for instance: jdbc:jtds:sqlserver://localhost/TPCW. Connections are opened automatically when created but should be closed explicitly when no longer needed to free up reserved resources. Settings specific to each Connection can also be retrieved and manipulated using the appropriate calls.

A Connection also works as an object factory to create *Statements*, which are used by clients to submit queries to the database. There are three types of Statement. The static version is simply called *Statement* and is created by calling *createStatement*(). The precompiled *PreparedStatement* is created by calling *prepareStatement*(…) and the

statement type for invoking stored procedures, *CallableStatement*, is created by calling *prepareCall*(…). In the two latter cases, the minimum method input is a query string. The Statement types are structured in an inheritance hierarchy, CallableStatement extends PreparedStatement, which extends Statement.

There are several ways to execute statements. The *createStatement*() method does not require any input, which means that – for a Statement – the complete query string must be provided when calling either *execute(*…), *executeQuery(*…) or *executeUpdate(*…). All parameter values must be embedded in this type of query string. For PreparedStatements and CallableStatements, the query string must be provided when the statement is created. Upon creation, the query string contains question marks instead of parameter values. Before the query is executed, each parameter has to be explicitly set to a value. The three execution methods *execute*(), *executeQuery*() and *executeUpdate*() then do not require any input, the query is built within the object (this applies to both PreparedStatement and CallableStatement).

The query output, a *ResultSet*, is either retrieved directly from the executeQuery() call or by calling *getResultSet*(), if execute() was used. The ResultSet contains the rows of columns that resulted from the query execution. It also contains a cursor, initially pointing before the first row, that advances one row every time the *next*() method is called. This is the basic mechanism for traversing a ResultSet. At each row, the methods to get or update a column value can be called, either by supplying the name of the column or its sequence number, starting at 1. There is a specialised get and update method for each of the basic column types, and the *getObject(…)/updateObject(…)* methods can be used if the exact column type is unknown to the client.

2.2 Web Services

Web Services (see http://java.sun.com/webservices/ and [1]) have evolved from object-oriented systems and middleware architectures. The main idea behind Web services is to provide a standardised interface and access method to applications. For most purposes, the relevant technologies associated with Web services are the *Simple Object Access Protocol* (SOAP) [2] and the *Web Services Description Language* (WSDL) [3], which both are used in **WS-JDBC**. Briefly described, WSDL acts as an *interface definition language* (IDL) adapted to the Internet. Through WSDL, an application can publish an API that other applications can use to invoke services made available by the first application. SOAP acts as a transport protocol that can be used to make the interaction machine and network independent. It relies on XML as intermediate data representation and the current specification describes how to transform Remote Procedure Call (RPC) interactions into XML documents that can be sent within SOAP messages using Hyper Text Transfer Protocol (HTTP).

In more detail, to communicate with a given Web service, a client RPC is serialised by converting it to XML, which is encapsulated in a SOAP message. These messages are called *envelopes* in the SOAP terminology, and contain a *header* and a *body*. The header is optional and could, for instance, contain coordination and security information. The body is mandatory and contains the actual data being exchanged, in this case the XML transcription of the RPC. The body is unpacked on the receiving side and converted back into a RPC, which can then be executed. The return values of the call undergo the same procedure when sent back to the requestor.

The API of a given Web service is published using a WSDL document. This is an XML document that describes what operations are available, what input is needed, what output is to be expected and bindings to underlying protocols. Also included is the *endpoint* URL, which is the location of the SOAP engine processing messages on behalf of the Web service in question. This may be located on a separate server from the Web server hosting the WSDL document, but often resides on the same server.

Through the URL to a WSDL document, client stub classes can be generated. These stub classes contain a corresponding method for each of the operations in the Web service API described in the document. Client programs can easily integrate these methods to make use of the Web service operations, whereby the stub classes take care of the call conversion and remote communication transparently.

Methods to be published as Web service operations are implemented the same way as normal methods in a Java program. The compiled classes then need to be made available to the servlet engine – or whatever means is used to run the Web services – integrating them into the engine's environment at start-up. After following the engine's deployment procedure, the Web service is ready to support the operations published in its WSDL document.

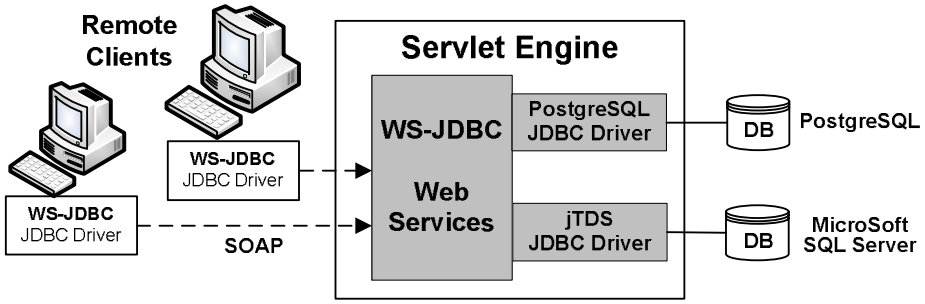

Figure 1. System architecture

3. System Architecture

3.1 System overview

WS-JDBC requires a servlet engine to run. The server-side APIs are deployed and configured to use the JDBC drivers necessary to connect to the databases within the network. No vendor-specific JDBC driver is needed on the client side, instead the custom-made **WS-JDBC** driver connects transparently to the server, and converts the data resulting from the remote calls as needed. The proxy type client objects tunnel calls to their server-side counterparts, which call the appropriate method in the underlying JDBC driver. The options chosen at Connection creation determine which database is being used and the DriverManager chooses the appropriate JDBC driver (see the overview in Figure 1). The client-side **WS-JDBC** driver communicates with the **WS-JDBC** Web services by – as described in section 2.2 – exchanging SOAP messages with the SOAP servlet running on the engine hosting the Web services. The APIs of the **WS-JDBC** Web services contain the methods of the standard JDBC API. Objects created using the underlying JDBC driver are stored in a table, and looked up when a **WS-JDBC** Web service operation is called. The operation is then performed on the object found in the look-up table.

