Embedded SOAP Server on the Operating System Level for Ad-Hoc Automatic Real-Time Bidirectional Communication

Thomas B. Hodel, Florian Specker and Klaus R. Dittrich
University of Zurich, Dept of Information Technology, Winterthurerstr. 190, CH-8057 Zürich, Switzerland
{hodel, dittrich@ifl.unizh.ch}, specker@icu.unizh.ch

ABSTRACT
Applications using SOAP messages for communication purposes invoke one or several web-services and normally receive a return message for each call. In addition to this, our solution proposes that such applications automatically configure an embedded SOAP server on the operating system level so that a specified server can send an independent SOAP request to this application. The result is a generic framework for an ad-hoc automatic real-time bidirectional communication using/over SOAP messages. In this paper we also describe our concept of extending the functionality of today’s applications and operating systems in order to realize an ad-hoc automatic real-time bidirectional communication using SOAP via HTTP. Finally, we take a look at a concrete implementation of an application which uses SOAP in this customized way.

INTRODUCTION
When thinking about distributed systems, the first thing that comes to mind is the “trench” that lies between the single components of systems. This trench will inevitably be an obstacle to interprocessing communication.

The obvious need for calling remote methods to build a distributed system is the origin of some very different approaches to communication across component borders [6,2]. Most of them are aimed strictly at a specific problem and/or a specific environment. This prevented the traditional protocols from being used in a wide range of scenarios; SOAP has the potential to succeed where their predecessors have failed. Its pros (loose coupling, late binding, platform-, language- and vendor-independence, freely available specification and documentation, easy to implement) make it suitable for many applications, while its cons (resource-intensiveness, no native security model) are not enough to prevent it from being used in most of them.

APPROACHES
If a client needs bidirectional communication, SOAP does not provide it with a solution. Such a communication can be described as “bidirectional, synchronous as well as asynchronous”. While the need for bidirectional communication (in the sense of sending messages in both directions) is obvious in itself, the need for synchronous as well as asynchronous communication can easily be explained. The Remote Procedure Calls are synchronous in the sense that one endpoint will wait for a call to be finished before sending the next call, but are asynchronous in the sense that the application server or client may initiate a transmission at any time (provided that the previous call of the same endpoint has been completed by that time).

To meet these needs, a SOAP server is needed at both endpoints. This does not comply with the traditional Client-Server model, but is necessary for the “server” to be able to send a message to the “client” without first having to wait for the “client’s” request. The result is similar to what is known as “peer-to-peer” [3,7], where each node acts as the client and server at the same time.

In order to provide the application client with the possibility of receiving SOAP messages, two solutions are feasible: the client either embeds its own protocol server or registers itself at an external protocol server which is already installed on the same machine.

EMBEDDED SOAP SERVER ON THE OPERATING SYSTEM LEVEL
An embedded SOAP server on the operating system level would enable bidirectional communication via SOAP and represent an additional solution, which one could investigate more closely. A possible scenario for such a SOAP Server Component (SSC) may look like Figure 1.

The operating system ships with a HTTP server and a SOAP router. Applications may register the methods that need to be callable via SOAP using the SOAP server API, if they provide a WSDL document containing a description of the services as far as the application can determine them (i.e. data type and the number of parameters, return values, etc.).

The SSC would register the application with its methods, completing the parts of the WSDL document which cannot be controlled by the application (e.g. the Transport Protocol, the Port numbers or the method names - since there may be two processes registering a method with the same name, the SSC must be able to distinguish them properly). The SSC should generate WSDL documents on the fly to allow dynamic registering and deregistering; furthermore, it should be able to generate WSDL documents containing only the methods provided by a specific application, especially when there is a significant number of applications that have registered methods. If abnormal termination of registered applications that leads to invalid register entries is an issue, the registrations process could be designed as a form of leasing, as known from DHCP. Registrations would only be valid for a specified amount of time, after which they would have to be renewed in order not to become purged.

If being independent of the SOAP binding is desired, the SSC should be capable of generating SOAP responses even when communicating via an SMTP binding. This should be done transparently to the applications registering in order to achieve a true independence from the binding the SSC uses.

A Routing Table would allow an incoming message to be passed to the process of the application which registered itself at the SSC for that.
EVALUATION

A method of communication established by components using SOAP, and realizing the described automatic bidirectional communication based on the concept of an embedded SOAP server on the operating system level, is a crucial service for business. As example we are offering this service within a collaborative word processing system [5]. Therefore we begin by briefly introducing the underlying concept.

TeNDaX is a Text Native Database eXtension. It enables the storage of text in current databases in a native form so that editing text is finally textually correct. We understand the following: writing and deleting text (characters), copying & pasting text, defining text layout & structure, inserting tables, pictures, and so on. All the actions regularly carried out by word processing users. With 'real-time transaction' we mean that editing text (e.g. writing a character/word, setting the font for a paragraph, or pasting a section of text) invokes one or several database transactions so that everything which is typed appears within the editor as soon as these objects are stored persistently. Instead of creating files and storing them on a file system, the content of documents is stored in a special database, which enables very fast real-time transactions for all editing processes.

The database schema and the above-mentioned transactions are created in such a way that everything can be done within a multi-user environment, as is known within the database technology field. As a consequence, many of the achievements (with respect to data organization and querying, recovery, integrity and security enforcement, multi-user operation, distribution management, uniform tool access, etc.) are now, by means of this approach, also available for word processing.

TeNDaX proposes a radically different approach, centered on natively representing text in fully-fledged databases, and incorporating all necessary collaboration support. Under collaboration support we understand functions such as editing, awareness, fine-grained security, sophisticated document management, versioning, business processes, text structure, data lineage, metadata mining, and multi-channel publishing - all within a collaborative, real-time and multi-user environment.

Collaborative Editor System

First of all, let us introduce the design and implementation of the collaboration support part of the application. The whole process begins with the creation of a WSDL document describing the method offered by the client, and how to call it. This document is generated according to the actual settings.

The server authenticates and registers at the server, handling the WSDL document created in the step above. The server adds the client to the list of registered clients. Since there might be clients who communicate using protocols other than SOAP, this list must contain additional information, such as the protocol the client expects, as well as further details. In this case, all of these further details are described in the WSDL document. The client then sends the WSDL document to the server just received. For now, the server saves it to be able to communicate with the client in the future. The server is now registered and ready to work.

Every action which the user of our client takes is reported to the server. If the user has permission to carry out the changes he tries to make, the server grants them, and it informs the other clients to whom the changes are relevant. If the user is not permitted to
make these changes, the client is informed about the illegality by
the server, and may then inform the user about it.
6. After the user has completed his work (e.g., he quits the client), the
client deregisters from the server and stops its embedded HTTP
server. After that, it can exit cleanly.
7. Upon the clients deregistration, the server removes the clients
entry from its list of registered clients and deletes its WSDL
document.
8. After deregistration of the last registered client, the server may be
stopped.

The Collaborative Editor
The client must be able to (de)register itself at the server, to open, close,
create and delete documents, and request the committing of the changes
the user tries to do to an opened document. All of these tasks can be
carried out using the classic Client-Server model [4]. For SOAP com-
unication, any existing library can be used, and quite a lot of them offer
SOAP client functionality. However with regard to the next require-
ment, the classic Client-Server model fails: the client must be notified
if someone else’s changes on a document opened by the client have been
committed. This is the reason why we also need to have a SOAP server
on the client side as well.

Collaborative Editor Server
The server must be able to (de)register the clients, to process their
requests, and to notify the affected clients about committed changes on
documents. SOAP support on the server side is no big thing.

Error Handling
Using SOAP for the Client-Server communication not only introduces
new possibilities and flexibility, but also new sources of possible errors.
Several things can go wrong when trying to send a SOAP message from
one endpoint to another. Some of these problems can be solved by the
client (or, less favourably, by the user) on its own, so good Error Handling
constitutes a large part of the usability of a client.