Since objects being transferred through a Web service must be serialised, this affects the ResultSet and all other intermediate representations chosen for the actual data transfer. The good news is that the server side transformation of the ResultSet into the intermediate representation only needs to rely on a few methods; ResultSet.*next()* to traverse the set of rows and *getObject(…)* to fetch each column, which simplifies the design.

The most straight-forward scenario – where the internal row/column structure is an Object matrix (Object[][]) – works without any further changes. This representation has the advantage that no type mappings need to be provided when the Web service is deployed. Independent of the transfer object types, the ResultSet (and all other classes in java.sql.* that are transferred from the Web services) must be available in the client-side **WS-JDBC** driver. The complete interface is provided, hiding the underlying drivers and their differences on the server side.

3.2 Object Categories

There are essentially 4 different categories of classes in the **WS-JDBC** framework: server side specific, transport, the unchanged and a few unique to **WS-JDBC**. The transport objects are client/server hybrids; returned by server side operations and wrapped by client side objects.

The part of the API that is of an administrative nature is implemented as Web services, there are 5 of those: Driver, DriverManager, Connection, DatabaseMetaData and Statement (shared by the 3 statement types). These objects are closely tied to the underlying JDBC driver and are therefore created and kept on the server, hidden behind the deployed Web service interface. Client side objects work like proxies to access those server objects transparently. Method calls made on these clients are tunnelled to the server. Each client object holds the id of its server counterpart, and the corresponding server object is looked up in a table, to perform the method call on it.

One notable consequence of the server side object caching is a more complex garbage collection. If the *close*() method (for Connection and Statement) is not called explicitly, server side objects may linger when client objects, referring to those server side objects, are destructed. Without a special handler for garbage collecting these lingering objects, they would waste an increasing amount of memory space, until the servlet engine is restarted. A minimal access log will therefore be kept for each server side object, which allows the specialised garbage collector to regularly tidy up objects that have not been used for a configurable period of time. This whole procedure is similar to the concept of weak references in Java, introduced in the Java Development Kit (JDK) 1.2.

The second category of classes holds those used as data carriers to transport the results from the server to the client. These objects are serialised, and therefore relatively simple, but are encapsulated by client side objects to offer the proper JDBC API. Examples are ResultSet, *ResultSetMetaData*, *Savepoint*, *Calendar* and other column specific type wrappers.

The third category holds the classes from java.sql.* that are used “as-is”, amd don’t need a custom-made implementation in net.sourceforge.ws_jdbc.jdbc.*, namely *DriverPropertyInfo*, *Date, Time* and *Timestamp*.

The fourth category contains the classes that are unique to **WS-JDBC**, some that are extended and used differently on the server and client: *Blob, Clob* & *TypeConverter* and others like *BinaryConversion*, *BinaryNumber & Context*. These classes are not in the standard JDBC API, but are needed on both the client and server in **WS-JDBC**. Context contains configuration information and BinaryConversion/BinaryNumber supports the binary encoding discussed in a later section.

3.3 Environment

The Apache Tomcat servlet container, v5.0.19 (see http://jakarta.apache.org/tomcat/index.html) and AXIS (Apache's SOAP implementation, see http://ws.apache.org/axis/) have been used to deploy and run the **WS-JDBC** Web services. To deploy a specific method, it needs to be listed in a Web Service Deployment Descriptor (WSDD) file, which the AXIS *AdminService* interprets. Afterwards, the Web service is active and the generated WSDL [2] document is linked on the *AxisServlet* web page.

AXIS provides an application to generate stub classes, called WSDL2Java. This has been used for the **WS-JDBC** WSDL files, thus generating **WS-JDBC** specific stub classes. These stubs have been modified to be flexible, so that the initially hard-coded server endpoint URL now is configurable at call time, by using the Context object. That way, an alternative server offering the **WS-JDBC** Web services can easily be incorporated into the system. This allows the use of **WS-JDBC** to implement load balancing and automatic redirection between database servers as needed. It also minimizes the problem of having a static binding from the clients to the server, which would be a serious design constraint in the case of a large set of remote clients located across the Internet..

Objects that are returned from a Web service need to be specified in the respective WSDD, using so called type-mappings. This ensures that the contents of the objects are serialised properly. The downside to this is that the Java sources for such type-mapped classes (as before, generated when running WSDL2Java) do not contain the original API methods defined on the server side, only the data and some helper methods. This has been worked around by encapsulating a transported object with a client side object, which offers the same API as the corresponding server side object and merely uses its contained object as a data source.

3.4 JDBC Internal Representations

Unfortunately, JDBC drivers offer compatibility at the interface level but not internally. Thus, it is very likely that the internal representation of, e.g., the ResultSet class varies from driver to driver. **WS-JDBC** uses the simplest possible representation: an Object for each Value, an array of Objects (the Columns) for each Row, and an array of Rows for the set of Rows. The following table compares the choice of representation between different drivers.