If, for example, the other endpoint is unreachable, the problem could
be anywhere between the client and the server. A router between them
may have crashed, or a network cable may have been plugged. The
application client can’t do much in such a situation, but the user possibly
could, if the client tells him what to do (check cables, check the
connection state, check the connection settings, etc.). If other network
problems such as timeouts due to network overload, or connection resets
due to a server crash occur, the client could at least display an error
message explaining what happened, and assist the user in calling
Technical Support to solve the problem.

Nasty things, like internal errors of the HTTP servers or configuration
errors, could be detected automatically by each endpoint, by calling a test
method. If, for example, the client notices that its embedded HTTP
server is not responding, it could try to restart and reconfigure it to solve
the issue. If that doesn’t help, it could still ask the user for help, and send
an exact error message to the developers.

As SOAP is a stateless protocol, endpoints will not notice a connection
failure or an abnormal termination of the communication partner until
they try to send a message. Since it might also be useful to notice
problems during inactivity, the concept of Heartbeats [1] could be
implemented. The client, as well as the server, sends control messages
to each other at defined intervals; if they have not received such a
control message for a while, they know that the other endpoint has
terminated abnormally, or that a connection error has occurred. If the
server registers the client’s Heartbeats as missing, it automatically
deregisters the client. If, on the other hand, the client registers the
server’s Heartbeats as missing, it should try to reconnect; if this fails,
it should check if the server can be pinged, so that it can give the user
accurate information about the problem.

While the errors described above should be the most likely, it is possible
that others will occur.

Proof-of-Concept
The Exchange Component is only a small part of the client. The task
was to write an implementation of the interfaces in the package
tdb.sys.exchange.

Client
The most important object of the SOAP Communication Layer is
SOAPServer in the server package, the class to which all the calls to
SOAPExchange are forwarded. It acts as a proxy of the application
server, hiding all the details of the communication from the application
client. This is the place where the SOAP methods on the application
server are called.

The SOAP server in the application client is implemented using Jakarta
Tomcat and Apache Axis. The service itself is implemented in a single
class. All this does, is to push the incoming message into the
TDBUpdateMessageQueue after removing its brackets. To allow
UpdateListener to get a reference on TDBUpdateMessageQueue, Up-
date became a Singleton, ensuring that it will only be instantiated once,
and that every object running in the same Java VM can easily get a
reference on it.

Server
To minimise the number of complex types to be sent across the wire,
methods which expect “unnecessary” complex types have been wrapped.
Only the ID of these objects is passed to the wrapper method, which
fetches the appropriate object to pass it on to the original method.

The last change on the server side was to distinguish between clients
connected via Caché JDBC and clients connected via SOAP to inform
them about changes in opened documents. The clients connected via
Caché JDBC are informed as they were previously, while the ones
connected via SOAP are informed by the SOAPUpdater: This class calls
the “processMessage” method of the SOAP server embedded into the
client.

Automatic Bidirectional Communication
The following parts trace in detail the chain (see Figure 3) of calls caused
by a character insertion, in order to understand the whole process more
fully.

Startup
On application startup, SOAPExchange is instantiated and initialised.
It initialises the SOAPServer, which creates a new session on the server
(NewCSessionByID, the wrapper method for NewCSession, will set the
session’s “ClientConnection” attribute to “SOAP”), and Update. Up-

Figure 3. Chain of Calls Caused by the Insertion of a Character

Copyright © 2005, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.
date instantiates the classes needed for the update system, and then starts Tomcat to wait for messages about changes in opened documents.

Operations: The behaviour of the file operations (creating, deleting, opening or closing) have remained the same as in the original implementation of the Collaborative Editor client [4].

The process of inserting a character into a document is not trivial, since the change has to be propagated to other clients that have the same document opened. This process in the SOAP implementation differs remarkably from the one in the JDBC implementation.