Table 1. Internal ResultSet representations
(array* refers to the built-in Java array [], not an object)

Driver name	RowSet	Row	Column value
jTDS	array* of	Vector of	Object
PostgreSQL	Vector of	array* of	byte array*
WS-JDBC	array* of	array* of	Object

3.5 Server-side API

To illustrate the server-side API, two of the main services within **WS-JDBC** are discussed in detail: *_Connection_WS* and *_Statement_WS*. _Connection_WS is the more straightforward of the two, and its API encompasses 36 methods, named after their counterparts in the java.sql.Connection interface. _Statement_WS has a wider scope. In order to preserve the inheritance hierarchy among the three types of Statement, the class behind the Web service combines the three interfaces and provides all 153 methods. If the three Statement types had been deployed in three separate Web services, many methods would have had to be duplicated across the services. For PreparedStatement, its own methods as well as those of Statement would have to be in the API of its service. For CallableStatement, its own methods as well as those of the other two types of Statement would have to be included to reflect the inheritance hierarchy.

Both _Connection_WS and _Statement_WS maintain a table of instantiated objects. A Connection is created by calling one of the *getConnection(…)* in the DriverManager, which returns a Connection object from the underlying driver (f.i. in jTDS its full name is *net.sourceforge.jtds.jdbc.TdsConnection)*. This Connection object is given an ID and stored in the internal Connection table in _Connection_WS. The ID is returned to the client and stored in the client-side Connection object. When a method is called on the client-side Connection object, the ID serves as reference and is provided in the input to the corresponding Web service operation. In the method receiving this input in _Connection_WS, the ID is used to look up the object in the table. The method call is then, in this example, applied on the TdsConnection object found in the table. The jTDS driver takes care of the communication with the database and delivers a result for the method call. This result is returned to the client by the _Connection_WS operation.

To illustrate how this is implemented and what the deployment details look like, two Java code snippets of Connection methods and their WSDL definitions follow (the binding, which is similar to the operation definition, has been left out for brevity). The *getConnection(id)* method looks up the ID in the list and returns the stored Connection.

Java code snippet for closing a connection

```
public void close(long id) {

    try {
          getConnection(id).close();
    } catch (Exception e) {…}
}
```

WSDL definition, operation close connection

```
…
<wsdl:message name="closeRequest">
  <wsdl:part name="id" type="xsd:long"/>
</wsdl:message>

<wsdl:message name="closeResponse">
</wsdl:message>
…
<wsdl:operation name="close" parameterOrder="id">
  <wsdl:input name="closeRequest" message="closeRequest"/>
  <wsdl:output name="closeResponse" message="closeResponse"/>
</wsdl:operation>
…
```

Java code snippet for submitting a prepared statement

```
public long prepareStatement(long id, String sql) {

    try {
        java.sql.Connection curConn=getConnection(id);
        java.sql.PreparedStatement newStat= curConn.prepareStatement(sql);
        long newStatID=Statement_WS.addElement(newStat, id);
        return newStatID;
    } catch (Exception e) {…}
}
```

WSDL definitions for prepared statement

```
…
<wsdl:message name="prepareStatementRequest">
  <wsdl:part name="id" type="xsd:long"/>
  <wsdl:part name="sql" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="prepareStatementResponse">
  <wsdl:part name="prepareStatementReturn" type="xsd:long"/>
</wsdl:message>
…
<wsdl:operation name="prepareStatement" parameterOrder="id sql">
  <wsdl:input name="prepareStatementRequest" message="prepareStatementRequest"/>
  <wsdl:output name="prepareStatementResponse" message="prepareStatementResponse"/>
</wsdl:operation>
…
```

The first example, close(), is self-explanatory. It takes the ID of the connection as input and produces no output. The second example, prepareStatement(), shows how the Connection_WS object interacts with the Statment_WS object to add a newly created PreparedStatement to its table of instantiated objects.

3.6 Binary Encoding

One of the weaknesses of XML is the need for tag parsing, which is a costly operation for large documents. In order to increase the speed of translation and parsing – on both the client and the server – and reduce the size of the data being transferred, ResultSets are encoded to a custom binary format in **WS-JDBC**. Note that compression could also be used. However, compression would only reduce the transmission time, not the XML conversion and parsing overhead at both client and server. In practice, compression might even add overhead because of the need to compress and uncompress the data. Another alternative would be to use binary attachments with SOAP messages. This solution helps with the SOAP message transmission but it is not standard (it is supported by AXIS, though) and still requires client and server to agree on the format of the binary representation. Because of this, a custom-made binary encoding was developed. This encoding should not be seen as a standardisation effort but an attempt to explore the inherent overhead of systems like **WS-JDBC**.

Six basic types have been implemented: *String*, *Date*, *Float*, *Long*, *Integer* and *Short*. A binary encoded value of each of these types start with two digits and a letter. This 3 byte over-head is necessary to specify the length (how many of the following bytes is the value) and the type. The 3 discrete types are straightforward, worth noting is that leading zeroes in values are removed to make the format more compact. A small Integer, for instance 256, is shortened from "04i0010" (length 4, type integer, and the value 256 in binary transcribed to 4 characters) to "02i10", before the string is converted to a byte array. This array has the same length as the string. At a first glance this does not look like an improvement, in terms of length, compared to the original value. The byte array representation is still 2 characters longer than the string length of the Integer value. But when taking the tags into account, the minimum length of representing the Integer column value in XML is 10 ("<v>256</v>"), double the size of the binary representation. The size decrease is even more apparent for larger numbers, e.g. "04i1000" is shorter than "16777216", even without counting tag lengths (which is at least 7).

The 3 other types each have specialties that separate them from each other. The time stamp of a Date is simply stored as a Long. The decimal point of a Float is disregarded and its digit sequence converted to a Long. The number of digits after the decimal point is kept in the first length byte. The byte value of a String is zipped to achieve compression. All the byte arrays representing the column values of a row are merged to one long byte array, which can then easily be decoded on the client side. The two first bytes reveal the length of the first value, whereby the start of the second column value is easily found. Iterating like this until the end of the array decodes the byte array back to a row of column values.

The fact that the treatment of XML documents and data formatting is by no means standard and completely specified in existing systems presents a problem. As long as only a single byte array is returned, AXIS can handle the value correctly by automatically applying Base64 encoding. This turns the byte array into a string, which is then enclosed by a single set of XML tags. When an *array of array of byte* is returned, AXIS treats the inner array as any other array, and inserts tags around every byte in that array. This makes the resulting XML document correct but unnecessarily large and also removes the advantage of less tag parsing, which is the main idea behind the described binary encoding. To solve this, each byte array resulting from a row is explicitly converted to a Base64 string on the server, and the method returns an array of this type of strings. In this way AXIS does not add unwanted tags.

3.7 ResultSet Partitioning

Another important aspect that was considered as part of **WS-JDBC** is the management of large result sets. It is often the case that an application will request a large result set and use only a few of the first rows returned. It is also possible to hide some of the cost of bringing a large data set behind the actual work the application has to do in processing subsets of the answer on reception.

A query returning many rows or columns produces large ResultSets, even when these are encoded in a binary format. Executing the entire query and transferring the whole ResultSet both become costly operations, and the

calling client is blocked waiting for the answer. In order to get quicker responses from the server and shorter ResultSet transfer time in **WS-JDBC**, a query can be partitioned into more manageable pieces. Then the first subset of data arrives relatively quickly, and the client can start processing it. Simultaneously, the other parts of the query are executing on the server. When a client command moves the cursor close to the end of the ResultSet subset it is currently working on, the next subset is fetched. To achieve this, the original query is cloned and augmented with a "*where*" clause that sets the primary key limits for each cloned query. These cloned queries can be executed either in sequence or in parallel, in the latter case each query runs in a separate thread.

4. Performance

4.1 Overview

In order to measure the performance of and overhead introduced by **WS-JDBC**, the TPC-W benchmark [4] was used. This benchmark simulates a bookstore application on the web, with 3 different loads or *mixes*: Browsing, Shopping and Ordering, each of them representing different variations of SQL statements.

4.2 Experimental Setup

The benchmark was used to measure and compare the **WS-JDBC** Web service implementation with a setup where the jTDS driver was called directly. Additionally, two remote clients were tested to get an idea of the impact of extra network time on the Internet. The remote clients are located (a) in a different country but on the same continent as the server (Madrid [ES]) and (b) on a different continent than the server (Montreal [CA]). The database and servlet engine ran on two separate servers in Zürich [CH], and the clients used for the local experiments ran on a third computer at the same location. See the client statistics in Table 2 below.

The 3 Web Interaction Mixes ran with the following scopes:

- Browsing Mix: 56623 statements
- Shopping Mix: 67221 statements
- Ordering Mix: 106872 statements

In each case, the relative amount of different statement types fulfils the TPC-W specification for these mixes. Since only the database part of TPC-W is interesting here, realistic sequences of SQL statements were generated with valid test data, to simulate the kind of database interactions the TPC-W setup leads to. The SQL statements were generated using code based on a Java implementation of TPC-W [5]. The database initially contains approximately 288K customers, 10K items and 259K orders.

In this first set of experiments, the statements were executed sequentially and the average response time (and the corresponding standard deviation) was calculated. Before each experiment, the database was re-set to an initial state. The four experiments performed are: access to the database through the jTDS JDBC driver (no Web service and hence the baseline for the experiments) by a local client; access to the database through **WS-JDBC** by a local client (Web service overhead); access to the database through **WS-JDBC** by a client in a different country; access to the database through **WS-JDBC** by a client on a different continent.

Table 2. Client evaluation setups: location & statistics

Client location and setup	Hops	Ping (ms)
Intranet [CH], using the jTDS driver	1	0.2
Intranet [CH], using **WS-JDBC**	1	0.2
Madrid [ES], using **WS-JDBC**	13	41
Montreal [CA], using **WS-JDBC**	19	102

Table 3. Response time measurements (Browsing mix)

Browsing (ms)	[CH] jTDS	[CH] **WS-JDBC**	[ES] **WS-JDBC**	[CA] **WS-JDBC**
Statement	64	109	323	597
	---	62.9	62.2	63.0
Prepared	59	108	414	732
	---	57.5	57.1	57.6
Callable	69	119	460	898
	---	62.9	64.6	62.2

Table 4. Response time measurements (Shopping mix)

Shopping (ms)	[CH] jTDS	[CH] **WS-JDBC**	[ES] **WS-JDBC**	[CA] **WS-JDBC**
Statement	55	97	283	551
	---	53.6	51.6	52.7
Prepared Statement	50	97	407	771
	---	48.5	47.9	48.6
Callable Statement	54	104	440	863
	---	51.8	50.8	52.1