Editing a Document: The user presses a key, and the insertChars() method of SOAPExchange is called on. It is passed on to SOAPServer, the class that now calls InsertCCharsByById() on the server via SOAP. InsertCCharsByById() is the wrapper method for InsertCChars(); it opens the CDocument and the CSession objects with the IDs passed from the client and calls the traditional InsertCChars(). If everything is all right, InformInsert() from the ServiceRoutines is called, which starts a new thread with GenericMessage() to inform the other clients about the change. For the clients connected using SOAP, this is done by SOAPUpdater.sendMessage(). It calls the processMessage() of the client’s UpdateListener, the one that is running as embedded SOAP server. The SOAP-specific part of the process ends with the pushing of the received message into the TDBUpdateMessageQueue.

Shutdown: Unlike the JDBC-based implementation which uses stateful connections, the SOAP implementation doesn’t need to close any existing connections. The Shutdown process is reduced to deleting the existing session object from the server, together with “DocumentSessions” that might still exist at that time.

Benchmarks

When comparing the behavior of the Collaborative Editor client connected via JDBC with one connected via SOAP, the overhead introduced by SOAP notably delays the real-time bidirectional communication between the client and the server. In order to be able to estimate the performance loss caused by this overhead, we have indicated the time taken by some regularly used methods in the table below.

As illustrated, JDBC outperforms SOAP by a factor of 10.7 to 90.5. This is the price to be paid for the advantages SOAP has to offer. The figures are put into perspective if we consider that opening and closing a document are not used as often as reading or writing a character, where the factor only ranges from 10.7 to 27.4.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Cache JDBC</th>
<th>SOAP</th>
<th>Performance Loss Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open CDocument</td>
<td>(3,4,9)</td>
<td>(80,194,467)</td>
<td>48.5</td>
</tr>
<tr>
<td>Close CDocument</td>
<td>(2,2,9)</td>
<td>(167,181,231)</td>
<td>90.5</td>
</tr>
<tr>
<td>GetCChars</td>
<td>(3,19,57)</td>
<td>(173,203,234)</td>
<td>10.7</td>
</tr>
<tr>
<td>InsertCChars</td>
<td>(9,10,24)</td>
<td>(217,274,833)</td>
<td>27.4</td>
</tr>
</tbody>
</table>

[JDBC, SOAP: The numbers printed are milliseconds (minimum, median, and maximum) of 10 measurements; Performance Loss Factor: $\frac{\text{med(JDBC)}}{\text{med(SOAP)}}$]

CONCLUSION

The implementation illustrates the advantages of SOAP: the tight coupling between the client and the server has been loosened, and the already existing platform independence has been upgraded to language independence. The Collaborative Editor is still usable, although it reacts with a small delay. Of course, the overhead introduced with SOAP restricts the scalability of TeNDaX.

To improve the usability of the SOAP-enabled operating systems for ad-hoc automatic real-time bidirectional communication the primary target will be user interaction and failure tolerance.

REFERENCES

Related Content

Impact of PDS Based kNN Classifiers on Kyoto Dataset
[www.irma-international.org/article/impact-of-pds-based-knn-classifiers-on-kyoto-dataset/233598](www.irma-international.org/article/impact-of-pds-based-knn-classifiers-on-kyoto-dataset/233598)

Ubiquitous Teachers’ Training and Lessons Learned with the uProf! Model

Securing Stored Biometric Template Using Cryptographic Algorithm
[www.irma-international.org/article/securing-stored-biometric-template-using-cryptographic-algorithm/214968](www.irma-international.org/article/securing-stored-biometric-template-using-cryptographic-algorithm/214968)

Creating Believable and Effective AI Agents for Games and Simulations: Reviews and Case Study
Iskander Umarov and Maxim Mozgovoy (2014). *Contemporary Advancements in Information Technology Development in Dynamic Environments* (pp. 33-57).
[www.irma-international.org/chapter/creating-believable-and-effective-ai-agents-for-games-and-simulations/111604](www.irma-international.org/chapter/creating-believable-and-effective-ai-agents-for-games-and-simulations/111604)

Parallel and Distributed Pattern Mining
[www.irma-international.org/article/parallel-and-distributed-pattern-mining/251898](www.irma-international.org/article/parallel-and-distributed-pattern-mining/251898)