Table 5. Response time measurements (Ordering mix)

Ordering (ms)	[CH] jTDS	[CH] **WS-JDBC**	[ES] **WS-JDBC**	[CA] **WS-JDBC**
Statement	56	86	258	419
	---	54.3	52.8	53.2
Prepared	52	92	356	682
	---	51.1	51.2	51.6
Callable	54	94	386	768
	---	53.1	52.8	52.6

4.3 Performance Results

Tables 3, 4 and 5 show the results for the three TPC-W mixes, each from the four locations and using the three Statement types. Note that, in our experiments, the more reads there are, the more expensive the load. This is why the browsing mix (with a very low percentage of updates) exhibits the slowest performance. In all tables, the upper figure for each type of statement shows the overall time and the lower figure the time at the database server. This gives a rough indication of the actual network and infrastructure overhead. As can be seen in the tables (column 2 & 3), the overhead imposed by the Web services infrastructure varies between 30 and 50 ms of additional response time when compared to using the driver directly, without a Web service layer. Yet, for the two remote clients (column 4 & 5), this is a relatively small overhead compared to the network latency. This is clear when the response time for a local client is compared with the response times for remote clients. In both cases the Web service overhead is paid but the dominant factor is Internet latency. Figure 2 shows the Browsing mix results (from table 3) in diagram form. The other two mixes show a similar pattern and are left out for brevity.

The response time measured in all the experiments is not narrowed down to only the actual query execution. It includes the statement creation and, for the set of queries that otherwise would return the entire contents of a table, the setting of an upper limit on the number of rows returned, by calling *setMaxRows*(…). The TPC-W query that returns the 50 first entries of the Bestseller list is an example of where this is necessary. In the case of CallableStatement, the same stored procedure is called for every query. The only input to the stored procedure is a complete query string, like for Statement. The input parameter is set by calling *setString*(…). The query is executed inside the stored procedure, which means overhead compared to Statement. When using PreparedStatement, *setObject*(…) needs to be called once for each parameter that needs to be set in the query string. The benchmark mixes contain queries that do not have any parameters, but most have one parameter and only a few have more than one. In short, CallableStatement always has one server round-trip more than Statement, and PreparedStatement sometimes has 1 to n round-trips more.

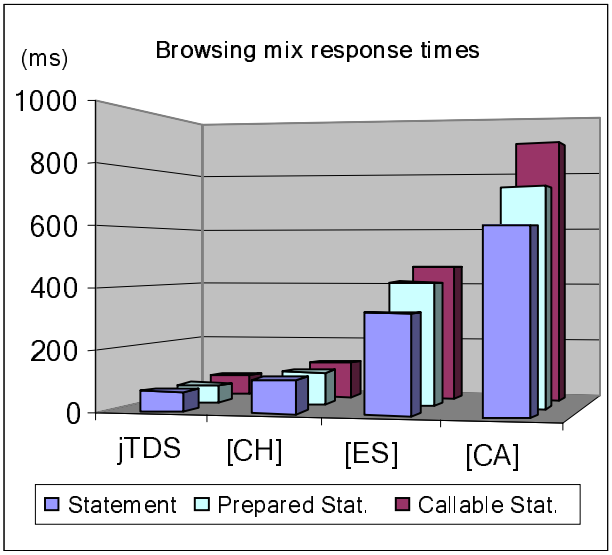

Figure 2. Browsing mix response times

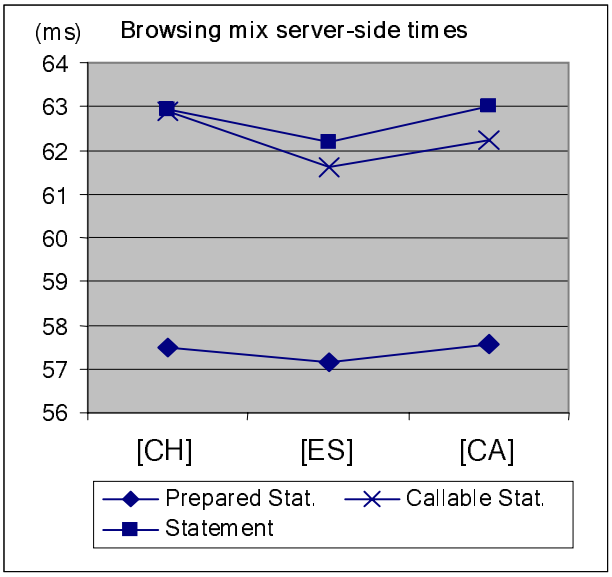

Figure 3. Browsing mix server-side times

In the light of these differences, the Browsing mix response times can be interpreted as follows. When calling the jTDS driver, the results for each statement type are comparable to what is measured as the server-side times for the other client location experiments. CallableStatement breaks the pattern slightly by having a higher overhead (6 ms compared to 1-2 ms), but this was not the case for the other two benchmark mixes. For the local client using **WS-JDBC**, the higher round-trip average for PreparedStatement brings it up to level with Statement. CallableStatement is notably slower, because of the stored procedure execution overhead. The two remote clients show the differences more clearly. The local advantage of PreparedStatement is lost due to the increasing network latency, because of the extra round-trips. Statement is clearly more efficient remotely, and CallableStatement the least efficient. The same line of reasoning applies for the other two mixes.

For comparison, Figure 3 shows how much of the response time is spent on the server. This time stays stable independent of the client location. Again, the other two mixes show a similar pattern and are left out for brevity. When the numbers in Figure 2 and 3 are compared, it becomes clear that the main bottleneck is the network latency.

4.4 Effect of Binary Encoding

To minimize the overhead associated with XML, the binary encoding explained above was realised and a number of experiments that test how the system behaves were performed. Two comparisons were made between the use of the normal XML representation and the binary encoding: the response time and the message sizes. To get a fair mix of column types, four tables from the TPC-W benchmark database were used: *cc_xacts*, *customer*, *item* and *orders*. Each contains at least one column with char/varchar, float, int and date/timestamp. The first 5000 rows were selected from each table, and the average response time calculated for repeated executions.

The results of this experiment can be seen in Figure 4, which shows the ratio between each experiment and the system without binary encoding. The experiments were done for each of the 4 chosen tables, which were queried from each of the client locations (same as above). Ratios greater than one indicate that binary encoding is faster than the system without binary encoding. As the graph shows, the gains provided by binary encoding are significant but can be eclipsed by the network latency. This is why the ratios sink as the network distance between the client and the server increases. A local client gets results at least five times faster with binary encoding. The remote clients get results at least twice as fast. This is because binary encoding minimizes the overhead of XML parsing but not necessarily the delay in getting answers from the server.

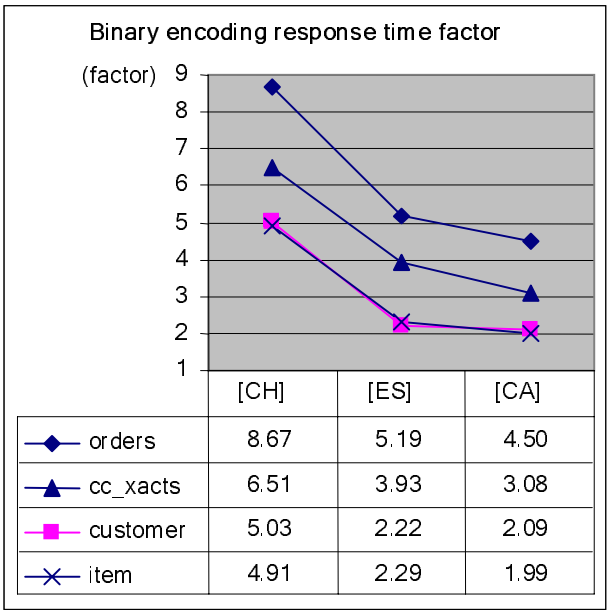

Figure 4. Binary encoding response time factor

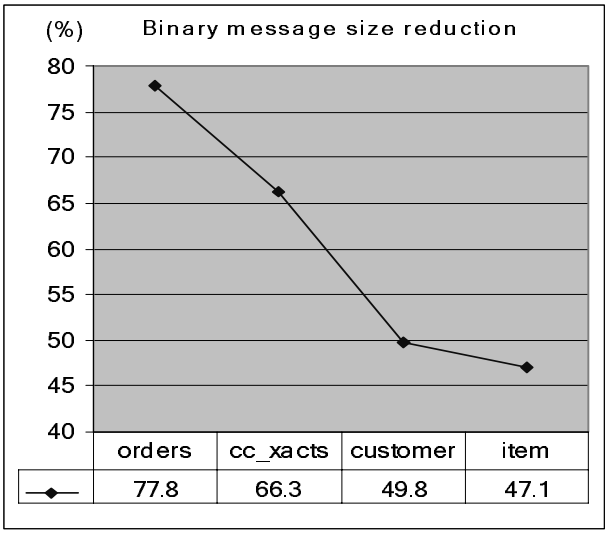

Figure 5. ResultSet partitioning results

For completeness, Figure 5 shows the ratio of reduction in message size when using binary encoding and when not using it. How much compression is achieved depends on the contents of the table. The rate of reduction reflects the relation between numerical and string columns in the tables. For instance, the orders table only contains 2 short varchar columns and 9 numerical, while both the customer and item table have many varchar columns, including one of length 500. In all cases, however, the reduction in message size is at least 47 %, a significant gain in practice.

4.5 ResultSet Partitioning Setup and Results

As discussed above, the second major performance problem in the considered settings is network latency. Since network latency is outside of the system, one could simply take it for granted and assume there is not much that can be done. In practice, however, the ResultSet partitioning technique explained above can be used to hide latency behind other operations.

For this experiment the orders table in the TPC-W benchmark was used. The table contains 259243 rows, and these were successively retrieved in a growing number of subsets. The first experiment queried all the rows, the next four experiments were divided in 2 to 5 subsets, and the last experiments in 10 subsets. For each of the six experiments plotted in Figure 6, the query "*select * from orders*" was cloned and augmented with a "*where*" clause to set the lower and upper bound of the table's primary key, *o_id*. For instance, "*select * from orders where o_id>0 and o_id<=25925*", for the first subset when using 10 subsets. In the experiments sequential and parallel execution of the subset retrieval is compared. With an increasing number of subsets, the parallel case improves. The smaller ResultSets create a smaller amount of simultaneous objects and therefore need less memory space. In the case of only one partition, the retrieval of the entire set of rows saturates the CPU and leads to swapping of virtual memory, which makes that alternative relatively slow.

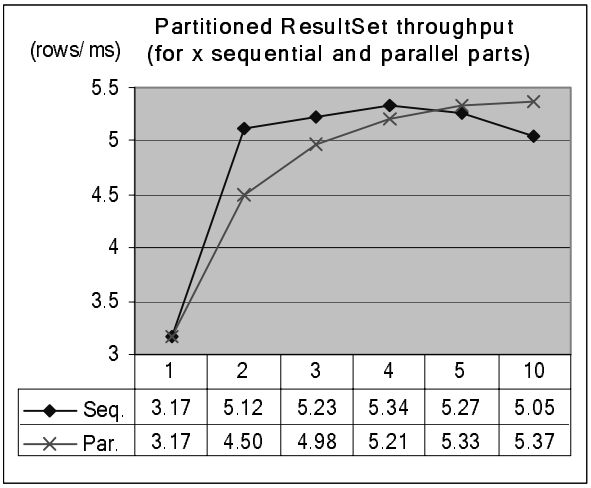

Figure 6. Sequential and parallel throughput

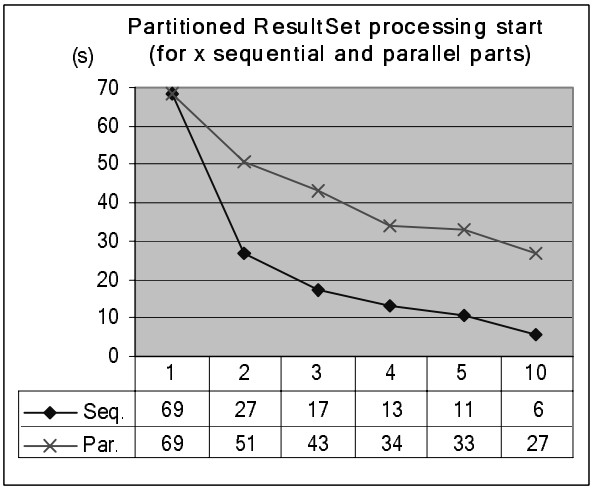

Figure 7. Sequential and parallel processing start

Figure 7 shows another view of the same experiments, namely how much time (in seconds) it took before the first subset was ready for processing by the client. Note that in the first experiment, the SOAP document containing all the rows is larger than 35 MB in size, and this is the reason why it takes more than a minute. The sequential alternative is much faster in all the other cases, since it is only executing one subset at a time. The parallel alternative will also be tested with remote locations, in which case its advantages should be visible.

5. Applying the Framework

The standardised API offered by **WS-JDBC** can help make database administration in big companies easier, especially in homogenous environments. Effectively, the framework acts as a portal to the databases available on the intranet. When new JDBC driver versions become available, they only need to be updated on the Web server where the **WS-JDBC** servlet engine is running. The database users do not need to download and install the updated driver, they do not even have to know or notice that a new version exists.

Also, the fact that client implementers use the general **WS-JDBC** framework allows the finished client to be distributed without specific driver software. Through the flexible configuration of the server endpoint, the different stages of the development cycle are easily supported: application test, integration test, system test and production. The cascading update mechanism described below even enables test sequences to be applied to several development stage versions in the same run.

WS-JDBC is an attractive alternative for organisations wanting to make their data sources public, since less specialised knowledge is required. The deployment of the **WS-JDBC** Web services can be achieved with a carefully described installation procedure, so there is no need to know the specifics about how to create and publish Web services, e.g. how to deploy customised objects. Also principles like object/relational mapping do not need to be taken into account. This is left up to the client to decide, what is best suited for the application integrating the database access.

The most obvious way to extend the **WS-JDBC** framework is to enable it to handle replication. Database replicas in dispersed locations can function either only as alternate storages (the databases are equivalent in structure, but not necessarily in contents) so that a backup location can take over when the primary is down, or as full replicas, where each statement being executed on a server is mirrored to identical databases at additional locations.

A number of different scenarios in the replication realm are of interest, both "lazy" and "eager". In lazy replication changes are propagated after the transaction updating the primary database has committed, in eager replication the propagation takes place within the transactional context [6]. These alternatives represent the »when« dimension, and the following the »where« dimension.

Replication can be triggered either on the client or on the server. Replicas can be specified by configuring additional **WS-JDBC** endpoints. Every client call to the master Web service is then repeated for all the configured replicas, each in a separate thread (Figure 8). Alternatively, a call received on the server side can be forwarded to all the replicas or only to the next in a chain of cascading calls (Figure 9). As the server in this case actually acts as a client, all the software is distributed in the same package. This also allows for other call propagation topologies.

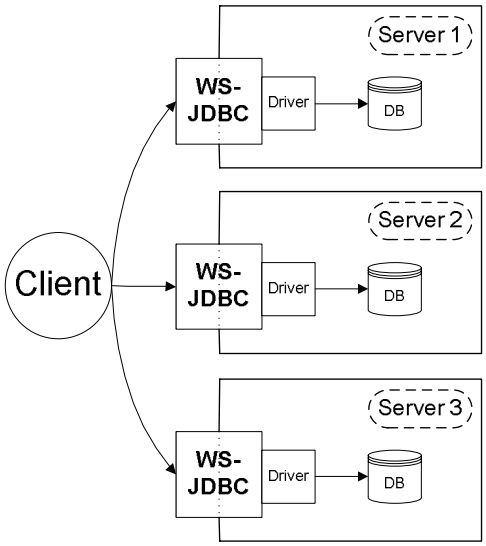

Figure 8

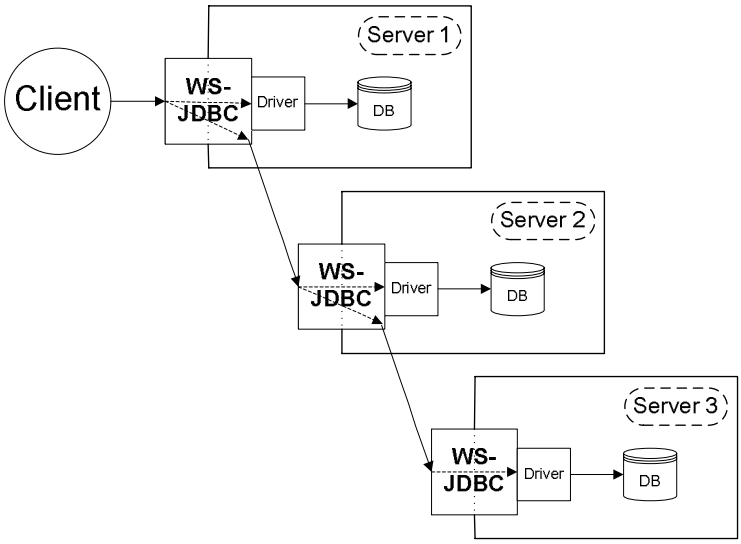

Figure 9

To guarantee consistency, all calls should be logged. This allows for statements to be re-tried after a server in the chain was temporarily out of service. Other alternatives include partial replication, both through vertical and horizontal table splitting. Further ideas concerning consistency are being formulated and will be pursued.

The functionality has many uses on hand-held devices; e.g., logging newly developed applications that are being tested, periodically storing the position of GPS-enabled devices or temperature and other weather data from remote measurement stations. This makes the data instantaneously available in a central relational data source, and therefore also reachable by other mobile peers. Centralising the collection of data like this makes it easier to evaluate statistics on them and to monitor running systems.

One of the main advantages of this new functionality is that it allows data from a remote location to be actively secured. If a remote device fails, all data up to the point of failure has been stored safely. Only the data being processed at the time of the crash is lost. Users of handheld devices may not be used to doing periodical backups, and will therefore benefit from having their data secured.

6. Future Work

The first version of the framework is a "pure" client/server solution, meaning that the server type classes are on the server side in their entirety, and the client type classes are transported to the client in their entirety. To increase performance and avoid unnecessary network traffic, more of the server side logic can later be shifted to the client

side, making the proxy objects more intelligent. For example getCatalog() and setCatalog(str) in Connection. While the 'set' call must be tunnelled to the server, the 'get' can rely on a local (client side) value. When the Connection is created, the server side getCatalog() can be called to get the default setting, which is stored in the client. On each 'set' the local value is updated, which enables the 'get' method to deliver a reliable, mirrored value of the current setting without an unnecessary server round-trip.

Different alternatives to fine-tune the ResultSet partitioning will also be explored, f.i. experiments with server-side settings to optimize the partitioning. The procedure implemented so far, where the number of wanted subsets is chosen, will be compared to alternatives where a limit for an absolute number of rows, or a percentage of the total number of rows, is set. Server logic should then examine the contents of a table before deciding the best partition size, and clone the original query. Each ResultSet subset can be stored in the main memory of the servlet engine, whereby the available resources limit how many query results can be kept simultaneously. The subsets will then be removed as soon as they have been requested by the client and transferred. This approach is obviously a prime candidate for dynamic tuning once the application load can be examined and analysed.

7. Conclusions

JDBC enhanced with Web services has been shown to be a valuable contribution towards making databases available everywhere. Providing a tunnel through firewalls to allow access to databases boosts the availability. The average response time overhead for using the **WS-JDBC** framework is relatively low in an Internet setting and by far outweighed by its usefulness. Binary-encoded ResultSets, and the partitioning of large sets of rows, further increase performance, both in speed of processing and the smaller amount of data being transferred.

With an increasingly widespread use of mobile devices, **WS-JDBC** offers transparent access to databases over the Internet. The use of standardised components will allow **WS-JDBC** to profit from future extensions in the Web services field. The **WS-JDBC** framework has many potential uses, the prime example is for replication.

8. References

[1] Alonso, Gustavo, Casati, Fabio, Kuno, Harumi and Machiraju, Vijay. "*Web Services: Concepts, Architectures and Applications*". Springer-Verlag, Berlin 2004 (ISBN 3-540-44008-9)

[2] W3C. "SOAP Version 1.2 Part 0: Primer". http://www.w3c.org/TR/2003/REC-soap12-part0-20030624/

[3] W3C. "Web Services Description Language (WSDL) 1.1". http://www.w3.org/TR/wsdl

[4] The Transaction Processing Performance Council. "TPC-W, a Transactional Web E-Commerce Benchmark". http://www.tpc.org/tpcw/

[5] Lipasti, Mikko H. "Java TPC-W Implementation Distribution (of Prof. Lipasti's Fall 1999 ECE 902 Course)". http://www.ece.wisc.edu/~pharm/tpcw.shtml

[6] Gray, Jim, Helland, Pat, O'Neil, Pat and Shasha, Dennis. "The Dangers of Replication and a Solution". In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173--182, Montreal, Canada, June 1996.